Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 133 (2005) 21–39
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

From Distributed Memory Cycle Detection to Parallel LTL Model Checking [1]

J. Barnat, L. Brim, J. Chaloupka

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

In [2] we proposed a parallel graph algorithm for detecting cycles in very large directed graphs distributed over a network of workstations. The algorithm employs back-level edges as computed by the breadth first search. In this paper we describe how to turn the algorithm into an explicit state distributed memory LTL model checker by extending it with detection of accepting cycles, counterexample generation and partial order reduction. We discuss these extensions and show experimental results.

Keywords: LTL model checking, breadth first search, distributed memory

1 Introduction

The model checking [10] became one of the most frequently used formal methods in the field of system verification. In general, the model checking question asks whether an abstract model of the verified system satisfies desired properties that are expressed by means of temporal logics [16].

Unfortunately, all the known algorithmic solutions to the model checking problem suffer from large space requirements needed to answer the model checking question. These requirements are caused by the fact that the size of the state space of a model grows exponentially with the number of components in the system. Several attempts to address the state explosion by exploiting the aggregate memory of a network of workstations appeared recently (see e.g. [18,14,12,5,6]).

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

doi:10.1016/j.entcs.2004.08.056

In this paper we consider linear temporal logic (LTL), a major logic used in formal verification. It is known for very efficient sequential algorithms based on automata and successful implementation within several verification tools. All the existing explicit state distributed memory approaches to LTL model checking ([4,7,1,15,9]) are known to have various disadvantages and shortcomings that prevent each individual algorithm from being considered to be the clear winner. Therefore, the research on distributed memory LTL model checking remains a challenging and open task.

The optimal sequential solution to the LTL model checking problem is based on the depth first search (DFS) traversal of the state space, in particular the *postorder* as computed by DFS is crucial for cycle detection which is the core problem in LTL model checking. However, when exploring the state space in parallel, the DFS postorder is not generally maintained due to different speeds of involved workstations. As a consequence, those algorithms that parallelize DFS based LTL model checking suffer from additional technical machinery that is necessary to maintain the DFS postorder in the distributed memory environment. In [2] we proposed a parallel graph algorithm for detecting cycles in very large oriented graphs distributed over a network of workstations. The main twist is that our new algorithm does not employ DFS postorder for cycle detection because it is based on the breadth first search (BFS). It exploits the fact that every cycle has the most distant and the nearest vertex and that when traveling through any (nontrivial) cycle we have to "return back" to a vertex that is closer to the source at least once. Hence, a necessary condition for a path in a graph to form a cycle is that it contains such a *back-level edge*. Back-level edges are computed by BFS which can be (unlike DFS) reasonably parallelized.

In this paper we describe how to turn the graph algorithm into a real explicit state distributed memory LTL model checker by extending it with the detection of accepting cycles, counterexample generation and partial order reduction.

The rest of the paper is organized as follows. In the next section we recall the main ideas of the graph algorithm as given in [2]. The following sections deal with accepting cycle detection, counterexample generation, and partial order reduction, respectively. In a separate section, we summarize the experimental evaluation of the algorithm.

2 Detecting Cycles in Parallel

The algorithm given in [2] is designed to be performed on a network of workstations. Its task is to decide whether there is a cycle in a distributed graph.

A distributed graph is such a graph whose vertices are divided into as many disjoint sets as there are participating workstations. In particular, a partition function is introduced to assign to each vertex a workstation the vertex belongs to (the workstation owns the vertex). The entire distributed computation is started and terminated by one of the workstations involved. This distinguished workstation is called the *manager*.

The graph is supposed to be given implicitly, i.e. by an *initial* vertex and a function that for a given vertex returns its immediate successors. During the computation of the algorithm all the generated vertices of the graph are stored on the corresponding workstations. Thus each workstation keeps its own part of the distributed graph. If an exploration should proceed to a vertex that does not belong to the workstation, a message containing the vertex is sent to the workstation owning it and the local exploration of the vertex is skipped. The vertex is further processed by the destination workstation.

Before explaining the idea of the algorithm we recall the definition of a back-level edge. In short, a back-level edge is such an edge in the graph that does not increase the distance from the initial vertex. For simplicity, we assume that all vertices of the given graph are reachable.

Definition 2.1 Let $\mathcal{G} = (V, E)$ be a graph with the initial vertex s and let u be an arbitrary vertex. The *distance* of the vertex u, denoted by $d(u)$, is the length of the shortest path between s and u. An edge $(u, v) \in E$ is called a *back-level edge* if and only if $d(u) \geq d(v)$. The set of all vertices with the same distance is referred to as a *level*. By $Level_k$ we denote the set of all vertices with the distance k, i.e. $Level_k = \{u \in V \mid d(u) = k\}$.

It is easy to see that for each cycle in the graph there is a maximal k such that the cycle contains a vertex from $Level_k$. Moreover, any edge on the cycle leading from a vertex in $Level_k$ has to be a back-level edge. Since all vertices in the cycle have a successor it is obvious that each cycle in the graph contains at least one back-level edge.

The cycle detection algorithm works as follows. There are two procedures that the algorithm performs alternately. The task of the first procedure (henceforth called *primary*) is to find all the back-level edges by exploring the graph gradually level by level, while the task of the second procedure (henceforth called *nested*) is to test each discovered back-level edge for being a part of a cycle. The primary procedure is implemented as a level synchronized breadth first search of the graph. As soon as a level is completely explored, the nested procedures are initiated for all the back-level edges emanating from a vertex on the current level (called *current back-level edges*) in order to detect cycles. Thus the goal of a nested procedure initiated for a back-level edge is to hit the vertex from which the back-level edge emanates (called *target*). If at least one nested procedure succeeds then the presence of a cycle is ensured and the algorithm is terminated. Otherwise, the primary procedure continues with the exploration of the next level. Each nested procedure searches for its target in a depth first manner. Since there are many nested procedures performed concurrently, the target of each nested procedure has to be propagated by the procedure itself. Unlike the standard DFS, the vertices are not marked as visited and so may be revisited. Note that the search space of nested procedures can be limited to the vertices that have already been visited by the primary procedure.

Obviously, the nested procedures may revisit some vertices many times. To prevent this we suggested several optimizations that decrease the revisiting factor fairly. The first idea is to eliminate repeated visits that are made by the same nested procedure. For this purpose we store at each vertex the identification (target) of the last nested procedure that walked through the vertex. If a nested procedure visits a vertex it has visited previously, it is not allowed to pass through it unless the vertex was visited by another nested procedure in the meantime. To illustrate the optimization let us consider the graph $a)$ in Figure 1. It can be easily seen that the nested procedure initiated for the back-level edge (A, B) will visit the vertex G four times (along all the paths from B to G) without the optimization, but only twice (from E and F) if the optimization is considered.

Another optimization we suggested reduces vertex revisits that are made by different nested procedures. In order to do so we introduced an *ordering* on the nested procedures induced by an ordering of their targets. At each vertex we store the identifier of the highest nested procedure that passed through it. Only a procedure with higher identifier is allowed to proceed through the vertex. This technique reduces revisiting of vertices quite a lot. Unfortunately, it breaks the cycle detection capability because a nested procedure that could reveal a cycle may be stopped before reaching its target. The situation is illustrated on the graph $b)$ in Figure 1. Let us suppose an ordering in which $A > F$. If the nested procedure $[A]$ (the one corresponding to the back-level edge (A, C) and having A as its target) reaches the vertex C before the nested procedure $[F]$ then the nested procedure $[F]$ is stopped at the vertex C. However, the nested procedure $[A]$ continues through the vertices E, F, B to the vertex C where it finishes without revealing the cycle.

To address this problem we change the identification of each nested procedure to a pair $[T, n]$ where T is the target vertex and n is the number of current back-level edges the procedure has passed through. Hence, the identification is dynamically modified as the procedure continues. We also need to redefine

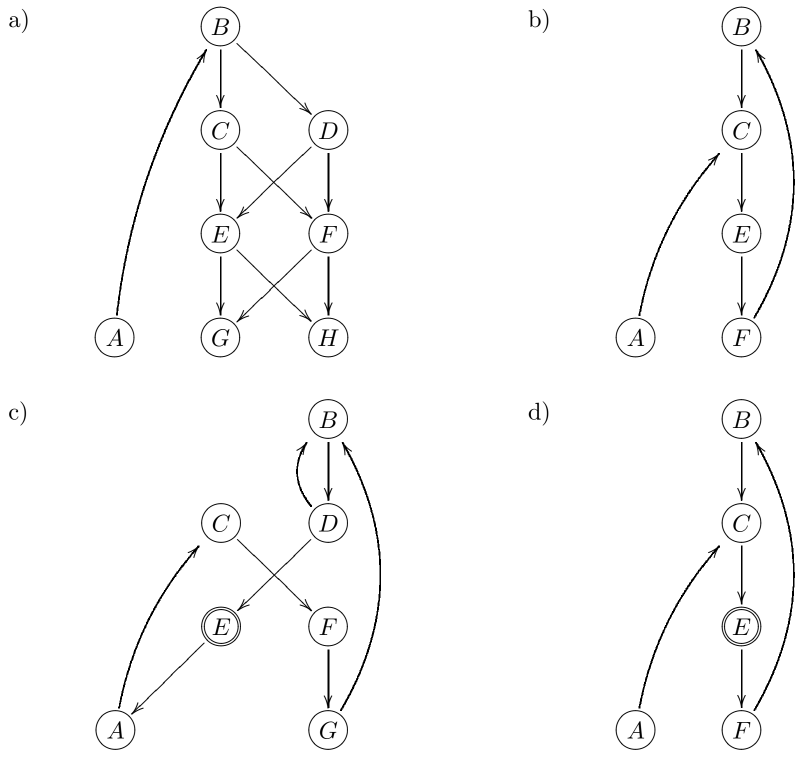

Fig. 1. Revisits reduction and accepting cycle detection

the ordering of the nested procedures. A nested procedure identified as $[A, x]$ is less than a nested procedure identified as $[B, y]$ if $A < B$ or $A = B$ and $x < y$. Now, 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 situation let us consider again the graph $b)$ in Figure 1 and an ordering such that $A > F$. There are two back-level edges on the current level and so there are two nested procedures initiated, namely $[A, 0]$ and $[F, 0]$. When they reach the vertex C they have both passed one current back-level edge. If the nested procedure $[A, 1]$ for the back-level edge (A, C) reaches the vertex C before procedure $[F, 1]$ for the back-level edge (F, B), the latter one is stopped. In such a case the procedure for the back-level edge (A, C) continues in exploration of the graph. When it visits the vertex C for the second time, its identification is $[A, 2]$ which is obviously greater then $[A, 1]$ and so the procedure is not stopped and continues through the vertex C again. It goes through the vertices E and F and increases its number of passed current back-level edges when it reaches the vertex B. At that moment the number of passed current back-level edges by the procedure for the back-level edge (A, C) exceeds the total number of current back-level edges (which is two), the presence of a cycle is detected and the algorithm terminates.

We now focus our attention on the model checking problem and show how to turn the distributed memory cycle detection algorithm into an explicit state distributed memory LTL model checker by extending it with the detection of accepting cycles, counterexample generation and we show how to combine it with partial order reduction.

3 Accepting cycles

The LTL model checking problem can be reduced to the language emptiness problem for Büchi automata [19]. The given system is modeled by a *system automaton* with some states marked as *accepting* and the (negation of the) verified LTL formula is transformed into a *property automaton.* The two automata are combined together to form a synchronous product automaton that is tested for the language emptiness. If the language of the automaton is empty then the model of the system satisfies the verified property. In the other case the accepting run of the product automaton (so called *counterexample*) gives a behavior of the model that breaks the property. A Büchi automaton accepts an empty language if and only if there is no reachable accepting cycle in the corresponding graph. As regards the terminology we switch to a more convenient one and speak of states instead of vertices and transitions instead of edges whenever appropriate.

Unless under special circumstances, we cannot directly use the algorithm from [2] to detect the language emptiness of a Büchi automaton because the algorithm cannot distinguish between accepting and non-accepting cycles. Nevertheless, we can exploit the verified property to decompose the examined graph into several components (each being a set of strongly connected components of the graph) to perform limited accepting cycle detection. In general, the components can be of one of the three types [11]: 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. It is obvious that as for the model checking, the accepting cycle detection need not be made in the non-accepting components of the graph at all and can be replaced by a simple cycle detection in the fully accepting components of the graph. Since the types of components are completely determined by the verified property we are able to precompute the decomposition of the product automaton easily in advance. In [2] we performed the LTL model checking on those graphs that were made of non-accepting or fully accepting components only.

The basic idea behind detection of accepting cycles in partially accepting components is to prevent the algorithm from detecting non-accepting cycles. 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 to count 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 to report a cycle only if it reaches the target state and the bit is set to true. The pseudo-code of procedure SEARCH-FOR-ACCEPTING-CYCLE is given in Figure 2.

We demonstrate the behavior of the procedure on the graphs $b)$ and $d)$ in Figure 1. The nested procedure for back-level edge (A, C) arrives at state C as a procedure $[A, 0]$ because A is not an accepting state and the accepting bit is initially set to false which means that the procedure does not increase its counter of passed back-level edges when it passes the edge (A, C). Similarly, the nested procedure for back-level edge (F, B) arrives at state B as procedure $[F, 0]$.

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

Let us assume now that $A > F$ and procedure $[A, 0]$ arrives at state C before procedure $[F, 0]$ does. In such a case procedure $[F, 0]$ is stopped when it arrives at state C, while procedure $[A, 0]$ continues in the search. In the case of the graph $b)$ procedure $[A, 0]$ passes 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 state C for the second time. In the case of the graph $d)$ the $[A, 0]$ procedure sets its

```
proc SEARCH-FOR-ACCEPTING-CYCLES($nmbrbl$)
  while ($\neg Synchronize() \vee BBLQ \neq \emptyset$) do
        if ($BBLQ \neq \emptyset$)
          then ($q, prelevel, target, bl, abit$) := $dequeue(BBLQ)$;
                if $d(q) < Level$
                  then if ($IsAccepting(q)$)
                          then $abit$ := true
                        fi
                        if ($q = target \wedge abit =$ true)
                          then ACCEPTING-CYCLE-DETECTED()
                        fi
                        if ($prelevel = Level - 1 \wedge abit =$ true)
                          then $bl := bl + 1$
                               $abit$ := false
                               if ($bl > nmbrbl$)
                                 then ACCEPTING-CYCLE-DETECTED()
                               fi
                        fi
                        if (($Level - 1 > q.level$) $\vee$
                              ($Level - 1 = q.level \wedge$
                                ($target > q.target \vee$
                                  ($target = q.target \wedge bl > q.bl$) $\vee$
                                    ($target = q.target \wedge bl = q.bl \wedge$
                                      $abit > q.abit$))))
                          then $q.target := target$
                               $q.bl := bl$
                               $q.level := Level - 1$
                               $q.abit := abit$
                               foreach $t \in Successors(q)$ do
                                 if ($Owner(t) \neq WorkstationId$)
                                   then Send message to $Owner(t)$ :
                                          $enqueue(BBLQ,$
                                                $(t, d(q), target, bl, abit))$)
                                    else $push(BBLQ,$
                                              $(t, d(q), target, bl, abit)$)
                                 fi
                               od
                        fi
                fi
        fi
  od
end
```

Fig. 2. Improved procedure CHECK-BL-EDGES with accepting cycle detection

accepting bit to true when it passes the accepting state E which allows the procedure to increase 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 state C for the second time being identified as $[A, 1]$. This means that the procedure is not stopped but it continues in the search. At state E it sets its accepting bit to true and passes the back level-edge (F, B) changing its identifier to $[A, 2]$. Then it goes through state C for the third time. At the state E it sets the accepting bit to true again 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 current level. See the cycle B, D, B in the graph $c)$ in Figure 1. 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 that there may be accepting cycles that cannot be detected by the first way of cycle detection (i.e. by hitting the target). Let us consider the graph $c)$ from Figure 1 and an ordering in which $G > A$. The nested procedure $[A, 0]$ is stopped at state G because $[A, 0] < [G, 0]$. The nested procedure $[G, 0]$ sets its accepting bit to true at state E and so turns itself into $[G, 1]$ when passing the back-level edge (A, C). Hence, whenever it arrives at its target its accepting bit is set to false. Therefore, in this case the cycle will be detected by procedure $[G, 3]$ at state C by exceeding the number of current back-level edges.

Theorem 3.1 *The accepting cycle detection algorithm always terminates and reports the presence of an accepting cycle if and only if there is an accepting cycle in the product automaton graph.*

The proof exploits the correctness of the algorithm for distributed back-level edge detection as presented in [2]. In addition, several facts have to be taken into account. If there is an accepting cycle in the graph then at least one (shallowest) cycle is explored completely by a nested procedure because either all queues $BBLQ$ are emptied before the next level of the graph is processed or the presence of an accepting cycle is reported. Another important fact is that no nested procedure can pass through a non-accepting cycle infinitely many times. For a more detailed proof see the full version of the paper [3].

4 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 section we present a technique to generate short counterexamples.

A counterexample consists of two parts: an *accepting cycle* and a *path* that reaches it from the initial state s. In the sequential *Nested DFS* algorithm, the counterexample is simply generated by following the DFS 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 search stacks, hence a different approach has to be considered.

Recall that our algorithm uses, similarly to the Nested DFS, two search procedures. Unlike the Nested DFS, the primary search is a breadth first search. Both search procedures build *parent graphs* that are utilized for counterexample generation.

More precisely, for each vertex v the value $par(v)$ is stored during the primary search which is the parent of the vertex v in the search (called BFS parent). Note that it is assigned only once during the whole computation. In addition, we also store the value $par\text{-}dfs(v)$ which is the parent of the vertex v in a nested search (called DFS parent). It is assigned every time a nested procedure is allowed to pass through the vertex. In both cases, the parents induce edges in the following way. If v is a vertex and $par(v)$ is the BFS parent of v, then $(v, par(v))$ is the induced edge in the *BFS parent graph*, similarly the DFS parents define the *DFS parent graph*. In the following we show how the counterexample can be found by traversing the BFS and DFS parent graphs.

The accepting cycle part of a counterexample is generated using the DFS parents. 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 vertex can be visited several times each time changing its DFS parent. As a result, the DFS parent graph may not contain a cycle even though the input graph contains a cycle. A possible situation is exemplified in Figure 3. Suppose the *first* nested procedure starts from the vertex F and the DFS parent graph is build as the procedure continues (see Figure 3.a). Suppose that after the procedure passes through the vertex C the *second* nested search

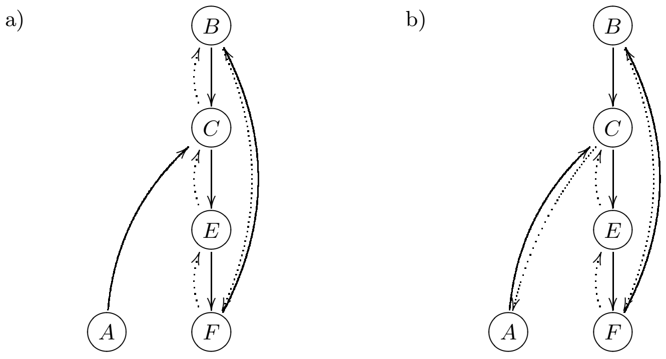

Fig. 3. Interrupting a cycle in the DFS parent graph

which has been started from the vertex A visits the vertex C before the first procedure actually closes the cycle. Further suppose that $A > F$. When exploring the vertex C by the second procedure, the DFS parent for C is reset to point to the vertex A, hence the possible cycle in the DFS parent graph is interrupted (see Figure 3.b).

To solve this problem we assign to the nested search that detected the cycle a new (highest) identification and we re-execute it independently. In our example in Figure 3 we start the nested search from the vertex F once more, the second search from A is not performed.

During the generation of the counterexample the manager workstation is used as a "collector" to which all the workstations participating in the generation send information. Note that the vertices forming a counterexample are spread across the network according to the partition function. The counterexample is generated in two steps. In the first one the DFS parent graph is traversed starting at the state where the cycle was detected. The visited vertices are marked in order to discover the cycle. Once an already marked vertex 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 vertex v with the smallest distance on the cycle. In the second step of the generation the BFS parents are traversed from v back to the initial vertex of the 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 our 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 section on experiments for a few examples.

5 Partial Order Reduction

In this section we describe how to combine partial order reduction with our distributed memory algorithm. We start by a brief review of the partial order reduction method following mainly the presentation of [10] before explaining the proposed distributed approach.

To describe partial order reduction method it is not sufficient to use graphs as the semantical models. The concurrent systems that we analyze are modeled in a more appropriate way as state transition systems (labeled transition systems). If S is the set of *states*, a *transition* is a relation $\alpha \subseteq S \times S$. A *state transition system* is then defined as a tuple $M = (S, s_0, T, L)$, where $s_0 \in S$ is an *initial state*, T is a set of transitions $\alpha \subseteq S \times S$, and $L : S \rightarrow 2^{AP}$ is a labeling function that assigns to each state a subset of some set AP of *atomic propositions*.

A transition $\alpha \in T$ is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$. The set of all transitions enabled in a state s is denoted *enabled(s)*. We presuppose that transitions are deterministic, i.e., for every α and s there is at most one s' with $\alpha(s, s')$, and denote it as $\alpha(s) = s'$. If $\alpha(s, s')$ we say that s' is a *successor* of s.

The partial order method exploits the fact that concurrent transitions can be interleaved in either order. This can be formalized by defining an independence relation on pairs of transitions.

An *independence* relation $I \subseteq T \times T$ is a symmetric, anti-reflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:

(i) Enabledness – If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha \in \mathit{enabled}(\beta(s))$.

(ii) Commutativity – If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The *dependency* relation is the complement of I. Heuristic methods are utilized for an efficient computation of a dependence relation according to the conditions mentioned above.

The independence relation suggests a potential reduction to the state transition system by selecting only one from the independent transitions originating from a state s. However, this cannot guarantee that the reduced state transition system is a correct replacement of the full one as it does not take into account the property to be checked. Also, eliminating one of the intermediate states $\alpha(s)$ or $\beta(s)$ may cause some of its successors (which are significant for verification) not to be explored. Additional conditions for the correctness of the reduction are needed. They are described in the following.

First, we make it precise what it means that a property is taken into account by defining the concept of *visibility* of a transition.

A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $\alpha(s, s')$, $L(s) \cap AP' = L(s') \cap AP'$ holds. A transition is *visible* if it is not invisible. The set AP' is usually induced by the set of atomic propositions included in the verified formula.

The reduced state transition system is generated by a modified generation algorithm which explores only a subset of transitions, enabled at each state encountered during the generation, called an *ample set*. The ample set can be defined in a manner that does not depend on the particular way the state transition system is generated. This is accomplished by a set of *conditions* relating the full state transition system to the corresponding reduced one. Note that there could be more than one ample set satisfying the conditions for a given state. We say that a state s is *fully expanded* whenever $ample(s) = enabled(s)$.

Let AP' be a set of atomic propositions. *Ample conditions* with respect to the set AP' are:

C0 $ample(s) = \emptyset$ if $enabled(s) = \emptyset$.

C1 Along every path in the full state graph M that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

C2 If $enabled(s) \neq ample(s)$, then every $\alpha \in ample(s)$ is invisible w.r.t. to AP'.

C3 (*cycle closing condition*) A cycle in the reduced state graph M_R is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

The conditions characterize the ample sets needed to generate reduced state transition systems sufficient for checking safety and liveness properties.

Our aim is to combine partial order reduction with our distributed memory algorithm. The reduced system is computed by a generation algorithm which systematically explores states in such a way that for every state s it chooses a set $ample(s) \subseteq enabled(s)$ and follows the transitions from $ample(s)$ only. The key part of such an algorithm is without any doubts the distributed memory checking of the ample conditions.

While checking the conditions **C0** and **C2** is easy and can be done locally, checking conditions **C1** and **C3** is as hard as solving the reachability problem. Nevertheless, the condition **C1** can be checked locally using the same approximating heuristics as in the sequential case (see [10]). Therefore, the cycle closing condition **C3** is the only one which is difficult to be checked in a distributed environment. In the sequential case when exploring the state graph by *depth first search*, the condition **C3** is checked in *constant time* using the *search stack*. In fact, the following stronger condition is used instead of **C3**.

C3-dfs *If a state s is not fully expanded, then no transition in ample*(s) *leads to a state on the search stack.*

Our aim is to develop a counterpart of the condition **C3-dfs** for the *breadth first search* based generation of the state transition system which is distributed among several workstations. A new proviso is motivated by the fact that a necessary condition for closing a cycle in breadth first search is that the state closing the cycle has already been visited during the search. Hence, we use the following cycle closing condition which can be easily checked.

C3-bfs *If a state s is not fully expanded, then ample*(s) *does not contain a back-level edge, i.e. a transition that leads to a state that is on the current or previous level of the breadth first search.*

To utilize partial order reduction within our distributed memory *on-the-fly* algorithm we use the similar approach as implemented e.g. in the model checker SPIN [13]. The approach combines the construction of the state space with checking that it satisfies the specification by exploring the product graph. The only condition that needs attention is obviously the cycle condition **C3-bfs**. It can be shown that it is correct to check the condition with respect to cycles of the product. We have implemented the method and experimentally confirmed reductions in space and time.

6 Experimental evaluation

We implemented the algorithm from scratch, thus the implementation misses many optimizations commonly used in standard sequential tools (e.g. SPIN). In particular, our implementation of the partial order reduction is quite far from being optimal. For comparison reasons we implemented the Nested DFS algorithm without any optimizations as well. All the experiments were conducted on a homogeneous network of twelve workstations interconnected by 100Mbps switched Ethernet. Each workstation was equipped with an Intel Pentium 4 2.6 GHz CPU and 1GB RAM.

We measured the performance of the algorithm on various model checking problems, each given by a model and a formula. The models include Elevator model (elev), Leader election protocol (lead), Peterson solution for mutual exclusion (pet), Dinning philosophers problem (phil), and RealTime Ethernet protocol (rte). All the models were parametrized: Elevator by the number of served floors, Leader election, Peterson, and Philosophers by the number of

Problem	Model (M)	Verified property (φ)	$M \overset{?}{\models} \varphi$
elev1	Elevator	G (r1 $\implies$ ($\neg$p1 U (p1 U (p1$\wedge$o))))	no
elev2	Elevator	G (r1 $\implies$ ($\neg$p1 U (p1 U ($\neg$p1 U (p1 U (p1$\wedge$o))))))	yes
lead1	Leader election	FG (one_leader)	yes
lead2	Leader election	F (more_leaders)	no
pet1	Peterson	G (p0_in_cs $\implies$ F ($\neg$p0_in_cs))	yes
pet2	Peterson	G ($\neg$p0_in_cs $\implies$ F (p0_in_cs))	no
pet3	Peterson-error	GF (someone_in_cs)	no
phil1	Philosophers	GF (phil0_eats)	no
phil2	Philosophers	GF (someone_eats)	yes
rte1	RT Ethernet	G (r0 $\implies$ ($\neg$ce U (ce U ($\neg$ce $\wedge$ (rt0 R $\neg$ce)))))	yes
rte2	RT Ethernet	G (w0 $\implies$ ($\neg$ce U (ce U ($\neg$ce $\wedge$ (rt0 R $\neg$ce)))))	no

Fig. 4. Models and verified properties

participating processes, and RT Ethernet by the number of network nodes. For each model we verified a few LTL formulas trying to cover both cases, i.e. the case when the model satisfies the formula and the case it does not. See the table in Figure 4 for the list of model checking problems we used.

The table in Figure 5 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 our 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 same table.

Other experiments were performed to evaluate the partial order reduction based on the condition **C3-bfs**. See the table in Figure 6. 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 [10] (the column "POR-DFS"). This value was measured using the standard sequential DFS algorithm ("—" means the algorithm exceeds the available memory). Note that a reduction was achieved only

Problem	Counterexample Length	Counterexample Length (NDFS)	Time	Time (NDFS)
elev1(9)	85	323	136	1
elev1(10)	65	356	527	1
lead2(4)	74	74	3	1
lead2(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	635
rte2(9)	207	17478	54	4
rte2(10)	244	60520	76	18

Fig. 5. Counterexample length and generation time

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

Fig. 6. Partial Order Reduction

on the Leader election and the Peterson models. 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 slightly worse in the case of Peterson model.

Finally, we measured the impact of the suggested partial order reduction on the counterexample generation. The table in Figure 7 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 and the time needed to discover and generate the counterexample (see the row pet2(4)). Nevertheless, the impact of the partial order reduction on the lengths of counterexamples is generally minimal.

Problem	CE Length	CE Length (POR)	Time	Time (POR)	Level	Level (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

Fig. 7. Impact of partial order reduction on counterexample generation

7 Conclusions and Related Work

We proposed an extension of the algorithm presented in [2] to a full LTL model checking algorithm without loosing any of its positive features. In particular, we show how to detect accepting cycles, generate counterexamples, and how to employ partial order reduction.

As expected the counterexamples were very short compared to those returned by the Nested DFS algorithm. This is due to the breadth first like nature of the state space generation. However, the time to find the counterexample was generally longer and more states had to be explored. We stress that the shortness of a counterexample is crucial in debugging.

Another positive feature of the algorithm is its behavior on graphs with a small number of back-level edges. In such cases the behavior of our algorithm for the full LTL model checking practically equals to the reachability analysis. There were several models confirming this.

On the other hand, there are several drawbacks associated with the breadth first approach. We have mentioned it may generally take more time to find a counterexample. Furthermore, the Nested DFS algorithm is sometimes able to find an error in extremely large graphs, while our algorithm can succeed only if the error is not "far" from the initial vertex in these cases. Another drawback arises 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.

As firstly argued in [14] the partial order reduction technique exploiting the **C3-dfs** condition is hardly transformed to the distributed memory state space generation. On the other hand, the reduction achieved by our algorithm combined with the **C3-bfs** condition is in many cases practically comparable to the reduction in the standard sequential Nested DFS algorithm. Of course, in some cases the **C3-bfs** is too weak to give a significant reduction. In [17] the distributed algorithm *Twophase* is proposed. In contrast to the sequential Nested DFS algorithm algorithm, *Twophase* is much simpler as it works only with singleton ample sets, i.e., whenever there is no singleton satisfying the ample conditions, the state is fully expanded. We have not compared our algorithm directly to the *Twophase* algorithm. Recently, yet another approach to partial order reduction in distributed environment has been proposed in [8]. The technique is used to generate reduced state space, however it is not possible to combine it directly with on-the-fly breath first search algorithms.

The distributed counterexample generation (if considered) is typically performed as two reachability procedures: the first one localizes a reachable vertex on an accepting cycle, while the second one generates the cycle. BFS parent graph have been used for the first search in [9,7].

References

[1] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In M. Leuschel and U. Ultes-Nitsche, editors, *Proceeding of the 3rd International Workshop on Verification and Computational Logic (VCL'2002)*, pages 1–10, October 2002.

[2] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[3] J. Barnat, L. Brim, and J. Chaloupka. Distributed Memory LTL Model-Checking Based on Breadth First Search. Technical Report FIMU-RS-2004-07, Faculty of Informatics, Masaryk University Brno, 2004.

[4] 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.

[5] G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Proc. Workshop on Parallel and Distributed Model Checking*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[6] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 543–558. Springer-Verlag, 2001.

[7] 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.

[8] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed partial order reduction of state spaces. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC'04). To appear.*

[9] 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.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[11] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In Matthew B. Dwyer, editor, *8th International SPIN Workshop*, number 2057 in LNCS, pages 57–79. Springer, 2001.

[12] H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, volume 2057 of *LNCS*, pages 216–234, Toronto, Canada, 2001. Springer-Verlag.

[13] 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.

[14] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*, pages 22–39. Springer-Verlag, 1999.

[15] A. Lluch-Lafuente. Simplified distributed model checking by localizing cycles. Technical Report 176, Institute of Computer Science at Freiburg University, 2002.

[16] Z. Manna and A. Pnueli. The Modal Logic of Program. In *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming (ICALP 1979)*, volume 71 of *LNCS*, pages 385–409. Springer-Verlag, 1979.

[17] R. Palmer and G. Gopalakrishnan. Partial order reduction assisted parallel model-checking (full version). Technical report, University of Utah, August, 2002.

[18] U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.

[19] 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.