Parallel Breadth-First Search LTL Model-Checking*

Jiří Barnat, Luboš Brim, and Jakub Chaloupka
Faculty of Informatics, Masaryk University Brno,
Botanická 68a, Brno, Czech Republic
{barnat,brim,xchalou1}@fi.muni.cz

Abstract

We propose a practical parallel on-the-fly algorithm for enumerative LTL model-checking. The algorithm is designed for a cluster 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 back-level edges. For each level the back-level edges are checked in parallel by a nested depth-first search 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 implementation of the algorithm shows promising results.

1. Introduction

With the increase in the 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.

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 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 efficient sequential solution to the LTL model-checking problem builds on searching for cycles in the state space graph by the depth-first search (DFS) technique. In particular, the *postorder* as computed by DFS is crucial for cycle detection. Our aim is to solve the LTL model-checking problem in distributed environment, i.e. by utilizing several interconnected workstations. However, when exploring the state space in parallel, the DFS postorder is not generally maintained due to different speeds of involved workstations. To that end we propose a parallel algorithm that does not employ DFS postorder for cycle detection.

A subproblem of the LTL model-checking problem, when safety properties are considered only (equals to *reachability analysis*), has been effectively distributed in [11]. The success of distributed reachability analysis comes from the fact it can be done by searching the state space using breadth-first search (BFS) which can be (unlike DFS) reasonably parallelized. On the other hand, BFS as such is not effective in searching for cycles.

The distinguished feature of BFS is that, besides "discovering" the reachable state space, it computes the distance (smallest number of edges) from the source vertex to each reachable vertex. The novelty of our parallel algorithm is in the intriguingly simple idea of exploiting the fact that every cycle has the most distant and the nearest vertex and that when traveling through any (nontrivial) cycle we have to

*This work has been partially supported by the Grant Agency of Czech Republic grant No.201/03/0509.

"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*. The idea of the algorithm is to detect all back-level edges by distributed BFS and then check in parallel whether at least one back-level edge belongs to a cycle by DFS. The use of DFS in the cycle-check is possible because we perform reachability only.

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 elaborates various techniques to check for cycles. Section 5 summarizes case-studies and experiments we have conducted and finally, in Section 6 we relate our approach to the other work on distributed LTL model-checking, give some conclusions and outline future work.

2. LTL Model-Checking

Model-checking is a fully automated formal technique for the verification of concurrent software. It is based on the 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.

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 autmaton 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 others successful technique 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 behavior. 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 exapnded.

Since LTL model-checking can be reduced to the detection of (accepting) cycles in the graph, we keep the graph notation. In particular, we speak of vertices instead of states, edges instead of transitions etc.

3. Basic Parallel Algorithm

We suppose the parallel algorithm is to be performed on a cluster of workstations communicating via message passing interface (MPI), hence no global information is directly accessible (no shared memory). Each workstation executes the same algorithm. In addition, we consider a distinguished workstation (called the *manager*) which is responsible for the initialization of the entire computation, termination detection, and *counter-example* (invalid execution) generation. The vertices of the input graph are divided into disjoint parts by a *partition function* and the parts are assigned to workstations. Each workstation is responsible for the graph induced by the owned subset of vertices. For each vertex v the partition function gives the identification *Owner(v)* of the workstation which owns the vertex v.

It is also important to note that in the on-the-fly LTL model-checking the graph to be processed is not completely given at the beginning of the computation through its adjacency-list or adjacency-matrix representation. Instead, we are given a source vertex together with a function (*Successor*) which for every vertex computes its adjacency-list. As successors of a vertex are determined dynamically there is no need to store any information about edges permanently. Moreover, the technique allows to generate only reachable part of the graph and thus reduces the space requirements for the graph representation.

Breadth-first search (BFS) is a simple algorithm for searching a graph. Given a graph $G = (V, E)$ and a *source* vertex $s \in V$, BFS systematically explores the edges of G to discover every vertex reachable from s. BFS expands the *frontier* between discovered and undiscovered vertices uniformly across the breadth of the frontier. The frontier vertices can be expanded in any order, therefore discovering of vertices can be computed asynchronously in parallel on several workstations.

Another task BFS does is that it computes the *distances* (smallest number of edges) from the source vertex s to all other reachable vertices. It also produces a *breadth-first tree* with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is the path containing the smallest number of edges. In contrary to discovering vertices, computing distances (and the BFS tree) in parallel requires some synchronization.

Before presenting the parallel algorithm for detection of cycles we give formal definitions and introduce some notation. We also explain how the BFS procedure can be extended to detect cycles. For simplicity, we assume that all vertices of the given graph are reachable. By $d(u)$ we denote the distance of the vertex u from the source vertex s. The set of vertices with the same distance is called *level*; by *level*(k) we denote the set $\{u \in V \mid d(u) = k\}$).

Definition 1 *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.

Lemma 1 *Any cycle of length at least one in the graph $\mathcal{G}$ contains a back-level edge.*

Proof: Let $\pi = v_0, \ldots, v_n, n \geq 0$ 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 2 *For each cycle $\pi = v_0, \ldots, v_n, n \geq 0$ 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* maxdepth(π).

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

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

We are now ready to describe the sequential BFS-CYCLE-DETECTION algorithm which detects cycles using BFS. The algorithm performs BFS and once a back-level edge from vertex v is detected the procedure SEARCH-FOR-CYCLE is called to check the existence of a cycle from v back to v. The SEARCH-FOR-CYCLE procedure can be implemented as a DFS or BFS procedure. We decided to use DFS for its "greedy" nature of search, particularly successful in LTL model-checking, and also for its capability of generating simulation traces, hence counter-examples.

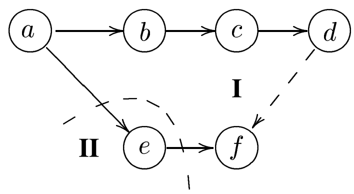

Figure 2. Undetected Back-Level Edge

If we want to perform the BFS-CYCLE-DETECTION algorithm in a distributed manner, several additional facts have to be taken into account. The main problem is that the "BFS frontier" can get splitted into different local parts on workstations. As a result back-level edges might remain undetected unless we conduct an expensive re-computation. This is exemplified in Figure 2. The vertex f can be visited by the BFS-CYCLE-DETECTION algorithm through the path a, b, c, d, f (performed entirely on the workstation **I**) *before* it is encountered through the path a, e, f part

```
proc PARALLEL-BFS-CYCLE-DETECTION(u, WorkstationId)
  Visited = ∅;  CLQ = ∅;  NLQ = ∅;
  Level = 0;
  if (WorkstationId = Manager) then enqueue(NLQ, u) fi
  while (¬finished) do
        swap(NLQ, CLQ); nmbrbl := 0;
        while (CLQ ≠ ∅) do
              v := dequeue(CLQ);
              if (v ∉ Visited)
                 then
                        add v to Visited; d(v) := Level;
                        foreach t ∈ Successors(v) do
                           if (Owner(t) ≠ WorkstationId)
                              then
                                     Send message to Owner(t) : "enqueue (NLQ,t)"
                               else enqueue(NLQ, t) fi
                        od
                 else
                        if (d(v) < Level) then enqueue(BLQ, v); nmbrbl := nmbrbl + 1 fi
              fi
        od;
        SYNCHRONIZE();
        while BLQ ≠ ∅ do enqueue(BBLQ, (dequeue(BLQ), Level − 1, id, 0)) od
        SEARCH-FOR-CYCLES(global_nmbrbl);
        Level = Level + 1
  od
end
```

Figure 1. Parallel BFS Cycle Detection Algorithm

of which is computed on the workstation **II**. Hence, the edge (d, f) will not be marked as a back-level edge. This is caused by different speeds of workstations and network communication.

Several solutions are at hand. We know that all the distances will eventually receive their correct values. Therefore, we can start to search for back-level edges (and cycles) in the graph after the graph has been completely explored. Alternatively, we can start such a search always when a vertex's distance has been improved. For our parallel algorithm we chose yet another approach which requires some synchronization but is more time efficient. All the independent BFS-CYCLE-DETECTION algorithms performed on workstations synchronize as soon as they have discovered all vertices at the same distance k (*level* k). Note that this may require to send successors of *level* $k-1$ to their owners. To this end we consider two queues: *CurrentLevelQueue – CLQ* which consists of vertices at the *current level* and *NextLevelQueue – NLQ* where successors of vertices on the current level are placed. After synchronization the *NextLevelQueue* becomes *CurrentLevelQueue* and *NextLevelQueue* is emptied. This approach ensures that incorrect distances are never assigned to vertices. The pseudo-code of the PARALLEL-BFS-CYCLE-DETECTION algorithm is given in Figure 1.

To complete the parallel algorithm we have to describe the SEARCH-FOR-CYCLES procedure. A sufficient method to decide on the presence of a cycle in a distributed environment is to check the reachability of a vertex from itself. In our implementation we use DFS-like reachability procedure. This test must be done for at least one vertex in a cycle. We perform this test on the destination vertex v for each back-level edge (u, v). During the pre-synchronization step all back-level edges from the current level are stored in the *BLQ* list (actually the vertices from which the nested search is to be started are stored only). Once the workstations synchronize the SEARCH-FOR-CYCLES procedure is initiated for each *current* back-level edge. All search procedures can be run in parallel. The SEARCH-FOR-CYCLES procedure does not explore the states bellow the current level. Also, it is guaranteed that there are no cycles "strictly above" the current level (i.e. each cycle contains a back-level edge from the current level). By Lemma 1 and Lemma 2 it is guaranteed that a cycle is detected whenever the search is executed for the most distant back-level edge of the cycle. However, searching for cycles in parallel requires that each procedure carries its goal (destination vertex). Due to asynchrony of multiple searches some vertices may be re-visited by different procedures. In the next section we develop a particular technique to reduce repeated visits of vertices.

The correctness of the parallel algorithm follows directly the same arguments as in the case of sequential BFS (see e.g. [6]). We conclude by the complexity analysis. Each procedure starts with the BFS exploration. The BFS algorithm runs in time linear in the size of the input, i.e. $\mathcal{O}(m + n)$, where m is the number of edges and n is the number of vertices. Each SEARCH-FOR-CYCLES requires linear time $\mathcal{O}(m + n)$ as well. In the worst case there can be $\mathcal{O}(m)$ back-level edges, hence the overall complexity of each PARALLEL-BFS-CYCLE-DETECTION procedure is $\mathcal{O}(m.(m + n))$. The relatively high theoretical complexity is what we have to pay for easy distribution of the problem.

Theorem 1 PARALLEL-BFS-CYCLE-DETECTION *algorithm terminates and returns true if and only if G contains a reachable cycle. Its time complexity is $\mathcal{O}(m.(m+n))$. Its space complexity is $\mathcal{O}(m + n)$.*

4. Speeding up the Algorithm

In this section we describe two modifications of the parallel algorithm that lead to significant improvements in time complexity in many practical situations. The first one works on general graphs, while the second one is tailored to the graphs that are encountered in LTL model-checking.

It is the SEARCH-FOR-CYCLES procedure that mostly adds to the overall complexity of the algorithm. There are two ways how to improve the algorithm. The first one is to avoid re-visiting of vertices already explored by another search procedure. The second one is to reduce the number of back-level edges that have to be checked for cycles, reducing thus the number of calls to the SEARCH-FOR-CYCLES procedure.

4.1. Reducing the Number of Re-Visits

Cycles are searched level by level, hence we assume that there are no cycles above the current level. A direct approach to prevent exploration of states that have already been visited by another search starting from the current level is simply to stop the search once an already visited state should be re-visited. However, this could lead to incorrect behavior of the algorithm as a cycle can get undetected. The reason is that in the distributed environment two searches running in parallel on a cycle can stop each other without letting at least one of them to complete the search throughout the entire cycle as is exemplified in Figure 3. Suppose that one search procedure explores the cycle from the vertex a and at the same time another procedure searches through the cycle from b. Then both vertices a and b have been visited when the other procedure arrives to them, hence the cycle is not detected.

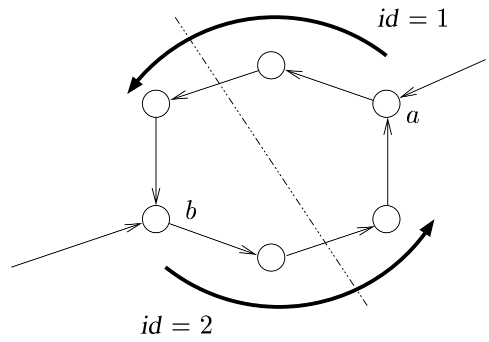

Figure 3. Two Searches Stop Each Other

The main idea behind our solution to the first improvement of the parallel algorithm is the following. We order the current back-level edges by numbering the vertices on the current level in such a way that the current level numbers are strictly greater than the numbers assigned to the vertices on previous levels. The SEARCH-FOR-CYCLES procedures are initiated from the destination vertices of back-level edges and each procedure is given the source vertex number of the back-level edge. Note, that it can happen that two or more procedures receive the same identification number (*id*).

Each time a procedure with a smaller number visits a vertex which has already been visited by a procedure with a greater number, the search is stopped. In the case the procedure visits a vertex that has already been visited by another procedure with the same *id* it is allowed to proceed only if it has passed through more current back-level edges than the last visiting procedure did. The variable *bl* holds the number of current back-level edges the procedure has passed through.

The example in Figure 3 demonstrates a possible situation. The procedure with *id=1* will be interrupted if the vertex b has already been visited by the procedure with *id=2*. However, the procedure with *id=2* will re-visit the vertices that have been visited by the procedure with *id=1* (and discover the cycle).

The search is stopped and a cycle reported immediately after the search visits the vertex from which it has been initiated (the *goal*). However, a search can be stopped before reaching its goal by another search. Such a search has a different goal, hence it may not recognize the cycle and the algorithm can terminate without discovering it.

To ensure the cycle detection we use the following

```
proc SEACRH-FOR-CYCLES(nmbrbl)
  while (BBLQ ≠ ∅) do
      (q, prelevel, id, bl) := dequeue(BBLQ);
      if d(q) < Level
        then
            if (q = id) then CYCLE-DETECTED() fi;
            if (prelevel = Level − 1)
              then bl := bl + 1;
                  if (bl > nmbrbl) then CYCLE-DETECTED() fi fi;
            if (id > id(q) ∨ (id = id(q) ∧ bl > bl(q))))
              then
                  id(q) = id;  bl(q) = bl;
                  foreach p ∈ Successors(q) do
                    if (Owner(p) ≠ WorkstationId)
                      then
                          Send message to Owner(t) : "enqueue (BBLQ,(p, d(q),id,bl))"
                      else
                          push(BBLQ, (p, d(q), id, bl)) fi
                  od
            fi
      fi
  od
end
```

Figure 4. Searching for Cycles in Parallel

method. If a cycle appears at the current level, at least one of the current back-level edges belongs to this cycle. A search procedure counts current back-level edges (each time the search passes through such an edge the value of *bl* is increased). Suppose the global number of back-level edges at the current level is *nmbrbl*. This number is computed from the local values during the synchronization. Once the value of *bl* in a search exceeds the value of *nmbrbl* at least one back-level edge must have been passed twice, hence a cycle must have been discovered and the computation can be terminated. The pseudo-code of the algorithm is given in Figure 4.

In the code the variable *prelevel* keeps the value of the depth of the predecessor of currently explored vertex. The meaning of the variables *id, bl, nmbrbl* has been explained above.

4.2. Reducing the Number of Back-Level Edges

We want to reduce the number of back-level edges that have to be checked for cycles. Some properties of graphs that arise from the LTL model-checking allow several very efficient heuristics for reducing the number of back-level edges that have to be explored during the second search.

One of the particular features of graphs we encounter in LTL model-checking is that not all cycles have to be considered. We are interested only in cycles containing at least one accepting vertex (*accepting cycles*). In this subsection we show, how information on presence or absence of accepting vertices in cycles can be used to reduce the number of SEARCH-FOR-CYCLES procedure calls.

We adopt the technique we have developed in [2] that utilizes the verified property to partition the state space in the distributed on-the-fly automata-based LTL model-checking in order to reduce the number of divided cycles.

In automata-based LTL model-checking the verification problem is represented as the emptiness problem of a Büchi automaton which is obtained as a *synchronous product* of two automata. Thus each vertex has two parts: the one given by the modeled system and the other one given by the negative claim automaton (representing negation of the verified formula). There are three *types* of strongly connected components in the negative claim automaton [8]: Type *F* – any cycle within the component contains at least one accepting vertex, type *P* – there is at least one accepting cycle and one non-accepting cycle within the component, and type *N* – there is no accepting cycle within the component. We do not have to check for cycles within components of type *N* as the only relevance of vertices in the component is in their reachability.

To process components of type *P* we need to distinguish accepting and non-accepting cycles. The method presented

Modeled problem (M)	Verified property (ψ)	$M \models \psi$?
Philosophers	GF (someone_eats)	yes
Philosophers	G (fork[0]=taken → F (fork[0]=released))	no
Elevator	G (floor[1]=requested → F (floor[1]=served))	yes
Elevator	G (floor[1]=requested → F (floor[3]=served))	no
Peterson	GF (someone_in_CS)	yes
Peterson (with error)	GF (someone_in_CS)	no
Leader election	No_leader U One_leader	yes
Leader election	F (Nmbr_of_leaders ≥2)	no
Token ring	GF (proc[1]_in_CS)	no

Table 1. Considered Model-Checking Problems

in Subsection 4.1 can be adapted to detection of accepting cycles. Since type *P* components are quite rare in real applications, the only serious challenge is to find an effective specialized distributed algorithm for type *F* components. For components of type *F* the cycles can be detected directly as described in the algorithm in Figure 4.

The modification of the algorithm according to the described optimization is not given in more details, because only small parts of the PARALLEL-BFS-CYCLE-DETECTION algorithm have to be changed. However, it has been implemented.

5. Experimental Results

A prototype of the algorithm has been implemented. In this section we give a first evaluation of the practicality of our algorithm by applying this prototype to several examples.

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 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. The SPIN tool was used only for translating LTL formulas into the corresponding Büchi automata.

All the experiments were performed on a heterogeneous cluster of Linux workstations equipped with various CPU (Intel Celeron 366MHz, AMD Duron 750, Intel Pentium III 450MHz,500MHz) and 384 MB of RAM. The workstations in the cluster were interconnected with a 100Mbps Ethernet. All the experiments performed on a single workstation (for comparison reasons) were undertaken on an Intel Pentium III 500MHz workstation.

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 made “ad hoc” without being optimized 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.

As the experimental measure for space complexity we used the number of stored vertices (multiplied by the size of the vertex descriptor this gives a rough estimate of random-access memory consumption). Concerning the time complexity we measured how many times the function for enumerating immediate successors of a vertex was called, which we believe is in perfect accord with the virtual time complexity of the algorithm. Of course, the amount of real time the distributed computation took was measured as well. As additional information related to our method, we also considered the maximum level reached during the

Leader election — No_leader U One_leader									
parameter	1	2	3	4	5	6	7	8	9
4	2	–	–	–	–	–	–	–	–
5	7	7	7	7	8	9	11	13	14
6	81	85	79	65	52	66	52	55	54
7	–	–	–	–	1001	888	875	812	834

Peterson (with error) — GF (someone_in_CS)									
parameter	1	2	3	4	5	6	7	8	9
4	18	15	12	12	12	14	14	15	17
5	–	319	257	226	212	201	218	203	240

Elevator — G (floor[1]=requested $\rightarrow$ F (floor[1]=served))									
parameter	1	2	3	4	5	6	7	8	9
6	–	–	–	–	–	–	–	–	–
7	45	39	31	27	26	27	25	30	33
8	179	143	107	81	81	72	81	68	76
9	–	533	407	342	378	286	265	253	331

Table 2. Detailed Runtime Results for Selected Problems

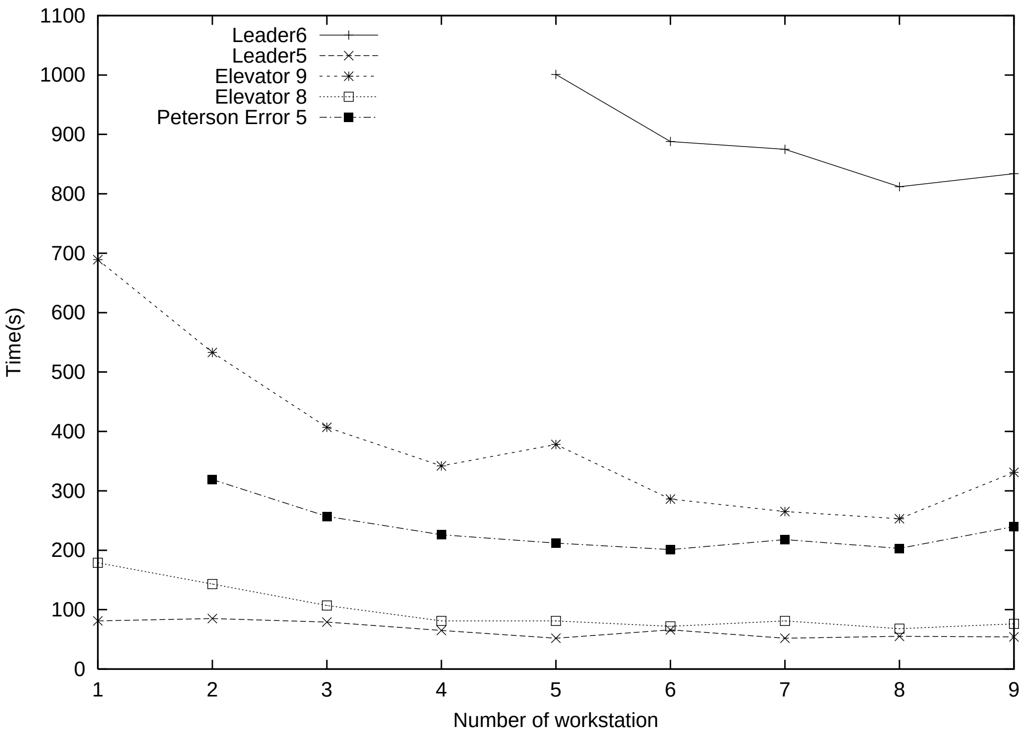

Figure 5. Summary of Runtime Results

	$M \models \varphi$		$M \not\models \varphi$	
Model	maximal level	bl-edges	maximal level	bl-edges
Elevator(8)	75	11782	23	15
Elevator(10)	85	70662	23	23
Elevator(13)	–	–	23	25
Elevator(14)	–	–	23	27
Peterson-error(3)	–	–	30	18
Peterson(3)	152	10168	–	–
Peterson-error(4)	–	–	34	48
Peterson-error(5)	–	–	38	100
Leader(5)	124	0	142	5
Leader(6)	152	0	170	6
Leader(7)	180	0	198	7
Philosophers(9)	39	105760	11	100

Table 3. Number of Back-Level Edges

computation and the number of discovered back-level edges (see Table 3).

For our set of experiments we chose several standard software protocols (see Table 1 for a list). 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 *Token ring* 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.

As stressed in the introduction, the main reason for the development of a distributed model-checking algorithms is typically to increase the available memory trading it for time. Of course, a good distributed algorithm should scale well, meaning that the total runtime should stay constant (additional speedup being a great advantage). We show the scalability results of our algorithm for a selected subset of experiments in Figure 5 and Table 2. Each column in Table 2 gives the size of the cluster (number of workstations involved), and the values in the appropriate row give the runtime (in seconds) of the distributed computation. We stress that in our heterogeneous environment the performance of individual workstations was different, therefore the scalability results are slightly distorted. Moreover, we included the network initialization time into the total runtime which explains slightly increasing values in the case of small models. For very small models we did not perform the distributed computation at all. On the other hand, there were large models that could not be verified without usage of external memory (swapping) with a small number of workstations. In these cases we do not give the runtime as well (denoted by "–" in the table).

There are two noteworthy points that could be drawn from our experiments. Firstly, in the presence of an accepting cycle in the graph, the cycle is 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). Secondly and most importantly, for some systems (e.g. *Leader election*), the graph contains very few back-level edges (possibly none) and therefore our parallel algorithm for *full* LTL model-checking is practically as efficient as a distributed state space generation algorithm.

6. Conclusions and Future Work

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 search state space generation with a back-level controlled depth-first search for accepting cycles. The aim of this paper was to describe the basic idea of the algorithm. Experimental results obtained with a prototype implementation have been encouraging, and demonstrated its practicality.

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 tends to be much shorter in comparison with those computed by a depth-first search based algorithm. 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).

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.

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 [3], the problem of LTL model-checking is reduced to detecting negative cycles in a weighted directed graph. Recently, in [4] 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 [4] 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.

For future research we aim to perform an extensive case-study analysis and a direct comparison to the other approaches cited above. We also intend to implement other heuristics to speedup the algorithm, in particular we would like to exploit the specification of the system in the similar way we were able to utilize the verified property and to prune the state space graph (those branches that have been fully explored need not be explored again).

Last but not least, our new algorithm is compatible with the partial order reduction method leading thus to additional signifficant impact on the efficiency of the algorithm. It is generally difficult to achieve full compatibility of partial order reduction with distributed LTL model-checking algorithms, in particular when the strategy used to generate the state space is based on DFS. The problematic part is how to enforce the *cycle condition* (at least one vertex on each cycle must be fully expanded). However, a simple heuristics for fulfilling the cycle condition in BFS approach is obvious. It involves full expansion of all start vertices of back-level edges. As back-level edges need not belong to a cycle this can cause that more states than necessary are fully expanded, getting thus lower reduction. However, if cycle detection would be performed on all back-level edges, unwanted expansions could be eliminated. We intend to include the partial order reduction into our implementation and perform detailed measurements in the future.

References

[1] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 217–234. Springer-Verlag, 2001.

[2] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedinfs of the 3rd International Workshop on Verification and Computational Logic (VCL'02 – held at the PLI 2002 Symposium)*, pages 1–10. University of Southampton, UK, Technical Report DSSE-TR-2002-5 in DSSE, 2002.

[3] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In R. Hariharan, M. 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.

[4] I. Cerná and R. Pelánek. Distributed explicit fair cycle detection. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 49–73. Springer-Verlag, 2003.

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

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

[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. *Software Engineering*, 23(5):279–295, 1997.

[11] 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*, Berlin, 1999. Springer-Verlag.

[12] R. 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. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.