FI MU

Faculty of Informatics
Masaryk University Brno

Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking

by

Luboš Brim
Ivana Černá
Pavel Moravec
Jiří Šimša

FI MU Report Series **FIMU-RS-2004-09**

 October 2004

Publications in the FI MU Report Series are in general accessible via WWW:

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

Further information can obtained by contacting:

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

Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking*

Luboš Brim
brim@fi.muni.cz

Ivana Černá
cerna@fi.muni.cz

Pavel Moravec
xmoravec@fi.muni.cz

Jiří Šimša
xsimsa@fi.muni.cz

October 11, 2004

Abstract

We present a new distributed-memory algorithm for enumerative LTL model-checking that is designed to be run on a cluster of workstations communicating via MPI. The detection of accepting cycles is based on computing maximal accepting predecessors and the subsequent decomposition of the graph into independent predecessor subgraphs induced by maximal accepting predecessors. Several optimizations of the basic algorithm are presented and the influence of the ordering on the algorithm performance is discussed. Experimental implementation of the algorithm shows promising results.

1 Introduction

Model-checking has become a very practical technique for automated verification of computer systems due to its push-button character and has been applied fairly successfully for verification of quite a few 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).

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

The use of distributed and/or parallel processing to combat the state explosion problem gained interest in recent years (see e.g. [4, 5, 10, 11, 12, 15]). For large industrial models, the state space does not completely fit into the main memory of a single computer and hence model-checking algorithm becomes very slow as soon as the memory is exhausted and system starts swapping. A typical 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 such as MPI. 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 approach to distributed 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 [16] and successful implementation within several verification tools. The basic idea is to associate a Büchi automaton with the verified LTL formula so that the automaton accepts exactly all the computations of the given model satisfying the formula. This makes possible to reduce the model-checking problem to the emptiness problem for Büchi automaton. A Büchi automaton accepts a word if and only if there is an *accepting state* reachable from the initial state and from itself.

Courcoubetis et al. [9] proposed an elegant way to find accepting states that are reachable from themselves (to compute *accepting cycles*) by employing a *nested depth first search*. The first search is used to search for reachable accepting states while the second one (*nested*) tries to detect accepting cycles. Our aim is to solve the LTL model-checking problem by distribution, i.e. by utilizing several interconnected workstations. The standard sequential solution as described above is based on the depth-first search (DFS), in particular the *postorder* as computed by DFS is crucial for cycle detection. However, when exploring the state space in parallel, the DFS postorder is not generally maintained any more due to different speeds of involved workstations and communication overhead.

The extremely high effectiveness of the DFS based model-checking procedure in the sequential case is due to a simple and easily computable criterion characterizing the existence of a cycle in a graph: a graph contains a cycle if and only if there is a *back-edge*. A distributed solution requires other appropriate criteria to be used as the DFS based ones do not have the same power in the distributed setting. E.g. in [1] the authors

proposed to use *back-level edges* as computed by breadth first search (BFS) as a necessary condition for a path to form a cycle. The reason, why such a criterion works well in a distributed environment is that BFS search can be (unlike DFS) reasonably parallelized. In [7] the used criterion is that each state on an accepting cycle is reachable from an accepting state. Every state can be tested for this criterion independently and thus the algorithm is well distributable. Another example of a necessary condition suitable for distribution and used in [3] employs the fact that the graph to be checked is a product of two graphs and it can contain a cycle only if one of the component graphs has a cycle.

The main idea of our new approach to distributed-memory LTL model-checking has born from a simple observation that all states on a cycle have exactly the same predecessors. Hence, having the same set of predecessors is a necessary condition for two states to belong to the same cycle and the membership in its own set of predecessors is a necessary condition for a state to belong to a cycle. In particular, in case of accepting cycles we can restrict ourselves to accepting predecessors only. Even more, it is not necessary to compute and store the entire set of accepting predecessors for each state, it is sufficient to choose a suitable representative of the set of all accepting predecessors of a given state instead. It is crucial that the cycle-check becomes significantly cheaper if representatives are used. We consider an ordering of states and we choose as a representative of a set of accepting predecessors the accepting predecessor which is maximal with respect to this ordering, called *maximal accepting predecessor*. A necessary condition for a graph to contain an accepting cycle is that there is an accepting state with itself as maximal accepting predecessor. However, this is not a sufficient condition as there can exist an accepting cycle with "its" maximal accepting predecessor lying outside of it. For this reason we systematically re-classify those accepting vertices which do not lie on any cycle as non-accepting and re-compute the maximal accepting predecessors.

The main technical problem is how to compute maximal accepting predecessors in a distributed environment. Our algorithm repeatedly improves the maximal accepting predecessor for a state as more states are considered. This requires propagating a new value to successor states each time the maximum has changed. In this way the procedure resembles the relaxation procedure as used in the single source shortest path problem. The main advantage of such an approach is that relaxations can be performed in an arbitrary order in a BFS manner, hence in parallel. There is even another source of parallelism in our algorithm. Maximal accepting predecessors define independent subgraphs induced by vertices with the same maximal accepting predecessor. These

subgraphs can be explored simultaneously and again in an arbitrary order. In both cases a re-distribution of the graph among the workstations involved in the distributed computing might be necessary to optimize the performance of the algorithm.

Another distinguished feature of the algorithm is that due to the breadth-first exploration of the state space the counter-examples produced by the algorithm tend to be short, which is very important for debugging.

There are several known approaches to distribution and/or parallelization of the explicit-state LTL model-checking problem and we relate our algorithm to other work in Section 6.

2 Model-Checking and Accepting Cycles

In the automata-based approach to LTL model-checking [16], one constructs a Büchi automaton $\mathcal{A}_{\neg\Psi}$ for the negation of the property Ψ one wishes to verify and takes its product with the Büchi automaton modeling the given system S. The system (more exactly the model) is correct with respect to the given property if and only if the product automaton recognizes an empty language, i.e. no computation of S violates Ψ. The size of the product automaton is linear with respect to the size of the model and exponential with respect to the size of Ψ.

The model-checking problem is thus reduced to the *emptiness* problem for automata. It can be reduced even further to a graph problem [8]. Let $\mathcal{A} = (\Sigma, S, \delta, s, \mathrm{Acc})$ be a Büchi automaton where Σ is an input alphabet, S is a finite set of states, $\delta : S \times \Sigma \to 2^S$ is a transition relation, s is an initial state and $\mathrm{Acc} \subseteq V$ is a set of accepting states. The automaton $\mathcal{A}$ can be identified with a directed graph $G_{\mathcal{A}} = (V, E, s, A)$, called *automaton graph*, where $V \subseteq S$ is a set of vertices corresponding to all *reachable states* of the automaton $\mathcal{A}$, $E = \{(u, v) \mid u, v \in V \text{ and } v \in \delta(u, a) \text{ for some } a \in \Sigma\}$, $s \in V$ is a distinguished initial vertex corresponding to the initial state of $\mathcal{A}$ and A is a distinguished set of accepting vertices corresponding to reachable accepting states of $\mathcal{A}$.

Definition 2.1. *Let* $G = (V, E, s, A)$ *be an automaton graph. The* reachability relation $\rightsquigarrow^+ \subseteq V \times V$ *is defined as* $u \rightsquigarrow^+ v$ *iff there is a directed path* $< u_0, u_1, \ldots, u_k >$ *where* $u_0 = u$, $u_k = v$ *and* $k > 0$.

A directed path $< u_0, u_1, \ldots, u_k >$ *forms a* cycle *if* $u_0 = u_k$ *and the path contains at least one edge. A cycle is* accepting *if at least one vertex on the path* $< u_0, u_1, \ldots, u_k >$ *belongs to the set of accepting vertices* A.

Note that according our definition every cycle in an automaton graph is reachable from the initial vertex.

Theorem 2.2. [8] Let $\mathcal{A}$ be a Büchi automaton and $G_{\mathcal{A}}$ its corresponding automaton graph. Then $\mathcal{A}$ recognizes a nonempty language iff $G_{\mathcal{A}}$ contains an accepting cycle.

In this way the original LTL model-checking problem is reduced to the accepting cycle detection problem for automaton graphs and we formulate our model-checking algorithm as a distributed algorithm for accepting cycle detection problem. The algorithm is based on the notion of predecessors. Intuitively, an automaton graph contains an accepting cycle iff some accepting vertex is a predecessor of itself.

To avoid computing of all predecessors for each vertex we introduce a concept of *maximal accepting predecessor*, denoted by *map*. We pre-suppose a linear ordering of the set of vertices given e.g. by their numbering. Other possible orderings are discussed in Section 4. From now on we therefore assume that for any two vertices u, v we can decide which one is greater. Furthermore, we extend the ordering to the set $V \cup \{null\}$ ($null \notin V$) and put $null < v$ for all $v \in V$.

Definition 2.3. *Let* $G = (V, E, s, A)$ *be an automaton graph. A* maximal accepting predecessor function *of the graph* G, $map_G : V \to (V \cup \{null\})$, *is defined as*

$$map_G(v) = \begin{cases} \max\{u \in A \mid u \rightsquigarrow^+ v\} & \text{if } \{u \in A \mid u \rightsquigarrow^+ v\} \neq \emptyset \\ null & \text{otherwise} \end{cases}$$

Corollary 2.4. *For any two vertices* $u, v \in V$, *the vertices cannot lie on the same cycle whenever* $map_G(u) \neq map_G(v)$.

The definition of the maximal accepting predecessor function *map* gives the sufficient condition characterizing the existence of an accepting cycle in the automaton graph.

Lemma 2.5. *Let* $G = (V, E, s, A)$ *be an automaton graph. If there is a vertex* $v \in V$ *such that* $map_G(v) = v$ *then the graph* G *contains an accepting cycle.*

The opposite implication is not generally true, for a counterexample see the graph in Figure 1. The accepting cycle $2 \rightsquigarrow^+ 2$ is not revealed due to the greater accepting predecessor 4 outside the cycle. However, as the state 4 the does not lie on *any* cycle, it can be safely deleted from the set of accepting states and the accepting cycle will still be discovered in the resulting graph. This idea is formalized in the notion of a *deleting*

transformation. Whenever the deleting transformation is applied to an automaton graph G with $map_G(v) \neq v$ for all $v \in V$, it shrinks the set of accepting vertices by deleting those ones which evidently do not lie on any cycle.

Definition 2.6. *Let $G = (V, E, s, A)$ be an automaton graph and map_G its maximal accepting predecessor function. A* deleting transformation*, del, is defined as $del(G) = (V, E, s, \overline{A})$, where $\overline{A} = A \setminus \{u \in A \mid \exists v \in V. map_G(v) = u\}$.*

Directly from the definition we have the following result.

Lemma 2.7. *Let G be an automaton graph and v an accepting vertex in G such that $map(v) > v$. Then v is an accepting vertex in $del(G)$.*

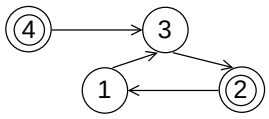

Figure 1: Undiscovered cycle

Note that the application of the deleting transformation can result in a different *map* function. For the graph G given in Figure 1, $del(G)$ has state 2 as its only accepting state, hence $map_{del(G)}(2) = 2$ (and the existence of the accepting cycle is certified by the new function).

The next Lemma states formally the invariance property just exemplified, namely that the application of the deleting transformation to a graph with an accepting cycle results in a graph having an accepting cycle as well.

Lemma 2.8. *Let $G = (V, E, s, A)$ be an automaton graph containing an accepting cycle and such that $map(v) \neq v$ for every $v \in A$. Then the graph $del(G)$ contains an accepting cycle.*

Proof: Let C be an accepting cycle in G and $v \in C$ be an accepting vertex. For every successor u of v we have $map(u) \geq map(v) > v$. Therefore the vertex v is accepting in $del(G)$. The transformation does not change the set of vertices and edges and the conclusion follows. □

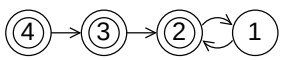

Figure 2: Deleting transformation

It can happen that even in the transformed graph $del(G)$ there is no vertex such that its *map* value would certify the existence of an accepting cycle. This situation is depicted in Figure 2. However, after a finite number of applications of the deleting transformation an accepting cycle will be certified.

Definition 2.9. *Let G be an automaton graph. For $i \in \mathbb{N}$ a graph G^i is defined inductively as $G^0 = G$ and $G^{i+1} = del(G^i)$. The set of accepting vertices of G^i is denoted A^i.*

Lemma 2.10. *Let* $G = (V, E, s, A)$ *be an automaton graph containing an accepting cycle. Then there is a natural number* $i \in \mathbb{N}$ *and a vertex* $v \in V$ *such that* $map_{G^i}(v) = v$.

Proof: Let C be an accepting cycle in G and $u \in A$ be the maximal accepting vertex on C. For any $j \in \mathbb{N}$ let R^j be a set of accepting predecessors of u in G^j, $R^j = \{v \in A^j \mid v \rightsquigarrow^+ u\}$. If $map_{G^j}(u) > u$, then obviously $|R^j| > |R^{j+1}|$. Since R^0 is finite, there is an index i for which $|R^{i+1}| = |R^i|$ and $map_{G^i}(u) = u$. In other words, after at most $|R^0| - 1$ applications of the deleting transformation on G the *map* value of u changes to u. □

Putting together Lemma 2.5 and Lemma 2.10 we can state the main theorem justifying the correctness of our algorithm.

Theorem 2.11. *Let* $G = (V, E, s, A)$ *be an automaton graph. The graph* G *contains an accepting cycle if and only if there is a natural* $i \in \mathbb{N}$ *and a vertex* $v \in V$ *such that* $map_{G^i}(v) = v$.

Note that for an automaton graph without accepting cycles the repetitive application of the deleting transformation results in an automaton graph with an empty set of accepting states.

This theory leads to the first naive version of the algorithm. Its structure is depicted in the Figure 3. At first the function *map* for given automaton graph is computed (line 3), while the procedure $MAP(G)$ is responsible for detection of accepting cycle. If no accepting cycle is found, than vertices from *shrinkA* are removed from the set of accepting states. Computing the set *shrinkA* can be performed concurrently with computing *map* in the procedure $MAP(G)$.

Described steps are performed, until some accepting cycle is reached or the set A is empty. Let denote the main *while*-cycle on lines 2-5 as the *iteration of algorithm*.

```
1 proc MAIN(G)  //G = (V, E, s, A)
2   while A ≠ ∅ do
3         MAP(G)
4         A = A \ shrinkA
5   od
6   return NO CYCLE
7 end
```

Figure 3: Basic structure of the algorithm

These results can be extended in the following way. Assume the first application of function *map* which doesn't find any accepting cycle. Than due to Corollary 2.4 if there is some accepting cycle in the graph, all its vertices have the same value *map*. Hence we can split the input graph into parts where all vertices with the same value *map* are included in one component called *predecessor subgraph*. Now each cycle is included in one component only. Hence we can continue in searching accepting cycles separately and independently in different predecessor subgraphs. This is formalized in the following definition.

Definition 2.12. *Let* $G = (V, E, s, A)$ *is automaton graph and* $shrinkA = \{v_1, v_2, \ldots\ldots v_n\}$, $n \in \mathbb{N}$. *Let define the* decomposition of G to predecessor subgraphs $G_0, G_1, G_2, \ldots, G_n$, *where* G_0 *is subgraph of* G *induced by vertices* $\{v \mid map(v) = null\} \setminus shrinkA$ *and for all* $i : 1 \leq i \leq n$, G_i *is subgraph of* G *induced by vertices* $\{v \mid map(v) = v_i\} \cup \{v_i\}$

For computing function *map* recursively to predecessor subgraphs, we can postulate similar theorems like Lemma 2.5, Lemma 2.10 and Theorem 2.11. Following algorithm computes function *map* recursively as is described, hence its proof of correctness is based on modified Theorem 2.11.

3 Distributed Detection of Accepting Cycles

It is now apparent how to make use of the *map* function and the deleting transformation to build an algorithm which detects accepting cycles. We first present a straightforward approach with the aim to introduce clearly the essence of our distributed algorithm (Subsection 3.1). The distributed-memory algorithm which employs several additional optimizations is presented in Subsection 3.2 and finally, the correctness and complexity of the algorithm is discussed in Subsection 3.3. We do not explicitly describe the actual distribution of the algorithm as this is quite direct and follows the standard technique used e.g. in [1, 6].

3.1 The Algorithmic Essence

The code is rather self-explanatory, we add a few additional comments only. The *MAP* procedure always starts by initializing the *map* value of the initial vertex to *null*, all the other vertices are assigned the undefined initial *map* value, denoted by $\perp$. Every time a vertex receives a new (greater) *map* value, the vertex is pushed into a *waiting* queue and

the new *map* value is propagated to all its successors. If an accepting vertex is reached for the first time (line 15) the vertex is inserted into the set *shrinkA* of vertices to be removed from A by the deleting transformation. However, if the accepting vertex is reached from a greater accepting vertex (lines 16 and 17) this value will be propagated to all its successors and the vertex is removed from the set *shrinkA* (Lemma 2.7).

```
1  proc Main(G)   //G = (V, E, s, A)
2    while A ≠ ∅ do
3          MAP(G)
4          A := A \ shrinkA
5    od
6    report (NO ACCEPTING CYCLE exists)
7  end

8  proc MAP(G)
9    foreach u ∈ V do map(u) := ⊥ od
10   map(s) := null
11   waiting.push(s)
13   while waiting ≠ ∅ do
14         u := waiting.pop()
15         if u ∈ A then if map(u) < u then propagate := u; shrinkA.add(u)
16                                     else propagate := map(u);
17                                          shrinkA.remove(u)
18                   fi
19               else propagate := map(u)
20         fi
21         foreach (u, v) ∈ E do
22           if propagate = v then report (ACCEPTING CYCLE found) fi
23           if propagate > map(v) then map(v) := propagate
24                                     waiting.push(v) fi
25         od
27   od
28 end
```

3.2 Distributed Algorithm

To build up an effective distributed algorithm we consider two optimizations of the above given basic algorithm. The first one comes out from the fact that every time the set of accepting states has been shrunk and a new *map* function is going to be computed,

the algorithm from 3.1 needs to traverse the whole graph, update the flags for vertices removed from the set of accepting vertices, and re-initialize the *map* values to $\perp$.

The second improvement is more important with respect to the distribution and it is a consequence of Corollary 2.4. An accepting cycle in G can be formed from vertices with the same maximal accepting predecessor only. A graph induced by the set of vertices having the same maximal accepting predecessor will be called *predecessor subgraph*. It is clear that every strongly connected component (hence every cycle) in the graph is completely included in one of the predecessor subgraphs. Therefore, after applying the deleting transformation the new *map* function can be computed separately and independently for every predecessor subgraph. This allows for speeding up the computation (values are not propagated to vertices in different subgraphs) and for an efficient distribution of the computation.

In the distributed algorithm *CycleDetection* (see Figure 4) we first compute in parallel the *map* function on the given input graph G (line 2). If no accepting cycle is detected and the set *shrinkA* of vertices to be removed from the set of accepting vertices is nonempty, then the vertices from *shrinkA* define predecessor subgraphs. Every predecessor subgraph is identified through the accepting vertex (*seed*) which is the common maximal accepting predecessor for all vertices in the subgraph. Seeds are stored in the *waitingseed* queue and are used as a parameter when calling the *DistributedMAP* procedure. After the *map* function is computed for every predecessor subgraph, the vertices that should be deleted from the set of accepting vertices form a new content of the *waitingseed* queue.

Vertices from the same predecessor subgraph are identified with the help of the *oldmap* value. For every vertex v, $oldmap(v)$ maintains the value of $map(v)$ from the previous iteration. When a vertex v with $map(v) = seed$ (line 31) is reached the value of $oldmap(v)$ is set to *seed*. Accepting predecessors are propagated only to successors identified to be in the same predecessor subgraph through the variable *oldmap* (line 35). Sets *waiting* and *shrinkA* are maintained in the same way as in the basic algorithm presented in Subsection 3.1.

For the distributed computation we assume a network of collaborating workstations with no global memory. Communication between workstations is realized by sending messages only. In the distributed computation the input graph is divided into parts, one part per each workstation.

```
1  proc CycleDetection(G)   //G = (V, E, s, A)
2    MAP(G)
3    waitingseed := shrinkA
4    shrinkA := ∅
5    while waitingseed ≠ ∅ do
6          while waitingseed ≠ ∅ do
7                seed := waitingseed.pop()
8                DistributedMAP(G, seed)
9          od
10         waitingseed := shrinkA
11         shrinkA := ∅
12   od
13   report (NO ACCEPTING CYCLE exists)
15 end

16 proc DistributedMAP(G, seed)
17   oldmap(seed) := seed
18   map(seed) := null
19   waiting.push(seed)
20   while waiting ≠ ∅ do
21         u := waiting.pop()
22         if (u ∈ A) ∧ (u ≠ oldmap(u))
23           then if map(u) < u then propagate := u
24                                  shrinkA.add(u)
25                              else propagate := map(u)
26                                  shrinkA.remove(u) fi
27            else propagate := map(u)
28         fi
29         foreach (u, v) ∈ E do
30           if propagate = v then report (ACCEPTING CYCLE found) fi
31           if map(v) = oldmap(u)
32             then oldmap(v) := oldmap(u)
33                  map(v) := propagate
34                  waiting.push(v)
35            else if (propagate > map(v)) ∧ (oldmap(v) = oldmap(u))
36                   then map(v) := propagate
37                        waiting.push(v)
38                  fi
39           fi
40         od
41   od
42 end
```

Figure 4: Distributed Cycle Detection Algorithm

In the *CycleDetection* algorithm every workstation has local data structures *waitingseed*, *waiting* and *shrinkA* and computes the values of the *map* function for its part of the graph. Workstations have to be synchronized every time the computation of the *map* function is finished and the set of accepting vertices is to be shrunk.

An important characteristic of the distributed algorithm is that the *map* values for different predecessor subgraphs can be computed in parallel, i.e. the procedure *DistributedMAP* can be called for different values of *seed* in parallel.

Another distinguished feature of our distributed algorithm is the possibility to make use of dynamic re-partitioning, i.e. of a new assignment of vertices to workstations after each iteration. The *map* function induces a decomposition of the graph into predecessor subgraphs. After a new *map* function is computed the graph can be re-partitioned so that the new partition function respects predecessor subgraphs as much as possible which can result in significant reduction in the communication among the workstations as well as in speed-up of the entire computation.

In the case the given graph contains an accepting cycle an output reporting such a cycle is required. The proposed algorithm can be simply extended to report an accepting cycle. Let v be a vertex certifying the existence of an accepting cycle ($v = propagate$). Then two distributed searches are initiated. The first one finds a path from the initial vertex s to v and the second one a path from v to itself. In the second search the predecessor subgraph of v is searched-through only.

3.3 Correctness and Complexity

Lemma 3.1. *Procedure DistributedMAP computes exactly for given predecessor subgraph the function map.*

Proof: Clear from the pseudocode.

Theorem 3.2. *The CycleDetection algorithm terminates and correctly detects an accepting cycle in an automaton graph.*

Proof: It is sufficient to prove "Algorithm reports cycle iff $\exists u \in V, i \in \mathbb{N} : map^i(u) = u$".
"$\Rightarrow$": Assume the algorithm detects accepting cycle with accepting vertex u during this iteration. Due to the property of *map*-values propagation of the algorithm, u has to be reachable from itself, hence there is some accepting cycle.
"$\Leftarrow$": Assume $\exists u \in V, n \in \mathbb{N} : map^n(u) = u$. Since the algorithm computes function *map*

and divides graph in each iteration correctly, at most the n-th iteration of the algorithm has to reveal an accepting cycle. □

Theorem 3.3. *The time complexity of the CycleDetection algorithm is* $\mathcal{O}(a^2 \cdot m)$*, where* m *is the number of edges and* a *is the number of accepting vertices in the input (automaton) graph.*

Proof: The cycle in the *CycleDetection* procedure is repeated at most a times. Every vertex is pushed to the *waiting* queue in the procedures *DistributedMAP* and *MAP* at most a times and all successors of a vertex popped from the *waiting* queue are tested. The overall complexity of both *MAP* and *DistributedMAP* is $\mathcal{O}(a \cdot m)$. □

Experiments with model-checking graphs (see Section 5) demonstrate that the actual complexity is typically significantly lower.

4 Ordering of Vertices

One of the key aspects influencing the overall performance of our distributed algorithm is the underlying ordering of the vertices used by the algorithm. The direct way to order the vertices is to use the enumeration order as it is computed in the enumerative on-the-fly model-checking. The first possibility is to order the vertices by the time they have been reached (sooner visited vertices receive smaller values). In this case the algorithm tends to return short counterexamples and generally detects the accepting cycles very quickly. Moreover, since the graph is split into "as many" subgraphs "as possible", less iterations are performed. On the other hand, the running time of each iteration increases, because the vertices with small values will be usually updated several times. Alternatively, we can employ the reverse ordering (sooner visited vertices receive larger values). The behavior of the algorithm is now completely different. Both the size of subgraphs and the number of iterations increase, while the number of the subgraphs as well as the running time of each iteration decrease. As a third possibility we can consider a combination of these two orderings, which can result in fast computation with small number of iterations.

Another set of heuristics can be based on different graph traversal algorithms (e.g. depth-first search or breadth-first search). Data structure *waiting* was mentioned as a queue for the sake of better distribution. For using partial order reduction, where is necessary to detect cycles, the depth first search will be better.

Finally, yet another simple heuristic is to compare the bit-vector representations of vertices. In the future we plan to implement, compare and systematically evaluate all the orderings of vertices mentioned above.

In our implementation each vertex is identified by a vector of three numbers – the workstation identifier, the row number in the hash table, and the column number in the row. The ordering of vertices is given by the lexicographical ordering of these triples. Note that there are six possible lexicographical orderings and by reversing these orderings one gets another six possibilities. This gives us a range of twelve possible orderings. We have implemented and compared six of them. The results we obtained show that there is no real difference among these six approaches, which in some sense demonstrate the robustness of an ordering with respect to the random partitioning of the graph among the workstations.

5 Experiments

We have implemented the distributed algorithm described in Section 3.2. The implementation has been done in C++ and the experiments have been performed on a network of thirteen Intel Pentium 4 2.6 GHz workstations with 1 GB of RAM each interconnected with a fast 100Mbps Ethernet and using tools provided by our own distributed verification environment – DiVinE.

The vertices have been partitioned among the workstations using random hash function and no re-partitioning was implemented. Messages were buffered and sent in packets containing 100 messages.

We performed three sets of experiments with different examples of model checking problems. First set of experiments evaluates the scalability of the algorithm *CycleDetection*. Second set compares this algorithm with *token-based distributed-memory nested depth-first search algorithm* (NDFS). The latest set of experiments was concerned to the influence of ordering of vertices to algorithm performance.

For scalability experiments we choose following models. First was a variant of the *Mutual exclusion protocol* problem based on a token ring and parametrized by the number n of processes (denoted by $\mathtt{TR}(n)$). Verified property was $\mathtt{GF}(P_0.\mathtt{CS})$, i.e. the process P_0 enter its critical section infinitely many times.

Next model was the solution of the *Producer-consumer protocol* problem parametrized by the number n of messages which can be lost in a row (denoted by $\mathtt{PC}(n)$).

The property being checked over the PC class was $GF(Consumer.consu- me_0 \vee Consumer.consume_1)$, i.e. the consumer will consume some value infinitely many times.

Third experiments were performed on the model of "Rether" (Real-time Ethernet protocol, $RE(n)$) with parameter n as number of communicating nodes. Here we checked two properties: reservation will be handled next turn: $G(No- de_0.reserved \Rightarrow (\neg cendU(cendU(\neg cend \wedge \wedge(Node_0.RT\_actionR\neg cend)))))$ (where $cend := Token.cycle\_end$) for the group of models $RE(n, 1)$ and property $GFNode_0.NRT\_action$ (i.e. communicating node $Node_0$ infinitely often performs some action) for the group of models $RE(n, 2)$.

The latest model was experimental version of *elevator* denoted as $EL(n)$ with a parameter n as number of floors. The property being checked over the EL class was $G(Person_1.waiting \Rightarrow FPerson_1.in\_elevator)$, i.e. whenever some person in the ground-floor waits for a lift, he/she will be served sometimes.

All properties have been satisfied by the respective models. The characterization of corresponding automata graphs are depicted in the Table 1.

model	**vertices**	**edges**	**min. #**	**min (s)**	**max (s)**
TR(15)	1474559	17432561	3	130	846
PC(20)	1021822	4516966	3	71	172
RP(8,4)	1662938	2711144	1	52	455
RP(10,1)	5759277	6248113	2	64	327
EL(15)	1125218	2433015	1	307	467

Table 1: The characterization of models

The results of the experiments are presented in Figure 5 and all the results are taken as an average of 5 executions of the distributed algorithm. Because of the size of state graphs (up to 6 millions vertices and the amount of memory needed to store a vertex description), we did not get results when running the algorithm on less than 3 workstations due to memory restrictions. Therefore, the shown speedups are calculated relative to 3 workstations instead of one. We found that we gain a linear speedup for reasonably large graphs.

The second set of tests was designed to evaluate the actual performance of the algorithm. We have implemented an experimental version of the *token-based distributed-memory nested depth-first search algorithm* (*Nested DFS*) and compared the running time of

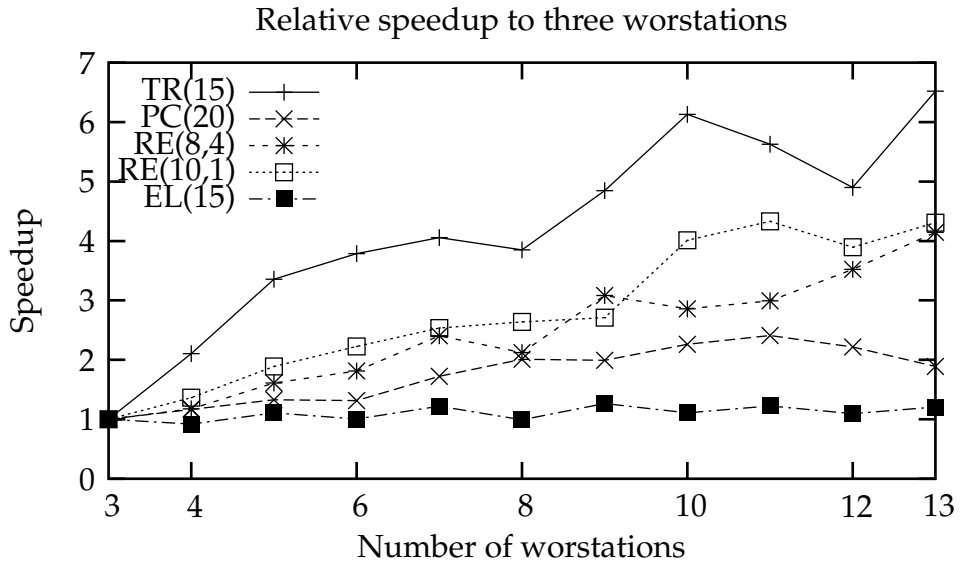

Figure 5: Scalability of the distributed algorithm

both algorithms. The comparison of our algorithm (*DACD*) and the Nested DFS algorithm (*NDFS*) is given in Table 2 for various numbers of workstations (*NW*) involved in the distributed computation. The results shown are running times in seconds. It can be seen that our algorithm outperforms the Nested DFS algorithm even when the number of workstation is small.

NW	**NDFS**	**DACD**	**Speedup**
3	1251	846	1.5
4	1801	402	4.5
5	1610	252	6.4
6	1958	223	8.8
7	1904	208	9.2
8	2132	219	9.7
9	2166	174	12.4
10	2306	137	16.8
11	2376	150	15.8
12	2465	173	14.2
13	2589	130	19.9

Table 2: Nested DFS vs. DACD on PC(20)

We have compared the sequential version of our algorithm to the sequential Nested DFS algorithm as well. As expected, the sequential version of our algorithm performs slightly worse. However, the experiments have demonstrated comparability of both approaches. Our algorithm needs, on average, around 30% more time and memory than Nested DFS algorithm.

We have also considered verification problems on models with an error (e.g. *Dining philosophers*, various models of an elevator, and some communication protocols). Since our algorithm is entirely based on the breadth-first search, the counterexamples were much more shorter than counterexamples provided by the Nested DFS algorithm. Moreover, in all cases the accepting cycle was detected very early by our algorithm (within tens of seconds), while the Nested DFS algorithm was incomparably slower. For the parametrized models where the size of the state space was larger than the size of the (distributed) memory (e.g. for forty dining philosophers), our algorithm detected a counterexample, while the Nested DFS algorithm failed due to memory limitations. These results were almost independent on the ordering of vertices chosen and on the number of workstations involved.

In the erroneous version of the general *Peterson algorithm* for 4 processes, where the error is very "deep" (according to the breadth-first search level). In this case the Nested DFS algorithm detected the counterexample very early, since the depth-first search tends to follow the erroneous path, while our algorithm failed. In several other examples with similar characteristics our algorithm was able to detect the error as well, but later than the Nested DFS algorithm. However, the "depth of an error" is typically small, hence our distributed algorithm will outperform the Nested DFS algorithm in most cases.

The third group of tests was concerned to the influence of ordering of vertices to algorithm performance. We performed larger number of experiments, the most representative are depicted in Figure 6 and Figure 7 for models TR(14) and PC(20) respectively. From that figures we can conclude, that even random orderings result into quite different behavior of the algorithm.

The last interesting conclusion we would like to point out is that the number of iterations in all models without an error was up to 20. On the other hand, an error was already detected during the first iteration in all performed tests. As a consequence the algorithm is usually able to detect the faulty behavior without exploring the entire graph (state space).

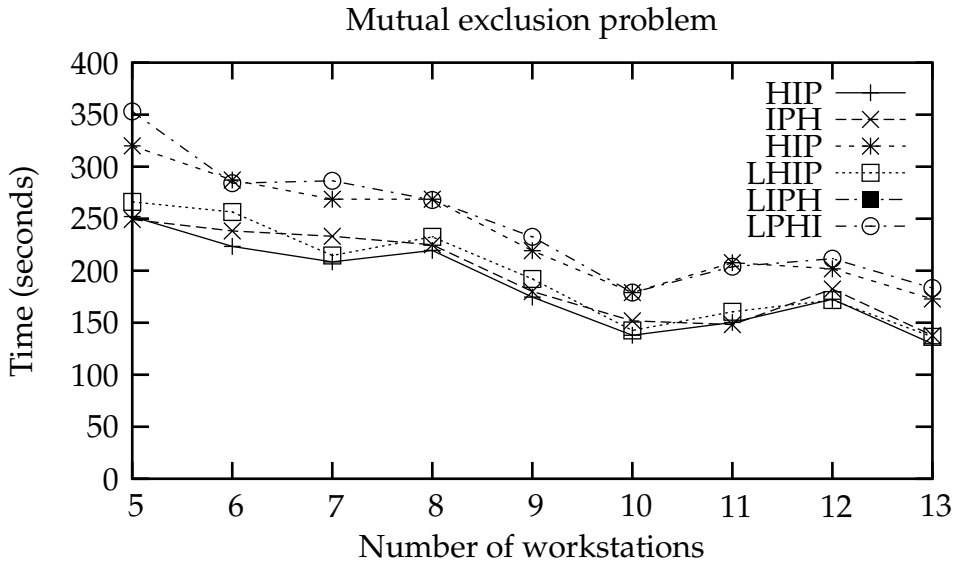

Figure 6: Comparison of orderings on model TR(14)

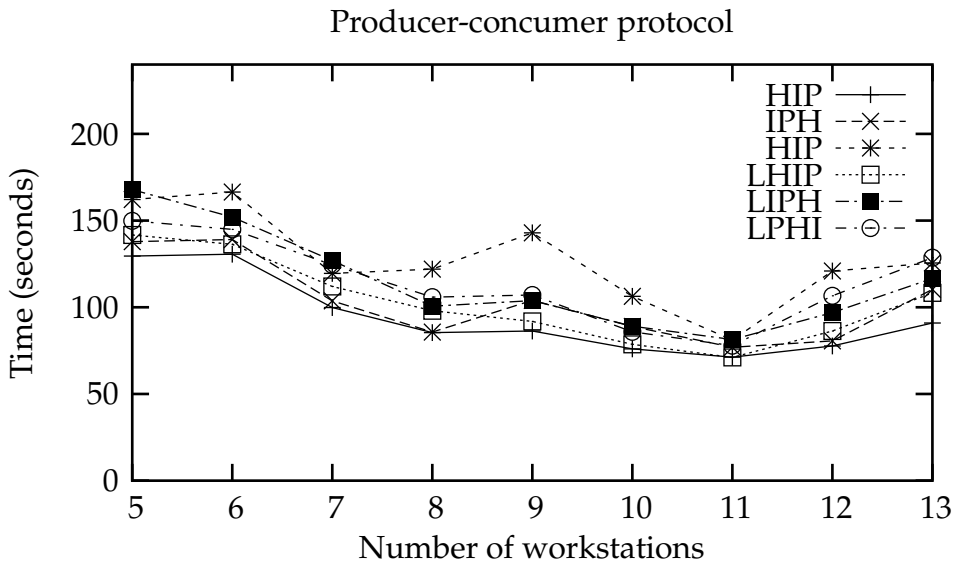

Figure 7: Comparison of orderings on model PC(20)

6 Conclusions

In this paper, we have presented a new distributed-memory algorithm for enumerative LTL model-checking.

We plan to implement two improvements of our algorithm. Both use additional conditions characterizing the existence of an accepting cycle in an automaton graph augmented with the maximal accepting predecessors information.

Suppose that the graph contains an accepting cycle such that the maximal accepting predecessor of this cycle is outside of it. Then there must exist a vertex on the cycle with the in-degree at least two. One of the incoming edges comes from the cycle, a different one comes from the maximal accepting predecessor. Therefore, we do not need to explore a predecessors subgraph which does not fulfill this condition.

For the second condition suppose again that the graph contains an accepting cycle such that the maximal accepting predecessor of this cycle is outside of it. Then the graph must contain at least one another accepting vertex (besides the maximal accepting predecessor). It is possible to combine these two methods. An effective way to check the conditions requires a more sophisticated techniques for computing the set *shrinkA* in the distributed environment.

There are several already known approaches to distributed-memory LTL model-checking. In [14] a distributed implementation of the SPIN model checker, restricted to perform model-checking of safety properties only is described. In [2], the authors build on the safety model-checking work of [14] to create a distributed-memory version of SPIN that does full LTL model-checking. The disadvantage of this algorithm is that it performs only one nested search at a time. Recently, in [7] 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 [13]. 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. Our algorithm is in some sense similar to [7], although their original ideas are different. Both algorithms work in iterations started from a set of accepting vertices. In general, the time complexity of [7] is better ($\mathcal{O}(n \cdot m)$ in comparison to $\mathcal{O}(a^2 \cdot m)$), but our algorithm has three advantages. It is adjustable according to the input problem by setting some special ordering of vertices, it can guess the counterexample very quickly before the whole graph is traversed and it has one instead of two synchronizations during the iteration cycle. Since the number of iterations is very similar, on the larger and slower nets this can be a significant factor. Similar arguments are valid if comparing our algorithm to another recently proposed algorithm [1]. This algorithm uses back-level edges to discover cy-

cles, works on-the-fly and is effective in finding bugs. All these three algorithms could be meant not to replace but to complement each other.

In [6], the problem of LTL model checking is reduced to detecting negative cycles in a weighted directed graph. Since the basic method (edge relaxation) is the same, the behavior of both algorithms will be generally similar. The algorithm in [6] suffers by clumsy cycle detection, our approach needs costly synchronization and many searches are often redundantly called.

For each of the above mentioned distributed-memory algorithm for the enumerative LTL model-checking there will most likely exist a set of input problems on which it is superior to the others. Our future work will be focused on systematic, mainly experimental, comparison of these algorithms.

References

[1] 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.

[2] 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'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[3] 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.

[4] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In L. Brim and O. Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89.1. Elsevier, 2003.

[5] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN'02)*, volume 2318 of *LNCS*, pages 128–147. Springer, 2002.

[6] 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, 2001.

[7] 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, 2003.

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

[9] 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.

[10] 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'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[11] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 12–21. IEEE Computer Society Press, 1999.

[12] T. Heyman, O. Grumberg, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *15th International Conference (CAV'03)*, volume 2725 of *LNCS*, pages 54–66. Springer, 2003.

[13] 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, *Proc. of the Fourth International Workshop CAV'92*, pages 396–409. Springer, 1993.

[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, Berlin, 1999. Springer.

[15] R. Palmer and Ganesh Gopalakrishnan. A distributed partial order reduction algorithm. In D. Peled and M. Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*, volume 2529 of *LNCS*, page 370. Springer, 2002.

[16] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science (LICS'86)*, pages 332–344. Computer Society Press, 1986.