How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors

L. Brim [1], I. Černá [2], P. Moravec, J. Šimša

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

Distributed automata-based LTL model-checking relies on algorithms for finding accepting cycles in a Büchi automaton. The approach to distributed accepting cycle detection as presented in [9] is based on maximal accepting predecessors. The ordering of accepting states (hence the maximality) is one of the main factors affecting the overall complexity of model-checking as an imperfect ordering can enforce numerous re-explorations of the automaton. This paper addresses the problem of finding an optimal ordering, proves its hardness, and gives several heuristics for finding an optimal ordering in the distributed environment. We compare the heuristics both theoretically and experimentally to find out which of these work well.

1 Introduction

Over the past decade, many techniques using distributed and/or parallel processing have been developed to combat the computational complexity of verification problems. They cover reachability analysis [3,14,17,21], verification of branching time logics [4,5,7,8,12,15], linear time logics [1,2,10], equivalence checking [6,18], and other verification problems.

In this paper we concentrate on the technique of maximal accepting predecessor for LTL model-checking as presented in [9]. We show how this technique can be extended and optimised to speed-up LTL model-checking in a distributed environment.

The maximal accepting predecessors (MAP) algorithm comes out from the automata approach which reduces the LTL model-checking problem to the emptiness problem for Büchi automata. A Büchi automaton accepts a

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

[2] Research supported by the Academy of Sciences of Czech Republic grant No. 1ET408050503

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

non-empty language if and only if there is a reachable *accepting cycle* in the Büchi automaton graph.

Reachability is a graph exploration technique that can be efficiently parallelised. The *MAP* algorithm exploits reachability for cycle detection in the distributed environment. The algorithm is derived from the observation that all vertices on a cycle have the same set of predecessors. To avoid computing sets of all predecessors the algorithm assigns to every vertex a single representative predecessor. Another core idea of the algorithm is to make use of vertex ordering to determine suitable representatives. Namely, supposing the vertices of the graph are ordered, the representative is the maximal accepting predecessor of the vertex (or null value if there is none). A sufficient condition for a graph to contain an accepting cycle is that there is an accepting vertex with itself as the maximal accepting predecessor. Unfortunately, this is not a necessary condition as there can exist an accepting cycle with "its" maximal accepting predecessor lying outside of it. For this reason the algorithm systematically re-classifies those accepting vertices which do not lie on any cycle as non-accepting and re-computes the maximal accepting predecessors. The overall complexity of the *MAP* algorithm is mainly derived from both computing the representatives and the number of iterations in which vertices are re-classified and the representatives are re-computed. It turns out that the vertex ordering is of crucial importance for improving the performance of the algorithm.

In [9] a few basic vertex orderings have been considered, a systematic exposition of vertex orderings and its impact on the algorithm effectiveness has been left open. In this paper we investigate the influence of the vertex ordering in detail. First of all, we introduce the notion of an *optimal ordering* as the ordering for which the *MAP* algorithm terminates in the very first iteration, i.e. without re-classifying the representatives. The optimal ordering can be computed for example by depth-first search traversal of the graph. However, as we prove, the problem itself is P-complete and its efficient distributed solution is not at hand (Section 3). Therefore, we formulate several heuristics to resolve the ordering problem in a distributed environment and investigate their theoretical properties (Section 4). All heuristics went through a detailed experimental evaluation (Section 5) giving a deeper insight into their practical usability in the distributed verification.

2 Maximal Accepting Predecessors

In this section, we recapitulate the main idea of the *MAP* algorithm as presented in [9], concentrating on the impact of vertex ordering on the complexity of the algorithm.

The *MAP* algorithm follows the automata-based approach to LTL model-checking [22]. The verification problem is reduced to the *emptiness* problem for Büchi automata and is represented as a graph problem. Let $\mathcal{A} =$

$(\Sigma, S, \delta, s, Acc)$ be a Büchi automaton where Σ is an input alphabet, S is a finite set of states, $\delta : S \times \Sigma \rightarrow 2^S$ is a transition relation, s is an initial state and $Acc \subseteq S$ 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 an *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 $\langle u_0, u_1, \ldots, u_k \rangle$ in G where $u_0 = u$, $u_k = v$ and $k > 0$.

A directed path $\langle u_0, u_1, \ldots, u_k \rangle$ 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 $\langle u_0, u_1, \ldots, u_k \rangle$ belongs to the set of accepting vertices A.

A Büchi automaton recognises a non-empty language iff its automaton graph contains an accepting cycle. The *MAP* algorithm detects accepting cycles by maximal accepting predecessors. It assumes a linear ordering $\prec$ on the set V of vertices. The ordering is extended to the set $V \cup \{null\}$ $(null \notin V)$ by setting $null \prec v$ for all $v \in V$.

Definition 2.2 Let $G = (V, E, s, A)$ be an automaton graph. A *maximal accepting predecessor function* of the graph G, $map_G : V \rightarrow (V \cup \{null\})$, is defined as

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

If there is a vertex $v \in V$ with $map_G(v) = v$, the algorithm reports an accepting cycle. However, it can happen that the graph contains an accepting cycle and for all $v \in V$ the inequality $map_G(v) \neq v$ holds. As all vertices on a cycle must have the same maximal accepting predecessor, this can only happen if this predecessor lies outside the cycle. Such a vertex can be removed from the set of accepting vertices without violating the existence of an accepting cycle in the graph. This idea is formalised 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 the vertices which evidently do not lie on any cycle.

Definition 2.3 Let $G = (V, E, s, A)$ be an automaton graph and map_G its maximal accepting predecessor function. A *deleting transformation* is defined as $del(G) = (V, E, s, \overline{A})$, where $\overline{A} = A \setminus \{u \in A \mid map_G(u) \prec u\})$.

Note that the application of the deleting transformation can result in a different *map* function but it preserves the property *"the graph contains an accepting cycle"*. The *MAP* algorithm alternately computes the *map* function

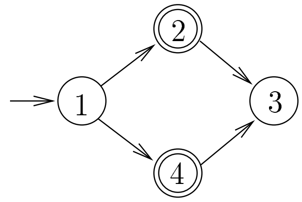


Fig. 1. Deleting transformation

and applies the deleting transformation till an accepting cycle is discovered or the set of accepting states is empty.

MAP Algorithm
while $A \neq \emptyset$ **do**
compute map_G;
if $(\exists u \in A : map_G(u) = u)$
then return CYCLE
else $G = del(G)$;
fi
od
return NO CYCLE

In our original algorithm [9] the deleting transformation has been defined using the set $\{u \in A \mid \exists v \in V.map_G(v) = u\}$ of accepting vertices to be removed. The new formulation of the deleting transformation used here is more appropriate in the context of optimising vertex ordering as it generally removes more vertices. E.g. consider the graph on Figure 2 with two accepting vertices 2 and 4 and the vertex ordering given by their numbers. The algorithm terminates in two iterations under the original definition (in the first iteration the vertex 4 is deleted, in the second one the vertex 2 is deleted) while it needs only one iteration to terminate under the new definition (both accepting vertices are deleted at once as $map_G(2) = null \prec 2$ and $map_G(4) = null \prec 4$). The correctness of the modified algorithm can be easily proved following similar arguments as given in [9].

3 Optimal Vertex Ordering for the *MAP* Algorithm

The time complexity of the distributed *MAP* algorithm is $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices and m is the number of edges in the automaton graph. Here the factor $a{\cdot}m$ comes from the computation of the *map* function and the factor a relates to the number of iterations, i.e., computations of the *del* function. In order to optimise the complexity one aims to decrease the number of iterations by choosing an appropriate vertex ordering. A natural way how to order the vertices is to use the enumeration order as it is computed in the enumerative on-the-fly model-checking. In [9], each vertex was identified with 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 was given by the lexicographical ordering of these triples. In this section, we define the notion of an optimal ordering and prove that the optimal ordering problem is P-complete.

Let $\prec$ be a linear ordering on vertices used by the algorithm *MAP* and $iter_{\prec}$ be the number of iterations of the main cycle till the algorithm *MAP* terminates.

Definition 3.1 An ordering $\prec$ is *optimal* iff $iter_{\prec} = 1$.

The optimality of an ordering is tightly related to a reachability relation on the set of accepting vertices.

Definition 3.2 An ordering $\prec$ *respects reachability* iff for all $u, v \in A$, whenever $(u \leadsto^{+} v \wedge v \not\leadsto^{+} u)$ then $u \prec v$.

Lemma 3.3 *If an ordering $\prec$ respects reachability then it is optimal.*

Proof. We prove that non-optimal ordering does not respect reachability.

Suppose the ordering $\prec$ is not optimal and there is an accepting cycle in the graph G. The algorithm does not detect an accepting cycle in the first iteration if for all accepting vertices u the value $map_G(u) \neq u$. Let v be the maximal accepting vertex lying on a cycle. Then $v \prec map_G(v)$, $map_G(v) \leadsto^{+} v$, and $v \not\leadsto^{+} map_G(v)$. Therefore $\prec$ does not respect reachability.

If there is no accepting cycle in the graph, then there is an accepting vertex v which is not re-classified as non-accepting after the first iteration of the *MAP* algorithm. It means that $v \prec map_G(v)$ and $map_G(v) \leadsto^{+} v$. From acyclicity we have $v \not\leadsto^{+} map_G(v)$, which implies that $\prec$ does not respect reachability.

Lemma 3.4 *For every automaton graph there is an optimal ordering. Moreover, an optimal ordering can be computed in time $\mathcal{O}(a \cdot m)$.*

Proof. We give algorithm which computes an optimal ordering. As a first step, the algorithm computes the reachability relation $R = \{(u, v) \mid u, v \in A, u \leadsto^{+} v\}$. This computation can be done for example by running a reachability procedure from all accepting vertices separately which takes time $\mathcal{O}(a \cdot m)$.

Now, if the graph does not contain any accepting cycle, then for $u, v \in A$ we put $u \prec v$ if and only if $(u, v) \in R$. Other pairs of vertices are ordered arbitrarily. If the graph contains an accepting cycle, then there is a vertex u with $(u, u) \in R$. Let $v \prec u$ for every accepting vertex v, $v \neq u$. Other pairs of vertices are again ordered arbitrarily.

Notice, that a graph can have several optimal orderings, as the ordering of non-accepting vertices and of accepting vertices, which are mutually unreachable, is not important.

The question is whether an optimal ordering can be computed more efficiently in the distributed environment. We provide a strong evidence that the computation of an optimal ordering cannot be significantly speeded up by the use of any reasonable number of parallel processors. Namely, we prove that the *optimal ordering problem* is P-complete and thus inherently sequential. A problem is P-complete if it belongs to P and every language $L \in P$ is log-space reducible to the problem (see [13] for details on P-completeness).

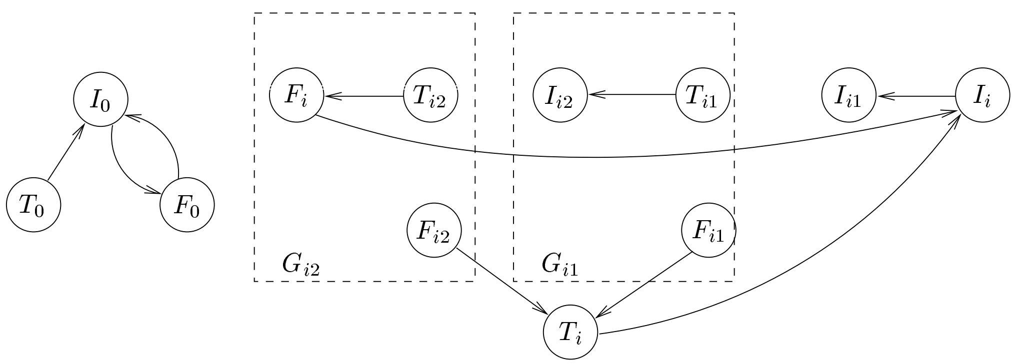
The optimal ordering problem is to decide for a given automaton graph and two accepting vertices u, v whether u precedes v in *every* optimal ordering of graph vertices. Lemma 3.4 shows that the optimal ordering problem is in P. We prove P-hardness by reduction from the NAND circuit value problem.

A *NAND boolean circuit* is a sequence $B = (B_0, \ldots, B_n)$ where $B_0 = 1$ and $B_i = \neg(B_{i_1} \vee B_{i_2})$, $i_1, i_2 < i$. Let $value(B_0) = true$, $value(B_i) = \neg(value(B_{i_1}) \vee value(B_{i_2}))$, and $value(B) = value(B_n)$. The *NAND circuit value (NANDCV) problem* is to decide for a given NAND boolean circuit B whether $value(B) = true$. Ladner [16] shows that the NANDCV problem is P-complete.

Theorem 3.5 *The optimal ordering problem is P-hard.*

Proof. By log-space reduction of the NANDCV problem to the optimal ordering problem. Let $B = (B_0, \ldots, B_n)$ be a NAND boolean circuit. We construct an automaton graph G and identify its two vertices u, v in such a way that u precedes v in every optimal ordering of graph vertices if and only if $value(B) = true$.

First, for each B_i we construct a graph G_i inductively. The graph $G_0 = (\{T_0, I_0, F_0\}, \{(T_0, I_0), (I_0, F_0), (F_0, I_0)\})$ is depicted in Figure 2a). Let $B_i = \neg(B_{i_1} \vee B_{i_2})$. Then G_i contains as its subgraphs G_{i_1} and G_{i_2}, new vertices T_i, I_i, F_i, and new edges as depicted in Figure 2b).



a) graph G_0 **b)** graph G_i

Fig. 2. Construction of the automaton graph

We prove that for all $i = 0, \ldots, n$ the graph G_i has specific reachability properties. Namely,

$$\text{if } value(B_i) = true \text{ then } T_i \leadsto^+ I_i \leadsto^+ F_i,\ F_i \leadsto^+ I_i,\ I_i \not\leadsto^+ T_i, \text{ and } F_i \not\leadsto^+ T_i,$$
$$\text{if } value(B_i) = false \text{ then } F_i \leadsto^+ I_i \leadsto^+ T_i,\ T_i \leadsto^+ I_i,\ I_i \not\leadsto^+ F_i, \text{ and } T_i \not\leadsto^+ F_i.$$

The assertion can be proved by induction on i. For $i = 0$, $value(B_0) = true$ and the assertion can be easily checked following Figure 2a).

For the induction step let us suppose $value(B_i) = true$. Then $value(B_{i_1}) = value(B_{i_2}) = false$ and by induction hypothesis there are paths from I_{i_1} to T_{i_1} and from I_{i_2} to T_{i_2}. These paths together with edges (T_i, I_i), (I_i, I_{i_1}), (T_{i_1}, I_{i_2}), and (T_{i_2}, F_i) form a path from T_i to F_i in G_i. On the other hand, as there is no path from I_{i_1} to F_{i_1} in G_{i_1} neither from I_{i_2} to F_{i_2} in G_{i_2}, there is no path both from I_i and F_i to T_i in G_i.

The case $value(B_i) = false$ divides into three subcases depending on values of $value(B_{i_1})$ and $value(B_{i_2})$, all subcases are handled analogously to the previous case.

To finish the proof of P-hardness of the optimal ordering problem, let us reduce the NAND boolean circuit B to the automaton graph G containing as its subgraph G_n, a new initial vertex S and edges from S to all vertices in G_n. Vertices T_n and F_n are accepting. From properties of G_n we have that if $value(B) = true$ then $T_n \rightsquigarrow^+ F_n \wedge F_n \not\rightsquigarrow^+ T_n$ and if $value(B) = false$ then $F_n \rightsquigarrow^+ T_n \wedge T_n \not\rightsquigarrow^+ F_n$. We claim that $value(B) = true$ iff in every optimal ordering T_n precedes F_n. Clearly, if $value(B) = true$ and F_n preceded T_n, then $map(T_n) = null$, $map(F_n) = T_n$, and the MAP algorithm would need two iterations to complete the cycle detection. For the opposite implication, if $value(B) = false$, then ordering in which F_n precedes T_n is optimal as $map(F_n) = null$ and $map(T_n) = F_n$. To conclude the proof we observe that the construction of the graph G can be done in space logarithmic with respect to the circuit size.

4 Heuristics for vertex ordering

As the optimal ordering problem is P-complete, we cannot expect the computation of an optimal ordering in the distributed environment to be significantly more efficient than in the sequential setting. Therefore we aim for non-optimal orderings. In this section, we describe several heuristics for computing a vertex ordering. All but one are easily computable in the distributed environment. For all orderings we indicate how "far" is the computed ordering from the optimal one. We elaborate a quantitative measure that characterizes the distance.

Definition 4.1 Let $\prec$ be an ordering and $\gamma = \langle u_1, \ldots, u_n \rangle$ be a path in G. Then $(u_{i_1}, \ldots, u_{i_k})$ is a *reverse subsequence* of the sequence $(u_1, \ldots, u_n)$ if $u_{i_1}, \ldots, u_{i_k}$ are accepting vertices and $u_{i_k} \prec \ldots \prec u_{i_2} \prec u_{i_1}$. The maximal length of a reverse subsequence of the path γ is the *index of the path* γ, $index_\prec(\gamma)$.

Index of a vertex u is defined as $index_\prec(u) = \max\{index_\prec(\gamma) \mid \gamma \text{ is a path from the initial vertex to the vertex } u \text{ in } G\}$.

Index of an automaton graph G is defined as $index_\prec(G) = \max\{index_\prec(u) \mid u \text{ is a vertex in } G\}$.

To illustrate the definition, let $\gamma = \langle 4, 2, 3, 5, 1\rangle$ be the path depicted on Figure 3 and $1 \prec 2 \prec 3 \prec 4 \prec 5$. Then $(4,2)$, $(4,3)$, $(4,3,1)$, and $(3,1)$ are reverse subsequences of the sequence $(4,2,3,5,1)$. On the other hand, the sequences $(4,2,3,1)$ and $(5,1)$ are not reverse subsequences of γ. Index of the path γ is 3.

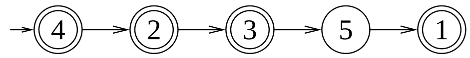


Fig. 3. Path with reverse subsequence $(4,3,1)$

Theorem 4.2 *For a graph G and a vertex ordering $\prec$, $iter_{\prec} = index_{\prec}(G)$.*

Proof. To prove the inequality $index_{\prec}(G) \leq iter_{\prec}$ let us assume there is a vertex u with $index_{\prec}(u) > iter_{\prec}$. Let $\sigma = (u_1, \ldots u_k)$ be a reverse subsequence of a path from s to u with $|\sigma| = index_{\prec}(u)$. Then at least two vertices u_i, u_j $(i < j)$ have to be deleted from A during the same deleting transformation. But $u_i \rightsquigarrow^+ u_j$, $u_j \prec u_i$ and therefore $u_j \prec map(u_j)$. This contradicts the definition of the deleting transformation.

For the opposite inequality $index_{\prec}(G) \geq iter_{\prec}$, let u be a vertex and $\gamma = \langle s, \ldots u\rangle$ be a path such that $index_{\prec}(\gamma) = index_{\prec}(u) = index_{\prec}(G) = k$. Let $\sigma = (u_1, u_2, \ldots u_k)$ be the reverse subsequence of the maximal length of the path γ. By induction on the index i we prove that the vertex u_i is removed from the set of accepting vertices during the ith iteration of the algorithm *MAP*.

For $i = 1$ the assertion follows from the maximality of γ. For the induction step assume that the vertex u_{i-1} was removed during the $(i-1)$th iteration. If u_i is not removed from the set of accepting vertices during the ith iteration then there is a vertex $v_i \in A$ with $s \rightsquigarrow^+ v_i \rightsquigarrow^+ u_i$ and $u_i \prec v_i$ (i.e. in the ith iteration $u_i \prec map(u_i)$). The vertex v_i is re-classified as non-accepting not sooner than during the ith iteration and we can repeat similar arguments for the vertex v_i. As a result we have vertices $u_i \prec v_i \prec v_{i-1} \prec \ldots v_1$ with $s \rightsquigarrow^+ v_1 \rightsquigarrow^+ \ldots v_i \rightsquigarrow^+ u_i$. Hence $(v_1, v_2, \ldots v_i, u_i, \ldots u_k)$ is a reverse subsequence with $k+1$ vertices of a path from s to u. This contradicts the maximality of γ and σ.

Now we define several vertex orderings which are based on different ways of graph traversal. All but the first one are envisaged to be appropriate for the distribution.

Definition 4.3 Let G be an automaton graph.

$\prec_{DFS}$: Suppose the graph G is traversed by depth first search (DFS). We define $u \prec_{DFS} v$ iff the vertex u is backtracked by DFS *later* than the vertex v (i.e., $\prec_{DFS}$ is the reverse of DFS-postorder).

$\prec_{BFS}$: Suppose the graph G is traversed by breadth first search (BFS). We define $u \prec_{BFS} v$ iff the vertex u is visited by BFS *before* the vertex v.

$\prec_{BFSpreds}$: Suppose the graph G is traversed by BFS. Let G' be the breadth first search tree. Let $visit(u) = (acc\_preds, BFS_{nr})$, where acc_preds is the number of accepting predecessors of the vertex u in G' and BFS_{nr} is the time when the vertex u is visited by BFS. We define $u \prec_{BFSpreds} v$ iff $visit(u)$ is lexicographically smaller than $visit(v)$.

The difference between $\prec_{BFSpreds}$ and $\prec_{BFS}$ is shown in Figure 4. In both graphs the successors of the initial vertex are proceeded from left to right. For the left hand side graph $iter_{\prec_{BFSpreds}} = 2$ and $iter_{\prec_{BFS}} = 1$ (since $a \prec_{BFS} b$, but $b \prec_{BFSpreds} a$) while for the right hand side graph $iter_{\prec_{BFSpreds}} = 1$ and $iter_{\prec_{BFS}} = 2$ (since $d \prec_{BFSpreds} c$, but $c \prec_{BFS} d$).

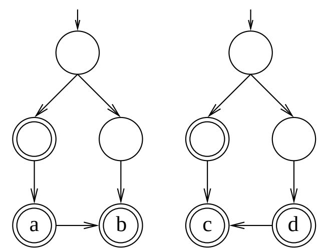


Fig. 4. Comparison of $\prec_{BFSpreds}$ and $\prec_{BFS}$

For the next ordering suppose the graph G is divided into subgraphs G_1, $G_2,\ldots,$ G_n. Further suppose G is traversed by a modified depth first search (cDFS) which differs from DFS in traversing cross edges (edges with vertices from distinct subgraphs). For each subgraph, cDFS maintains a queue of vertices from which it starts a local DFS. A local DFS traverses only the respective subgraph. When a cross edge is encountered, its endpoint is enqueued to the respective queue and the search backtracks. cDFS is initiated with a local DFS from an initial vertex and terminates when no local DFS is running and all queues are empty. A straightforward way to distribute the computation of cDFS is to place subgraphs G_1, $G_2,\ldots,$ G_n on different computers and run local DFSs in parallel.

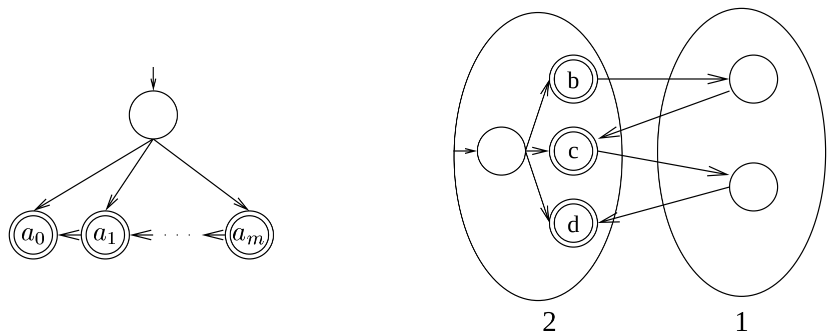
$\prec_{cDFS}$: Suppose the graph G is traversed by cDFS. For $u \in G_i$, $v \in G_j$ we define $u \prec_{cDFS} v$ iff $i < j$ or ($i = j$ and u is backtracked later than v).

Lemma 4.4 *$\prec_{DFS}$ is an optimal ordering, i.e., $index_{\prec_{DFS}}(G) = 1$.*

Proof. According to Lemma 3.3 it suffices to prove that $\prec_{DFS}$ respects the reachability relation. Let $u, v \in A$, $u \leadsto^+ v$ and $v \not\leadsto^+ u$. If u is visited by DFS before v, then u is backtracked after all its successors and therefore $u \prec_{DFS} v$. If u is visited later than v, then v must have been backtracked before u was reached, because there is no path from v to u. Hence $u \prec_{DFS} v$. The optimality of $\prec_{DFS}$ corresponds with P-completeness of the DFS problem [20].

Lemma 4.5 *For each $\prec \in \{\prec_{BFS}, \prec_{BFSpreds}, \prec_{cDFS}\}$ there is an automaton graph G such that $index_{\prec}(G) = |A|$.*

Proof. Graph certificating the upper bound for $\prec_{BFS}$ and $\prec_{BFSpreds}$ is depicted in Figure 5a) (successors of the initial vertex are traversed from left to right, $a_0 \prec_{BFS} a_1 \prec_{BFS} \dots a_m$ and $a_0 \prec_{BFSpreds} a_1 \prec_{BFSpreds} \dots a_m$) and for $\prec_{cDFS}$ in Figure 5b) (successors of the initial vertex are traversed bottom up, $d \prec_{cDFS} c \prec_{cDFS} b$).



a) upper bounds for $\prec_{BFS}$ and $\prec_{BFSpreds}$ b) upper bound for $\prec_{cDFS}$

Fig. 5. Upper bounds

5 Experiments

We have implemented variants of the *MAP* algorithm using vertex orderings described in the previous section. The experiments have been performed on a network of ten Intel Pentium 4 2.6 GHz workstations with 1 GB of RAM each interconnected with a 100Mbps Fast Ethernet and using tools provided by our own distributed verification environment DiVinE [11].

Name	**Vertices**	**Acc. Vertices**	**Error**
Elevator10_1	891372	307692	NO
LookUpProc8_2	1954569	1458848	NO
PublicSubscribe_1	2051215	204612	NO
Rether10_4	11325003	5649118	NO
Rether08_2	2898644	850689	YES
PLCshedule600_1	5096287	3827319	YES
Lifts4_1	998570	331596	NO
Phils14L_3	7193116	2410147	NO
TrainGate8_2	17572372	11668232	YES
Peterson3Err_1	1135804	796734	YES

Table 1
Summary of graphs

TrainGate8_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	89	69	45	24	10
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	116	67	34	23	16
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	77	65	26	20	14
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	–	–	1417	855	744
	Iter.	–	–	1	1	1

PLCshedule600_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	9	109	45	62	13
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	7	9	3	14	18
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	3	3	2	3	3
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	–	820	588	450	242
	Iter.	–	1	1	1	1

Peterson3Err_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	81	127	52	70	65
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	167	387	246	216	165
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	116	213	114	98	72
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	141	162	219	129	114
	Iter.	1	1	1	1	1

Rether08_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	86	70	32	40	31
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	465	285	146	158	93
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	342	136	88	131	95
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	465	281	232	186	129
	Iter.	1	1	1	1	1

Table 2
Experimental results: Graphs containing accepting cycles

In order to examine the performance of the algorithm, we performed an extensive experimental evaluation using graphs representing various verification problems. The graphs are identified in Table 1 along with their most important characteristics – the number of reachable vertices and the number of reachable accepting vertices. The column *Error* indicates the presence or absence of an accepting cycle in the graph. Most of the graphs could not be stored on a single computer.

We compared vertex orderings $\prec_{BFS}$, $\prec_{BFSpreds}$, and $\prec_{cDFS}$. Moreover, there are several natural vertex orderings derived from the random hash function used for storing states (see [9] for more details). We used the best one from [9], denoted $\prec_{RND}$, as a “benchmark” for the comparison with newly presented orderings.

Detailed results of all experiments are reported in Tables 2 and 3. For every graph and every ordering we performed experiments on various numbers of workstations. For each setup we give the number of iterations performed by the algorithm and its run time in seconds. The run time represents an average taken from several runs. The sign ’–’ means that the setup resulted

Elevator10_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	295	193	167	153	119
	Iter.	14	14	14	14	14
$\prec_{\mathbf{BFS}}$	Time	296	265	346	382	208
	Iter.	5	7	8	10	8
$\prec_{\mathbf{BFSpreds}}$	Time	159	130	119	117	90
	Iter.	3	4	4	4	4
$\prec_{\mathbf{cDFS}}$	Time	841	530	637	294	294
	Iter.	33	48	49	33	48

PublicSubscribe_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	152	92	67	66	50
	Iter.	8	8	8	8	8
$\prec_{\mathbf{BFS}}$	Time	159	97	72	56	52
	Iter.	4	6	6	6	6
$\prec_{\mathbf{BFSpreds}}$	Time	152	91	67	64	52
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	336	195	285	195	142
	Iter.	7	8	8	8	8

Lifts4_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	225	112	76	67	60
	Iter.	12	10	8	8	10
$\prec_{\mathbf{BFS}}$	Time	227	121	91	73	60
	Iter.	3	4	4	4	3
$\prec_{\mathbf{BFSpreds}}$	Time	299	242	190	121	105
	Iter.	4	4	5	5	4
$\prec_{\mathbf{cDFS}}$	Time	397	225	360	216	151
	Iter.	11	21	26	26	28

Lup8_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	714	678	266	245	196
	Iter.	12	12	12	12	12
$\prec_{\mathbf{BFS}}$	Time	1640	866	547	508	365
	Iter.	5	7	9	8	9
$\prec_{\mathbf{BFSpreds}}$	Time	427	293	185	167	129
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	1780	1038	1354	995	690
	Iter.	34	41	38	48	45

Phils14L_3		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	2718	1983	2220	3269	2709
	Iter.	11	11	11	11	11
$\prec_{\mathbf{BFS}}$	Time	3444	2606	4812	1935	2813
	Iter.	4	4	9	6	8
$\prec_{\mathbf{BFSpreds}}$	Time	3430	1735	2597	1427	1226
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	-	6237	6304	5635	5121
	Iter.	-	13	12	12	14

Rether10_4		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	–	–	–	1130	722
	Iter.	–	–	–	20	20
$\prec_{\mathbf{BFS}}$	Time	–	–	–	1390	945
	Iter.	–	–	–	10	11
$\prec_{\mathbf{BFSpreds}}$	Time	–	–	–	594	406
	Iter.	–	–	–	4	5
$\prec_{\mathbf{cDFS}}$	Time	–	–	–	5692	15278
	Iter.	–	–	–	165	171

Table 3
Experimental results: Graphs without accepting cycles

in a computation which does not finish due to memory limitations.

In the case of graphs with an accepting cycle, all computations performed only one iteration. In other words, an optimal ordering was found immediately. Although this may seem strange from a theoretical point of view, there is some experimental evidence for this. The number of iterations is bounded by the *quotient graph height*. The quotient graph of $G = (V, E)$ is a graph (W, H), such that W is the set of strongly connected components of G and $(C_1, C_2) \in H$ if and only if $C_1 \neq C_2$ and there exist $r \in C_1, s \in C_2$ such that $(r, s) \in E$. The *height* of the quotient graph is the length of the longest path in the quotient graph. As argued in [19], the quotient graph height is for model checking graphs typically low and thus the *MAP* algorithm tends to have only a few iterations. In the presence of an accepting cycle, the number of iterations is typically just one.

Furthermore, in some cases you can notice that a computation on fewer workstations takes less time than a computation on more workstations. These irregularities are caused by the hash function used for partitioning and are not related to the algorithm's behaviour.

Yet another observation drawn from the experiments is that in some cases the number of iterations necessary to finish the computation is quite different under different orderings, but the resulting times are very close. This is caused by the uneven number of re-explorations during one iteration. However, lower number of iterations generally results in a faster computation.

As for the orderings, though $\prec_{BFS}$ and $\prec_{BFSpreds}$ are both based on the BFS traversal, $\prec_{BFSpreds}$ outperformed $\prec_{BFS}$ in most experiments. In fact, our experiments suggest the $\prec_{BFSpreds}$ ordering to be the best one among the compared orderings.

The $\prec_{cDFS}$ ordering can be considered from the theoretical point of view as a promising one, as it tries to mimic the optimal $\prec_{DFS}$ ordering. However, it fails to scale well. The high number of iterations is caused by the direct influence of graph distribution on vertex ordering and by the high number of cross edges in the distributed graph. Due to these reasons is the positive impact of distribution dampened.

The random ordering $\prec_{RND}$ can be classified as a "better average". It is interesting to note that despite its randomness it sometimes outperforms orderings which have been designed to employ specific graph features.

Finally, for the $\prec_{BFS}$, $\prec_{BFSpreds}$ and $\prec_{RND}$ orderings the algorithm works on-the-fly as it simultaneously computes the *map* function and performs cycle detection. The experiments clearly demonstrated that in the presence of an accepting cycle, the algorithm was able to detect it during the first iteration. Thus it was not necessary to generate the whole graph. For graphs without accepting cycles the number of workstations had typically small impact on the number of iterations (except for the ordering $\prec_{cDFS}$).

6 Conclusions

The paper complements the distributed LTL model-checking algorithm *MAP* arising out from the maximal accepting predecessors concept. First, we prove that for every graph there is an optimal ordering of graph vertices for which the *MAP* algorithm terminates in one iteration. The optimal ordering can be computed in time linear to the size of the graph, however the problem itself is P-complete and thus hard to parallelise. Therefore we provide and evaluate several heuristics computing a vertex ordering on-the-fly and such that they are easy to incorporate into the distributed *MAP* algorithm.

Conclusions both from theoretical and experimental evaluation are that none of the heuristics outperforms the others. On average, the most reliable heuristic is $\prec_{BFSpreds}$ (based on breadth first search) followed by $\prec_{RND}$ based on (random) hashing.

The presented approach to the optimisation of the time complexity of the *MAP* algorithms aims at decreasing the number of iterations of the algorithm. An alternative direction is to optimise the computation of the *map* function in each iteration. This computation is based on the relaxation of graph edges (in the same way as in the Belmann-Ford algorithm) and we do not find this too promising.

References

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

[2] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[3] G. Behrmann. Distributed Reachability Analysis in Timed Automata. *Software Tools for Technology Transfer*, 7(1):19–30, 2005.

[4] A. Bell and B. R. Haverkort. Sequential and Distributed Model Checking of Petri Net Specifications. In *Parallel and Distributed Model-Checking (PDMC'02)*, volume 68.4 of *ENTCS*. Elsevier, 2002.

[5] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 390–404. Springer, 2000.

[6] S. Blom and S. Orzan. Distributed state space minimization. In *Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *ENTCS*. Elsevier, 2003.

[7] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free μ-Calculus. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[8] B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation-Free mu-Calculus. In *SPIN Workshop on Model checking of Software (SPIN'02)*, volume 2318 of *LNCS*, pages 128–147. Springer, 2002.

[9] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

[10] I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection. In *SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[11] DiVinE – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[12] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[13] R. Greenlaw, H. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[14] 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 *Petri Nets and Performance Models (PNPM'99)*, pages 12–21. IEEE Computer Society Press, 1999.

[15] T. Heyman, O. Grumberg, and A. Schuster. A Work-Efficient Distributed Algorithm for Reachability Analysis. In *Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 54–66. Springer, 2003.

[16] R. E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[17] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

[18] S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.

[19] R. Pelánek. Typical structural properties of state spaces. In *SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[20] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[21] U. Stern and D. L. Dill. Parallelizing the Murφ Verifier. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.

[22] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press, 1986.