Distributed LTL Model Checking Based on Negative Cycle Detection*

Luboš Brim, Ivana Černá, Pavel Krčál, and Radek Pelánek

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
{brim,cerna,xkrcal,xpelanek}@fi.muni.cz

Abstract. This paper addresses the state explosion problem in automata based LTL model checking. To deal with large space requirements we turn to use a distributed approach. All the known methods for automata based model checking are based on depth first traversal of the state space which is difficult to parallelise as the ordering in which vertices are visited plays an important role. We come up with entirely different approach which is dependent on locating cycles with negative length in a directed graph with real number length of edges. Our method allows reasonable distribution and the experimental results confirm its usefulness for distributed model checking.

1 Introduction

Model checking is a very successful technique for verifying concurrent systems and many verification tools were proposed in the last two decades. These tools verify a desired behavioural property of a reactive system over a given model through exhaustive enumeration of all the states reachable by the system and the behaviours that traverse through them. As a matter of fact, the main limiting factor in applications of such tools to practical verification problems is the real computational power available (time and especially memory). Therefore verification of complex concurrent systems requires techniques to avoid the state-explosion problem [9]. Several sequential methods (partial order reductions, on-the-fly search) to overcome this barrier have been proposed and successfully implemented in automatic verification tools. Recently, some attempts to use multiprocessors and networks of workstations have been undertaken.

In [23] the authors describe a parallel version of the verifier Murφ. The table of all reached states is partitioned over the nodes of the parallel machine and the explicit state enumeration is performed in parallel. A similar approach to distributed reachability analysis has been taken in [18]. A distributed version of the UPPAAL model checker based on the same idea as parallel Murφ has been reported in [3]. Yet another distributed reachability algorithm has been proposed in [1], but has not been implemented. We stress that all mentioned

* This work has been partially supported by the Grant Agency of Czech Republic grants No. 201/00/1023 and 201/00/0400.

algorithms solve only the reachability problem and do not admit the complete linear time model checking. A distributed version of the LTL model checker SPIN [16] based on nested depth first search approach has been explored in [2]. Other recent papers attempt to use distributed environment of workstations for parallel symbolic model checking. [15] presents a parallel reachability analysis algorithm based on BDDs while in [4] distributed symbolic method has been applied to check safety RCTL properties. Papers [14, 5] significantly extend the scope of properties that can be verified by presenting distributed symbolic model checking for μ-calculus and alternation free μ-calculus.

In automata based LTL model checking the verification problem is represented as the emptiness problem of a Büchi automaton which turns out to be equivalent to finding a cycle reachable from an initial state and containing an accepting state in the graph corresponding to the Büchi automaton. The best known algorithm for finding cycles in directed graphs is the Tarjan's depth first search algorithm (DFS) [24]. The practical limitation of this algorithm is the amount of the randomly accessed memory which the algorithm requires. A space efficient alternative to Tarjan's algorithm (so called nested DFS) allowing to optimise the amount of randomly accessed memory exists (see i.e. [17]) and is implemented in SPIN verification tool [16]. However, even this optimisation does not solve the state space explosion problem sufficiently.

A very natural way how to overcome the memory limitation is to distribute the given graph onto several processors (computers) and to perform a distributed computation. As depth first search is P-complete, promising parallel DFS-based algorithms are unlikely to exist [21]. A completely different approach to distributed emptiness problem is needed. This paper demonstrates the methodology of reducing the automata based LTL model checking problem to the *negative cycle detection* problem. The problem is to find a negative length cycle in a directed graph whose edges have real number lengths.

The problem of negative cycles is closely related to the single-source shortest path (SSSP) problem. For this problem effective PRAM algorithms working with adjacency matrix representation of graphs are known, see i.e. [22]. However, the adjacency matrix representation is not compatible with other space-saving techniques like *on-the-fly* search. Other algorithms (for excellent survey see [8]), which are based on relaxation of graph's edges, are inherently sequential and their parallel versions are known only for special settings of the problem. For general digraphs with non-negative edge lengths parallel algorithms are presented in [19, 20, 12]. For special cases of graphs, like planar digraphs [25, 13], graphs with separator decomposition [10] or graphs with small tree-width [7] more efficient algorithms are known. Yet none of these algorithms is applicable on directed graphs with potential negative cycles.

We present a scalable distributed algorithm for the negative cycle problem and thus for automata based model checking of LTL formulas. Our method parallelises the model checking problem on a network of processors with disjoint memory that communicate via message passing.

The paper is organised as follows. We first review automata based LTL model checking and define the corresponding graph theoretic problem (Section 2). Its reduction to the negative cycle problem is outlined in Section 3. A distributed algorithm for the negative cycle problem is given in Section 4. Section 5 summarises the experimental results achieved.

2 Automata Based LTL Model Checking

Automata based approach to model checking of linear temporal logic formulas is a very elegant method developed by Vardi and Wolper [26]. The essence of using automata for model checking is that both the modelled system and the specification the system is supposed to fulfil are represented in the same way — as Büchi automata.

Definition 1. A Büchi automaton *is a tuple* $A = (\Sigma, S, s, \rho, F)$*, where*
- Σ *is a finite alphabet*
- S *is a finite set of states*
- $s \in S$ *is the initial state*
- $\rho : S \times \Sigma \to 2^S$ *is a transition relation*
- $F \subseteq S$ *is a set of accepting states*

A run *of* A *over an infinite word* $w = a_1 a_2 \ldots$ *is a sequence* $s_0, s_1, \ldots$ *such that for all* $i \geq 1 : s_i \in \rho(s_{i-1}, a_i)$*. A run* $s_0, s_1, \ldots$ *over* w *is* accepting *iff* $s_0 = s$ *and* $\{t \mid t = s_i \textit{ infinitely often}\} \cap F \neq \emptyset$*. A word* w *is* accepted *by* A *if there is an accepting run over* w*. The set of words accepted by* A *is denoted by* $L(A)$*.*

States of the modelled finite-state system M are identified with the states of a Büchi automaton A_M where all the states are accepting. Then, the set of behaviours of the system is the language $L(A_M)$. On the other hand, for each LTL formula φ one can construct a Büchi automaton A_φ that accepts exactly the set of runs satisfying the formula φ. Hence for the system M and LTL formula φ the verification problem is to verify whether $L(A_M) \subseteq L(A_\varphi)$ or equivalently whether $L(A_M) \cap L(A_{\neg\varphi})$ is empty. Moreover one can build an automaton A for $L(A_M) \cap L(A_{\neg\varphi})$ having $|\, M \,|\; 2^{\mathcal{O}(|\varphi|)}$ states. We need to check this automaton for emptiness [26].

Let $A = (\Sigma, S, s, \rho, F)$ be a given automaton. Consider the directed graph $G_A = (S, E_A)$ such that $E_A = \{(u, v) \mid v \in \rho(u, a), a \in \Sigma\}$. The following assertion can be easily verified [26].

Theorem 1. *Let* A *be a Büchi automaton. Then* $L(A)$ *is non-empty iff* G_A *has a cycle that is reachable from the initial state* s *and contains some accepting state.*

Detection of a reachable accepting cycle in a graph corresponding to a Büchi automaton is thus at the heart of most automata based model checkers. The depth first search strategy (DFS) provides a suitable time efficient approach. However, in large applications graphs are often too massive to fit completely inside the computer's internal memory. The resulting input/output paging between fast internal memory and slower external memory (such as disks) is then a major performance bottleneck.

In order to overcome problems with the limited size of randomly accessed memory we suggest to divide the graph onto several processors. The simplest solution is to run some DFS based algorithm on those processors. Instead of paging, computation is handed over to a processor owning related data i.e. paging is substituted by communication. As communication among processors is rather time consuming this approach could end up with algorithms which are comparatively slow (this finding is supported by experiments presented in Section 5).

Our methodology is based on the reduction of the Büchi automaton emptiness problem to a problem of detecting a negative cycle in an directed graph as is illustrated in the following section.

3 Negative Cycles

The negative cycle problem is a well-studied problem in connection with the single-source shortest path (SSSP) problem. We are given a triple (G, s, l), where $G = (V, E)$ is a directed graph with n vertices and m edges, $l : E \to R$ is a *length function* mapping edges to real-valued lengths, and $s \in V$ is the *source* vertex. The *length of path* $\rho =< v_0, v_1, \ldots, v_k >$ is the sum of the lengths of its constituent edges, $l(\rho) = \sum_{i=1}^{k} l(v_{i-1}, v_i)$. We define the *shortest path length* from s to v by $\delta(s, v) = \min\{l(\rho) \mid \rho \text{ is a path from } s \text{ to } v\}$ if there is such a path and $\delta(s, v) = \infty$ otherwise. A *shortest path* from vertex s to vertex v is then defined as any path ρ with length $l(\rho) = \delta(s, v)$. If the graph G contains no cycle c with negative length $l(c)$ (*negative cycle*) that is reachable from source vertex s, then for all $v \in V$ the shortest path length remains well-defined and the graph is called *feasible*. If there is a negative cycle reachable from s, shortest paths are not well-defined as no path from s to a vertex on the cycle can be a shortest path. If there is a negative cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

The SSSP problem is to decide whether, for a given triple (G, s, l), the graph G is feasible and if it is then to compute shortest paths from the source vertex s to all vertices $v \in V$. The negative cycle problem is to decide whether G is feasible.

The connection between the negative cycle problem and the Büchi automaton emptiness problem is the following. A Büchi automaton corresponds to a directed graph G_A as defined in Section 2. Let us assign lengths to its edges in such a way that all edges out-coming from vertices corresponding to accepting states have length -1 and all others have length 0. With this length assignment, negative cycles simply coincide with accepting cycles and the problem of Büchi automaton emptiness reduces to the negative cycle problem.

Theorem 2. *Let A be a Büchi automaton. Let $G^A = (G_A, s, l)$ where $l : E_A \to \{0, -1\}$ is the length function such that $l(u, v) = -1$ iff $u \in F$. Then $L(A)$ is non-empty iff G^A has a negative cycle reachable from s.*

4 Distributed Negative Cycle Detection Algorithm

The general sequential method for solving the SSSP problem is the *scanning* method [11, 8]. For every vertex v, the method maintains its distance label $d(v)$, parent vertex $p(v)$ and status $S(v) \in \{unreached, labelled, scanned\}$. The subgraph G_p of G induced by edges $(p(v), v)$ for all v such that $p(v) \neq nil$, is called the *parent graph*. Initially for every vertex v, $d(v) = \infty$, $p(v) = nil$ and $S(v) = unreached$. The method starts by setting $d(s) = 0$, $p(s) = nil$ and $S(s) = labelled$. At every step, the method selects a *labelled* vertex v and applies to it a scanning operation. During scanning a vertex v, every edge (v, u) outcoming from v is *relaxed* which means that if $d(u) > d(v) + l(v, u)$ then $d(u)$ is set to $d(v) + l(v, u)$ and $p(u)$ is set to v. The status of v is changed to *scanned* while the status of u is changed to *labelled*. If all vertices are either *scanned* or *unreached* then d gives the shortest path lengths and G_p is the graph of shortest paths.

Different strategies for selecting a *labelled* vertex to be scanned next lead to different algorithms. Our strategy comes out from the Bellman-Ford-Moore [8] algorithm which uses FIFO strategy to select a labelled vertex. The next vertex to be scanned is removed from the head of the queue; a vertex that becomes labelled is added to the tail of the queue if it is not already on the queue.

For graphs where negative cycles could exit the scanning method must be modified to recognise the unfeasibility of the graph. As in the case of scanning various strategies are used to detect negative cycles [8]. However, not all of them are suitable for our purposes – they are either uncompetitive (as for example time-out strategy) or they are not suitable for distribution (such as the admissible graph search which uses hardly parallelizable DFS or the level-based strategy which employs global data structures). For our distributed algorithm we have used the *walk to root* strategy.

The *walk to root* strategy is based on the fact that any cycle in G_p is a negative cycle. Suppose the relaxation operation applies to an edge (v, u) (i.e. $d(u) > d(v) + l(v, u)$) and the parent graph G_p is acyclic. This operation creates a cycle in G_p if and only if u is an ancestor of v in the current tree. This can be detected by following the parent pointers from v to s. If the vertex u lies on this path then there is a negative cycle; otherwise the relaxation operation does not create a cycle. However, the *walk to root* method increases the cost of applying the relaxation operation to an edge to $\mathcal{O}(n)$ since the cost of the search is $\mathcal{O}(n)$. Therefore the *walk to root* is performed only after the underlying relaxation algorithm performs $\Omega(n)$ work. The running time of *walk to root* is thus amortised over the relaxation time and overall time complexity is increased only by a constant factor. To preserve the termination of the strategy we will change and explain its behaviour afterwards.

The negative cycle detection algorithm *NC* we are proposing works in a distributed environment (no global information is directly accessible) where all processors communicate via message passing. We suppose that the set of vertices of the inspected graph is divided into disjoint subsets. The distribution is determined by the function *owner* which assigns every vertex v to a processor α.

For every vertex v processor $owner(v)$ knows its adjacency list. The distribution can be realized *on-the-fly*. Each processor α is responsible for its own part $G^\alpha = (V_\alpha, E_\alpha)$ of the graph G determined by the owned subset of vertices. Good partition of vertices among processors is important because it has direct impact on communication complexity and thus on run-time of the program. We do not discuss it here because it is itself quite a difficult problem and depends on the concrete application.

The main idea of the distributed algorithm *NC* can be summarised as follows. The distributed computation is initiated by the process *Manager* which performs the necessary initialisations. All processors participating in the algorithm execute the same program. Each processor performs repeatedly the basic scanning operation on all its vertices with *labelled* status (procedure *MAIN*). Such vertices are maintained in the processor's local *queue* Q^α. To process a vertex v which belongs to a different processor a message is sent to the owner of v. In each iteration it first processes messages received from other processors. Several types of messages could arrive:

- a request to update parameters of a vertex u. The procedure *UPDATE* compares the current value $d(u)$ with the received one. If needed, parameters are updated and the vertex u is placed into the *queue*.
- a request to continue in a walk, satisfied by executing the *WTR* procedure.
- a request to continue in removing marks, satisfied by executing the *REM* procedure.

Pseudo-Code of the Distributed Algorithm *NC*

```
proc MAIN() {running on each processor α}
  stamp := 0;
  if α = Manager then Q^α = {s}; d(s) := 0; p(s) := nil else Q^α := ∅ fi
  while not finished do process_messages; v := pop(Q^α); SCAN(v) od
end

proc SCAN(v)
  foreach (v, u) ∈ E do
    if owner(u) = α
      then UPDATE(u, v, d(v) + l(v, u))
       else send_message(owner(u), "start UPDATE(u, v, d(v) + l(v, u))") fi od
end

proc UPDATE(u, v, t)
  if d(u) > t then if walk(u) ≠ [nil, nil]
                   then if owner(v) = α
                          then push(Q^α, v)
                           else send_message(owner(v), "do push(Q, v)") fi
                    else d(u) := t; p(u) := v;
                         if WTR_amortization then WTR([u, stamp], u);
                                                  stamp + + fi;
                         if u ∉ Q^α then push(Q^α, u) fi fi fi
end
```

```
proc WTR([origin, stamp], at) {Walk To Root}
  done := false;
  while ¬done do
        if owner(at) = α
          then
               if walk(at) = [origin, stamp] →
                 send_message(Manager, "negative cycle found");
                 terminate
               □ (at = source) ∨ (walk(at) > [origin, stamp]) →
                 if origin ∈ V_α
                   then REM([origin, stamp], origin)
                    else send_message(owner(origin),
                                  "start REM([origin, stamp], origin))" fi
                 done := true;
               □ (walk(at) = [nil, nil]) ∨ (walk(at) < [origin, stamp]) →
                 walk(at) := [origin, stamp];
                 at := p(at)
               fi
           else
               send_message(owner(at), "start WTR([origin, stamp], at)");
               done := true
        fi
  od
end
```

```
proc REM([origin, stamp], at) {Remove Marks}
  done := false;
  while ¬done do
        if owner(at) = α
          then if walk(at) = [origin, stamp]
                 then walk(at) := [nil, nil];
                      at := p(at)
                  else
                      done := true fi
             else send_message(owner(at), start REM([origin, stamp], at));
               done := true fi
  od
end
```

The $SCAN$ procedure scans a vertex v. Every edge (v, u) outcoming from v is relaxed which means that if $d(u) > d(v) + l(v, u)$ then $d(u)$ is set to $d(v) + l(v, u)$ and $p(u)$ is set to v. If the vertex u lies on a walk to root path its parameters are not changed and the vertex v is placed back into the queue.

The WTR procedure is responsible for the negative cycle detection. The procedure follows the parent pointers starting from the state where the procedure has been invoked ($origin$). It is initiated after relaxation of an edge and according to a suitable amortisation strategy ($WTR\_amortisation$ condition becomes true every n-th time it is called). In the distributed environment it may be the case that even if the vertex v does not lie on any cycle, the parent graph can contain a cycle created in the meantime by some other processor. It can happen that WTR initiated from v reaches such a cycle and never finishes. The amortisation brings about this problem as well. To fix it each processor maintains a *counter* of started WTR procedures. WTR marks (variable *walk*) each vertex through which it proceeds by the name of the vertex where the walk has been initiated (*origin*) and the current value of the processor counter (*stamp*). A cycle is detected whenever a vertex with the actual *origin* and *stamp* is reached.

Moreover, it can happen that more than one WTR procedure is active at a time. In such a situation the concurrent walks could overwrite its own marks preventing thus detection of a cycle. It is sufficient to complete only one of them – if there is a cycle it will be detected. To decide which walk should continue let us suppose that a total linear ordering on vertices is given. A walk with lower *origin* is stopped.

There are four possible situations that can happen during the walk:

- the procedure reaches the source vertex s (line 9). A negative cycle has not been detected and the REM procedure is started.
- the procedure reaches a vertex marked with the same *origin* and the same *stamp* (line 6). This indicates that a negative cycle has been recognised. The cycle can be easily reconstructed by following parent edges. If necessary, the path connecting the cycle with the source vertex can be found using a suitable reachability algorithm.
- the procedure reaches a non-marked vertex, a vertex already marked with lower *origin* or a vertex marked with the same *origin* but lower *stamp* (line 15). The vertex is marked with $[origin, stamp]$ and the walk follows the parent edge.
- the procedure reaches a vertex already marked with higher *origin* (line 9). The walk is stopped and the REM procedure is started.

Whenever WTR has to continue in a non-local vertex a request to the vertex owner is sent and the local walk is finished.

The purpose of the REM procedure is to remove marks introduced by the WTR procedure. These marks could otherwise obstruct some possible future runs of WTR through marked vertices. Marks to be removed are found with the help of parent edges (this is why the updating of a marked vertex is postponed (line 2 of $UPDATE$)). The REM procedure follows the path in the parent graph starting from the *origin* in a similar way as WTR does. It finishes when it reaches a source vertex or a vertex marked with different *origin*. However, this does not guarantee that all marks are removed at that very moment. Note that these marks will be removed by some other REM procedure eventually. The correctness of cycle detection is guaranteed as for the cycle detection the equality of both *origin* and *stamp* is required.

The distributed algorithm terminates when either all queues of all processors are empty and there are no pending messages or when a negative cycle has been detected. The *Manager* process is used to detect termination and to finish the algorithm by sending a *termination* signal to all the processors.

Theorem 3 (Correctness and Complexity).
If G has no negative cycle reachable from the source s, then the algorithm terminates, $d(v) = \delta(s, v)$ for all vertices $v \in V$, and the parent graph G_p is a shortest path tree rooted at s. Otherwise the existence of a negative cycle is reported.
If G is distributed over P processors each of which owns $\mathcal{O}(n/P)$ vertices, then the worst case computation complexity is $\mathcal{O}(n^3/P)$.

For detailed proof of the correctness and the complexity analysis see [6].

5 Experiments

We have implemented the algorithm proposed in Section 4. The implementation has been done in C++ and the experiments have been performed on a cluster of eight 366 MHz Pentium PC Linux workstations with 128 Mbytes of RAM each interconnected with a fast 100Mbps Ethernet and using Message Passing Interface (MPI) library.

In the implementation of the *NC* algorithm we have employed the following optimisation scheme. For more efficient communication between processors we do not send separate messages. The messages are sent 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 about 100 single messages.

As far as we know there is no other distributed algorithm for negative cycle problem (see Section 1). Therefore our objective was to compare the performance of the *NC* algorithm with algorithms used in LTL model checkers. For comparison we have used very effective nested depth first search (*NDFS*) algorithm [17] used in SPIN verification tool [16]. In its distributed version the graph is divided over processors like in the *NC* algorithm. Only one processor, namely the one owning the actual vertex in the *NDFS* search, is executing the nested search at a time. The network is in fact running the sequential algorithm with extended memory. The worst case space complexity of *NDFS* is asymptotically the same as the one of our algorithm *NC*. The worst case time complexity of *NDFS* is linear in the number of vertices and edges.

We performed several sets of tests on different instances in order to verify how fast is the algorithm in practice, i.e. beyond its theoretical characterisation. Our experiments were performed on two kinds of systems given by random graphs and generated graphs. Graphs were generated using a simple specification language and an LTL formula. In both cases we tested graphs with and without cycles to model faulty and correct behaviour of systems. As our real example we tested the parametrised Dining Philosophers problem. Each instance is characterised by the number of vertices and the number of cross-edges. The number of cross-edges significantly influences the overall performance of distributed algorithms.

For each experiment we report the average time in minutes and the number of sent messages (communication) as the main metrics. Table 1 summarises the achieved results.

The experiments lead basically to the following conclusions:

- *NC* algorithm is comparable with the *NDFS* one on all graphs.
- *NC* algorithm is significantly better on graphs without negative cycles.

		NDFS		NC	
Vertices	Cross–edges	Time	Messages	Time	Messages
Generated, without cycle					
40398	34854	1:01	79376	0:11	809
71040	1301094	31:13	3008902	0:48	1108
696932	1044739	27:02	2387220	1:31	14029
736400	5331790	126:46	12316618	5:17	48577
777488	870204	21:36	1887252	2:02	13872
1859160	1879786	49:04	4226714	6:00	25396
Generated, with cycle					
18699	22449	0:06	22	0:05	68
33400	2073288	0:37	30824	0:24	555
46956	83110	0:05	108	0:09	702
448875	1863905	0:51	21106	0:56	3435
Random, without cycle					
4000	353247	14:03	1390262	0:17	17868
5000	839679	31:48	3151724	0:32	2489
80000	522327	30:11	2042212	1:39	87002
60000	1111411	57:19	4131210	4:08	98686
947200	5781959	184:23	13338462	9:49	47030
Random, with cycle					
18000	1169438	1:20	104822	0:09	862
Philosophers					
(12) 94578	42154	2:06	168616	0:13	756
(14) 608185	269923	16:11	1079692	1:40	4500

Table 1. Summary of experimental results

Experiments show that in spite of worse theoretical worst time complexity of *NC* algorithm its behaviour in practice can outperform the theoretically better *NDFS* one. This is due to the number of communications which has essential impact on the resulting time. In *NC* algorithm the messages can be grouped into packets and sent together. It is a general experience that the time needed for delivering t single messages is much higher than the time needed for delivering those messages grouped into one packet. On the other hand, *NDFS* algorithm does not admit such a grouping. Another disadvantage of *NDFS* is that during the passing of messages all the processors are idle, while in *NC* algorithm the computation can continue immediately after sending a message. Last but not least, in *NDFS* all but one processor are idle whereas in *NC* all can compute concurrently. We notice that all mentioned advantages of *NC* algorithm demonstrate themselves especially for systems without cycles where the whole graph has to be searched. This is in fact the desired property of our algorithm as the state explosion demonstrates itself just in these cases. Both algorithms perform equally well on graphs with cycles.

We have accomplished yet another set of tests in order to validate the scalability of the *NC* algorithm. The tests confirm that it scales well, i.e. the overall time needed for treating a graph is decreasing as the number of involved processors is increased.

6 Conclusions

Parallel and distributed algorithms for reachability analysis and model checking have recently been investigated as a possible method to handle large state spaces. The core problem of automata based model checking is the detection of reachable accepting cycles in the state space. The classical depth first strategy provides a suitable approach to cycle detection in a sequential case. However, the depth first search approach is difficult to distribute.

The paper proposes a novel approach to the cycle detection problem in a distributed environment. The main idea is to transform the accepting cycle detection problem to the single-source shortest path problem in graphs with real number edge lengths – negative cycle problem. We have proposed a scalable distributed algorithm to solve this problem and we have performed a series of experiments to evaluate its performance.

The performance of the algorithm was compared with a distributed DFS based algorithm. The experimental results show that the distributed algorithm based on negative cycle detection significantly outperforms the DFS based one due to higher degree of asynchronous parallelism which allows to optimise necessary communication. DFS based algorithms rely on strict synchronisation.

In the future we aim to embed the algorithm in a suitable automata based verification tool (e.g. SPIN) to be able to test its applicability to a non-trivial series of real systems. Furthermore, we intend to explore various heuristics and implementation techniques to optimise its performance.

References

1. S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Application*, volume 103 of *LNCS*, pages 40–56. Springer, 1987.
2. J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *Proc. SPIN 2001*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.
3. G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.
4. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proc. FMCAD 2000*, 2000.
5. B. Bollig, M. Leucker, and M Weber. Parallel model checking for the alternation free mu-calculus. In *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

6. L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed shortest path for directed graphs with negative edge lengths. Technical Report FIMU-RS-2001-01, Faculty of Informatics, Masaryk University Brno, http://www.fi.muni.cz/informatics/reports/, 2001.
7. S. Chaudhuri and C. D. Zaroliagis. Shortest path queries in digraphs of small treewidth. In *Proc. ESA 1995*, volume 979 of *LNCS*, pages 31–45. Springer, 1995.
8. B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, (85):277–311, 1999.
9. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.
10. E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.
11. T. H. Cormen, Ch. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT, 1990.
12. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proc. MFCS 1998*, volume 1450 of *LNCS*, pages 722–731. Springer, 1998.
13. P. Spirakis D. Kavvadias, G. Pantziou and C. Zaroliagis. Efficient sequential and parallel algorithms for the negative cycle problem. In *Proc. ISAAC 1994*, volume 834 of *LNCS*, pages 270–278. Springer, 1994.
14. O. Grumberg, T. Heyman, and A. Schuster. Distributed model checking for mu-calculus. In *Proc. 13th Conference on Computer-Aided Verification CAV01*, LNCS. Springer, 2001.
15. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 20–35. Springer, 2000.
16. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
17. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.
18. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. SPIN 1999*, number 1680 in LNCS. Springer, 1999.
19. U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proc. EUROPAR 2000.* LNCS, 2000.
20. K. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.
21. J.H. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
22. S.H. Roosta. *Parallel processing and parallel algorithms.* Springer, 2000.
23. U. Stern and D.L. Dill. Parallelizing the Murφ verifier. In *Proc. CAV 1997*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.
24. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.
25. J. Traff and C.D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Proc. IRREGULAR-3 1996*, volume 1117 of *LNCS*, pages 183–194S. Springer, 1996.
26. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS 1986*, pages 332–344. Computer Society Press, 1986.