Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 6 pages

Distribution of Explicit-State LTL Model-Checking

Luboš Brim [1,2], Jiří Barnat [3]

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

We give a brief summary of recent achievements in the *Parallel and Distributed Systems Laboratory* at the Faculty of Informatics in Brno that are related to the distribution of explicit-state LTL model-checking. Distribution and parallelization of verification algorithms is one of the current key research themes in the laboratory.

Model-checking of complex industrial systems requires techniques to avoid the state-explosion problem. For large models, the state space does not completely fit into the main memory of a computer and the model-checking algorithm becomes very slow as soon as the memory is exhausted and system starts swapping. A possible approach to dealing with these practical limitations is to increase the computational power (especially random-access memory) by building a powerful parallel computer as a network (cluster) of workstations.

In this paper we survey our own work on distributing of explicit-state (enumerative) model-checking algorithms for linear temporal logic (LTL).

The other research performed in the laboratory aims at the development of new original methods and techniques for the automated verification of large-scale industrial critical systems, with emphasis on practical aspects of their application, at applying these and other already known methods and techniques to real-life systems, optimizing these techniques to make them sufficiently efficient, and providing software support to use them.

1 Sequential LTL Model Checking

Automata based approach to model-checking of linear temporal logic is a very elegant method developed by Vardi and Wolper [8]. The idea is in reducing the model-checking problem to the non-emptiness problem for Büchi automata. A Büchi automaton accepts a word iff there exists a reachable *accepting cycle*,

[1] Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509.
[2] Email: brim@fi.muni.cz
[3] Email: barnat@fi.muni.cz

i.e. iff there is an *accepting state* reachable from the initial state and from itself. Courcoubetis et al. [6] have proposed to use a *nested depth first search – Nested DFS* to detect reachable accepting cycles. The first search (*primary*) is used to search for reachable accepting states while the second one (*nested*) tries to detect accepting cycles.

Before turning the attention to distributed solutions, it is worth to recall some facts on the theoretical complexity of the LTL model-checking problem. The LTL model-checking problem is PSPACE-complete with time complexity $\mathcal{O}((m+n).2^{|\varphi|})$. Performance of parallel algorithms is measured by two functions of input of size: the time complexity and the processor complexity. A parallel algorithm is said to be efficient if its time complexity is polylogarithmic with a polynomial number of processors. The class of problems efficiently solvable by parallel algorithms is denoted by *NC*.

According to the standard opinion in the complexity theory problems that are P-complete (hence not in *NC*) most likely do not admit efficient parallel algorithms. Such problems are considered to be inherently sequential. This is also the case of LTL model-checking. On the other hand, general complexity results are not the most relevant for assessing the cost of model-checking in practical situations. Better characteristics can be obtained by considering the two parameters (the size $|M|$ of the system and the size $|\varphi|$ of the formula) separately. In practice the size of the systems is usually quite large while the size of the formula is small. The *program complexity* of model-checking is its computational complexity measured as a function of $|M|$ only with the temporal formula being fixed. The program complexity of LTL model checking is in *NC*.

2 Distributed LTL Model-Checking

Our aim is to solve the LTL model-checking problem by distribution, i.e. by utilizing several interconnected workstations. The sequential approach 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 order is not generally maintained any more due to different speeds of involved workstations. This fact makes the distribution of LTL model-checking a challenging task. We will sketch three possible approaches to dealing with the problem. The first idea is to use additional data structures to maintain the global DFS order on at least the most critical vertices. The second one is to use a different search procedure that is not sensitive to the violation of the DFS order in the distributed environment and/or to reduce the problem to another one with better parallelization potential. The third idea is to distribute the state space in such a way that cycles are not splitted among the workstations, hence the "global" DFS order does not matter. We will now briefly introduce three techniques to the distribution of explicit-state LTL model-checking that demonstrate application of these ideas. We

will try to explain the main concepts only, for the technical details we give a reference. All the algorithms have been implemented and experimentally evaluated. Performance results we obtained show significant improvements with respect to sequential techniques, both in extension of the size of the problem and in computational times, along with adequate scalability with the number of processors.

2.1 Using additional data structures to maintain the DFS order

A straightforward approach to the distribution of the *Nested DFS* algorithm is to allow simultaneous (parallel) execution of the algorithm on each workstation (with a randomly partitioned state-space). However, such an approach could lead to an incorrect result because the DFS postorder is not preserved. The only situation in which the order does not matter is the verification of safety properties (the problem can be reduced to the *reachability* problem). This is the case of the primary DFS which searches for accepting states. On the other hand, the nested DFS must be started from the accepting states in the postorder defined by the primary DFS, otherwise an existing cycle could be missed. The order of accepting states is important. A special data structure (*dependency structure*) is used to maintain the proper order of accepting states. Nested DFS procedures for accepting states are then initialized separately in the correct order which is determined by the dependency structure. Only one nested DFS procedure is started at a time. The algorithm thus performs a "limited" nested depth-first search which requires some synchronization during the execution.

The dependency structure is built in such a way that a nested DFS can start from an accepting state only if all the accepting states "below" (in the sense of the global postorder) have already finished their respective nested DFS procedures. Each workstation maintains its own local dependency structure. The structure is dynamic, vertices are added and removed. The vertices are border states and accepting states and the edges represent reachability among these states as discovered by the primary search. The primary DFS creates vertices on the way down. A state received from another workstation becomes a new vertex (if not already in the structure) augmented with the identification of the sending workstation. A vertex is deleted when the primary DFS backtracks through the state and all of its successors have been deleted. When an accepting state is deleted it is sent to the global *Seed queue* kept at the manager process. The manager process dequeues a state and initializes the nested DFS procedure for this state as soon as the previous one is finished.

The distributed algorithm requires additional memory. The number of states stored in the dynamic structure is $O(n.r)$ on average, where r is the maximal out-degree and n is the number of states. In most real systems the amount of non-determinism (represented by r) is limited and small. Another

drawback of the algorithm is that nested DFS procedures are not performed in parallel. However, under certain circumstances, which can be effectively recognized from the dependency structure, it is possible to start more than one nested DFS procedure at a time. For more details we refer to [1].

2.2 Negative cycles

We reduce the problem of detecting accepting cycles to a problem of detecting negative length cycles. The connection between the negative cycle problem and the Büchi automaton emptiness problem is the following. A Büchi automaton corresponds to a directed graph. 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.

The negative (length) cycle problem is closely related to the single source shortest path problem (SSSP). The general sequential method for solving the SSSP problem is the *scanning* method. For every vertex v, the method maintains its distance label $d(v)$ and its parent vertex $p(v)$. The label-correcting variant of the algorithm scans vertices and updates (corrects) the distance labels and pointers to parent vertices. These updates are "local" in the sense that they depend on neighbors only, hence can be computed in *any* order, therefore in parallel. The algorithm runs in $\mathcal{O}(mn)$ time in the worst case, where m is the number of edges.

For graphs where negative cycles could exist the scanning method must be modified in such a way that negative cycles are detected. Various strategies are used. For our distributed algorithm we have chosen the *walk to root* cycle detection strategy. In the walk to root strategy the graph built from parent vertices is tested for acyclicity. This test is done by starting a "walk" in the parent graph from a vertex being updated "back" to the root. Using additional data structure, several walks can be performed in parallel.

The algorithm is work-efficient as its worst time complexity is $\mathcal{O}(\frac{m.n}{p})$, where p is the number of processors. Compared to the sequential complexity of the nested DFS algorithm we pay by increase in time - from $\mathcal{O}(m+n)$ to $\mathcal{O}(mn)$. Also a limited amount of additional memory is required. On the other hand, we gain a reasonable parallelization of the model-checking problem. The method has been published in [4].

2.3 Property based distribution

The difficulty in cycle detection in distributed environment is caused by dividing them among more workstations. A suitable partition function can significantly improve the performance of cycle detection algorithms. Some of the state space generation and partition techniques exploit certain characteristics of the system, and hence work well for systems possessing these characteris-

tics, but fail to work well for systems which do not have them. In some cases it is possible to decide in advance whether the system under consideration has the required characteristic. However, in most situations this is not the case.

In [2] we proposed a technique that uses the verified property to partition the state space in a distributed on-the-fly automata-based LTL model checking in order to eliminate division of accepting cycles.

An obvious approach is to decompose the graph into maximal SCCs first and then to partition the graph according to this decomposition. However, decomposing the system in advance into SCCs would actually solve the verification problem.

Our aim is to partition the graph in such a way that no accepting cycle is divided among more workstations. In addition, such a decomposition allows to limit the nested (second) DFS search to those paths that can really form a cycle in the graph only, i.e. the paths that belong to one SCC. Therefore, there is no state that could be visited by two different nested DFS procedures originating from two different workstations.

In automata-based LTL model-checking the verification problem is represented as the emptiness problem of a Büchi automaton which is obtained as a *synchronous product* of two automata. Thus each state has two parts: the one given by the modeled system and the other one given by the negative claim automaton (representing negation of the verified formula). We use the decomposition of the negative claim automaton into maximal SCCs as a heuristic to partition the state space. The main idea is that the partition function checks which SCC the formula part of a state in the product automaton belongs to and places the state on the same workstation as all the other states whose formula part is in the same component in the decomposition of the negative claim automaton. The partition function is static and can be pre-computed efficiently in advance. As the nested search "remains" on one workstation, the level of asynchronous behavior of the algorithm can be increased by allowing execution of other nested DFS procedures on different workstations simultaneously.

The idea can be further refined in the following way. There are three *types* of SCCs in the negative claim automaton [7]: Type F – any cycle within the component contains at least one accepting state, type P – there is at least one accepting cycle and one non-accepting cycle within the component, and type N – there is no accepting cycle within the component.

We can distribute states belonging to a component of type N arbitrarily among the workstations as the only relevance of these states is in their reachability. For components of type F the cycles can be detected sequentially without using the nested search and we place each component on a separate workstation. Type P components can be either placed on a single workstation or distributed and checked for cycles by one of the previously mentioned distributed algorithms.

Since the type P components are quite rare in real applications, the only

real challenge is to find an effective specialized distributed algorithm for type F components.

2.4 DiVinE - *Distributed Verification Environment*

Development of a tool that would support the distributed verification of systems is one of our recent projects. The goal is to build an environment for easy implementation of our own distributed verification algorithms on clusters of workstations, for their experimental evaluation and comparison. The main characteristics are in support for the distributed generation of the state space, dynamic load balancing, distributed generation of counter-examples, fault-tolerance, re-partitioning. All our algorithms will be implemented within the tool. The distributed environment quite naturally allows for methods and algorithms integration and cooperation. This is another goal of the DiVinE project.

References

[1] Barnat, J., L. Brim and J. Stříbrná, *Distributed LTL Model-Checking in SPIN*, in: *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, LNCS **2057**, 2001, pp. 217–234.

[2] Barnat, J., L. Brim and I. Černá, *Property Driven Distribution of Nested DFS*, in: *VCL 2002, Technical Report DSSE-TR-2002-5 in DSSE*, 2002, pp. 1–10.

[3] Brim, L., J. Crhová and K. Yorav, *Using Assumptions to Distribute CTL Model Checking*, in: *PDMC'02*, ENTCS **68.4**, 2002.

[4] Brim, L., I. Černá, P. Krčál and R. Pelánek, *Distributed ltl model checking based on negative cycle detection*, in: *FSTTCS 2001*, number 2245 in LNCS, 2001, pp. 96–107.

[5] Brim, L., I. Černá, P. Krčál and R. Pelánek, *How to employ reverse search in distributed single-source shortest paths*, in: *SOFSEM'01*, number 2234 in LNCS, 2001, pp. 191–200.

[6] Courcoubetis, C., M. Vardi, P. Wolper and M. Yannakakis, *Memory-Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design **1** (1992), pp. 275–288.

[7] Edelkamp, S., A. Lluch-Lafuente and S. Leue, *Directed model-checking in HSF-SPIN*, in: *8th International SPIN Workshop*, number 2057 in LNCS, 2001, pp. 57–79.

[8] Vardi, M. Y. and P. Wolper, *An automata-theoretic approach to automatic program verification*, in: *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86*, IEEE Computer Society Press, Washington, DC, 1986 pp. 332–344.