Distribution of Explicit-State LTL Model-Checking*

Lubos Brim and Jiří Barnat

Faculty of Informatics, Masaryk University Brno,
Botanická 68a, Brno, Czech Republic
{brim,barnat}@fi.muni.cz

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

Model checking of complex industrial systems requires techniques to avoid the state-explosion problem. Several methods to overcome this barrier have been proposed and successfully implemented in automatic verification tools. The use of distributed processing to combat the state explosion problem gained interest in recent years. For large models, the state space does not completely fit into the main memory of a computer and hence model checking algorithm becomes very slow as soon as the memory is exhausted and system starts swapping. A 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 survey our own work on distributing of explicit-state (enumerative) model checking algorithms for linear temporal logic (LTL). LTL is a major logic used in formal verification, LTL model checking has very efficient sequential solutions based on automata.

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 [1]. 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 enables the reduction of model-checking problem to the non-emptiness problem for Büchi automaton. A Büchi automaton accepts a word iff there exists an accepting state reachable from the initial state and from itself. The direct approach to solving the non-emptiness problem is to decompose the underlying graph into strongly connected components (SCCs) which can be done in time linear in the size of the graph using the Tarjan’s algorithm [2]. However, constructing SCCs is not memory efficient since the states in the components must be stored explicitly during the procedure. Courcoubetis et al. [3] have proposed an elegant way to avoid the explicit computation of SCCs. The idea is to use a nested depth first search – Nested DFS to find accepting states that are reachable from themselves (to compute accepting path). 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 LTL model checking problem. LTL model checking problem has two inputs: a model $M$ and a temporal formula $\varphi$. The complexity of model checking algorithms is given in terms of the sizes $|M|$ and $|\varphi|$. LTL model checking problem is PSPACE-complete with time complexity $O(|M| \cdot 2^{2|\varphi|})$. Performance of parallel algorithms is measured by two functions of

* This work has been supported in part by the Grant Agency of Czech Republic grant No. 201/03/0509.
input of size $n$: the time complexity $T(n)$ and the processor complexity $P(n)$. A parallel algorithm is said to be efficient if its time complexity is poly-logarithmic with a polynomial number of processors. The class of problems efficiently solvable by parallel algorithms is denoted $NC$.

According to the standard opinion in complexity theory problems that are P-complete 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 LTL model checking problem by distribution, i.e. by utilizing several interconnected workstations. The standard sequential solution 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 split among the workstations, hence the “global” DFS order does not matter. We will now briefly introduce three approaches 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.

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 can be 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 post-order) 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 [?].

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 \( O(mn) \) 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 \( O(n^2/P) \), where \( P \) is the number of processors. Compared to the sequential complexity of nested DFS algorithm we pay by increase in time - from \( O(m + n) \) to \( O(mn) \). 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 [?], implemented and experimentally evaluated with promising results.

2.3 Property Based Distribution

The difficulty in detecting cycles 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 characteristics, 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 [?], 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 reduce the number of divided cycles.

An obvious approach is to decompose the graph into maximal strongly connected components first and then to partition the graph according to this decomposition. However, decomposing
the system in advance into strongly connected components 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 strongly connected component. 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 strongly connected components as a heuristic to partition the state space. The main idea is that the partition function checks which strongly connected component 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 strongly connected components in the negative claim automaton [2]: 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 and intentionally 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.

3 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, repartitioning. 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.

4 Other work related to formal methods

The 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.

In addition to the above mentioned work on LTL, the research in the laboratory is focused on the distribution of other verification problems. In particular, results related to the distribution of model checking for several branching time logics (CTL, RegCTL, AFMC) have been investigated,
we continue our work on the distribution of equivalence checking. Also, theoretical work which underlies formal techniques (not necessarily connected to distribution and parallelization) is performed. Results on expressibility and classification of various temporal logics, on decidability and complexity of several verification problems including infinite state systems have been achieved.

As a side-effect of collecting information on verification tools, which now evolved in its own life, we built a repository of tools called YAHODA (http://anna.fi.muni.cz/yahoda/). It was developed to provide well-organized and up-to-date overview of formal verification tools. It is especially aimed at collecting information on tools for model checking and equivalence checking.