Distributed Modular CTL Model Checking*

Jitka Crhová

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

E-mail: `xcrhova@fi.muni.cz`

Abstract

A distributed model checking algorithm is presented. It is designed to be run on a network of workstations that communicate via message passing. The algorithm handles large state spaces by partitioning them into smaller units. Such a partition of state space into partial state spaces can be employed to perform several smaller verification problems independently on several computers increasing thus not only speeding up the verification, but most importantly the available memory.

We also suggest partition function that can be used as input to the distributed algorithm. For software systems a quite successful approach is to partition them following the syntactical structure of the program. Other state based temporal logics and various model checking algorithms can be adapted easily as well.

1 The research question

Modern computer systems are highly complex, but yet they must be extremely reliable and correct. In recent years, *formal methods* have received much attention. They can be viewed as a collection of mathematical methods that allow to *verify* a system, i.e. to ensure that the system meets its specification. In contrast to testing and simulation, verification considers all possible behaviors of the system.

Model checking techniques [7] are formal methods that deal with a finite-state model of a system and its specification expressed in some type of temporal logic [13]. These techniques have been successfully applied to hardware system in commercial practice, and they proved even to be efficient in validating crucial properties of concurrent software systems as well.

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

The most challenging technical limitation to applying model checking in practice is the so called *state explosion problem*. Basically, there are two reasons for that. The first one comes from the need of dealing with all possible behaviors of the environment of the system. Thus, we have to consider all possible inputs of the system. The second reason is dealing with concurrency. Concurrent processes are viewed as interleaved traces in which actions of each task can appear in an arbitrary order. Hence, we have to consider all those orderings. The state explosion problem is even more severe for software systems where a state space of a single process itself can be extremely large. So, the question is how to fight the state explosion problem in model checking.

2 Current solutions

Many approaches have been developed to combat the state explosion problem, such as partial order reduction, symbolic algorithms or probabilistic techniques. These methods can extend possibilities of model checking significantly, but still they are insufficient in many cases. Hence a lot of attention is paid to find ways to reduce memory requirements of model checking algorithms.

One of the approaches that tries to solve the state explosion problem is to distribute the state space among a number of computers so as to be able to verify large systems.

The important feature of algorithms running in a distributed environment is to solve the given task by distributing the data among the participating workstations with as small amount of coordination as possible. One of the main issues in distributed model checking algorithms is how to partition the state space (data) among the individual computers called *network nodes*. There are two aspects that significantly influence the overall effectiveness of model-checking in the distributed environment: *locality* and (spatial) *balance* of the state space partition. Locality means that most of the state's descendants are assigned to the same node as the parent state, reducing thus communication and cooperation overhead. Balance means that each network node is assigned approximately the same number of states, achieving thus good speed-up.

The main idea of the majority of known approaches to distribute the state space is similar: the state graph is partitioned among the network nodes, i.e., each network node owns a subset of the state space. The differences are in the way the state space is partitioned (*partition function*). Random techniques to partition the state space have been used [10, 12, 1]. A technique which exploits some structural properties derived from the verified formula has been proposed in [2].

The model checking algorithm running on each network node has thus access only to a part of the entire system. It communicates with other nodes to achieve the required (global) result. The way of communication depends on a distributed algorithm.

Another approach that deals with the state explosion problem is *modularity*, i.e. exploiting the structure of the system. Traditional model checking methodologies take a given program written in some kind of higher-level specification language and then flattens it into a huge, but finite Kripke structure. That means that information concerning the structure of the system is lost. Exploiting this information can strengthen possibilities of model checking heavily.

The notion of a *module* is crucial in modular model checking. There are two main conceptions of module.

In the first case, a system is a parallel composition of its modules. That is, each module contributes to the whole system in a multiplicative way. If we have two modules M_1 and M_2, then the size of the system $M_1||M_2$ is $|M_1| \cdot |M_2|$. Thus, the number of all states in all modules is typically much smaller than the size of the whole system.

In this context, it is useful to distinguish between *closed* and *open* systems. A *closed system* is a system whose behavior is completely determined by the state of the system. An *open system* is a system that interacts with its *environment* and whose behavior depends on this interaction. Typically, model checking is performed on a closed system modeled as a parallel composition of two open systems - the system under consideration with behavior following exactly its definition, and an *environment*, which can behave in a completely arbitrary way. Thus, when we want to examine a module in isolation, we view this module as a system and the other modules as its environment.

Of course, a system will behave properly only if its environment does. The specification of correctness of the system (a temporal logic formula in model checking) should consist of two parts. One part describes the guaranteed behavior of the module. The other part describes the assumed behavior of the environment with which the module is interacting. This is called the *assume-guarantee* paradigm (see e.g. [8, 14]).

Suppose we want to verify that a system $M_1||M_2$ satisfies a formula φ. Using an assumption-guarantee paradigm, we can show that the module M_1 satisfies some formula ψ and then, assuming that the environment of M_2 satisfies ψ, show that M_2 satisfies φ. The assumption-guarantee paradigm is a very powerful technique. However, it is not fully automatic as the user should specify the assumptions (i.e. the formula ψ).

Another technique is that of *interface processes* [3]. It allows to reduce one of the two modules in composition, say M_1 to a process that behaves in such a way that the other module M_2 cannot distinguish between the full and the reduced version of M_1.

The second approach to modularity defines a module as a set of states of the whole state space, possibly with additional restrictions on the structure of the module. In other words, each module contributes to the whole system in an additive way, and the number of states of all modules is approximately the same as the number of states of the system created by the modules. Intuitively, the system is a union of its modules. In [9], a module is defined via syntactic structure of the program, that means, in a module there are states that represent a syntactic block of code. On the other hand, in [11] a definition of a module follows the semantics of the system.

As well as in the assumption-guarantee approach, the notion of *assumptions* can be employed. Again, assumptions reflect the behavior of the rest of the system. Most often an *assumption function* that specifies for a state and a formula whether the state is assumed to satisfy the formula is used. Frequently it is said that it is assumed that a formula is true in a state meaning that the previous computation ensures that the formula holds in the state. A notion of "don't know" value ($\perp$) is employed to say that no assumption can be made. The assumptions are not needed for all states, but only for those that represents an "interface" between modules. There can be found many types of assumption functions in literature depending on the problem under consideration, see e.g. [9, 6, 5].

3 Preliminary ideas and the proposed approach

The aim of this work is to exploit modular techniques to distribute model checking algorithms. This can be useful especially for software, as the software model checking algorithms suffer from state explosion even more severely that the hardware model checking techniques. In contrast to hardware, where the systems are usually composed of many components working in parallel, a software system can be extremely large even when it consist of one sequential finite-state component.

The 'additive' approach to modularity is more suitable to solve this problem, as a module in this approach corresponds to partitioning of the state space used by distributed algorithms.

The goal of this research is to give a theoretical background for distributed model checking based on a modular approach. In addition, we want to exploit modular approaches to partitioning the state space, in particular to define partition functions that reduce the necessary communication in distributed environment.

4 The contributions and results of the work

Recently, in [4] we have proposed a technique that extends the approach of Yorav (Laster) and Grumberg [9, 15] to partitions not necessarily resulting from the syntactical structure of the program, allowing thus a distributed model checking. We have introduced a distributed CTL model checking algorithm.

For simplicity reasons we suppose the system under consideration to be modeled as a *total* Kripke structure $M = (S, R)$ (i.e., each state has at least one successor) with a unique initial state $\hat{s}$.

The main ideas of the distributed algorithm are following (for more details see [4]. The Kripke structure on each network node should form a *fragment* of the whole Kripke structure. A fragment of M is a substructure of M satisfying the property that every state in the fragment has either no successor or it has exactly the same successors as in M.

Definition: A Kripke structure $M_1 = (S_1, R_1)$ is a *fragment* of a Kripke structure $M = (S, R)$ iff

1. $S_1 \subseteq S$,
2. $R_1 \subseteq R$
3. $\forall(s_1, s_2) \in R$: if $s_1 \in S_1$, then either $(s_1, s_2) \in R_1$ or $\nexists t \in S_1 : (s, t) \in R_1$.

A partition function considered to be an input of the algorithm assigns each network node n a set of states S_n. The Kripke structure managed by the node n is a fragment of the whole structure $Fragment_M(S_n)$ defined as follows:

Definition: Let $M = (S, R)$ be a Kripke structure, $T \subseteq S$. We define the Kripke structure $Fragment_M(T) = (S_T, R_T, I_T)$ as follows:

1. $S_T = \{s \in S \mid s \in T \vee \exists s' \in T \text{ s.t. } (s', s) \in R\}$
2. $R_T = \{(s_1, s_2) \in R \mid s_1 \in T, s_2 \in S_T\}$

Thus, a node n contain states from S_n (called *original states*) and their successors (called *subsequent states*). As the whole structure M is total, subsequent states are exactly those with no successor in the fragment (i.e. *border states* of the fragment). In fact, they belong to some other network node.

We have considered a state based branching time temporal logic CTL (for definition see e.g. [9]). To define the semantics of CTL formulas over Kripke structures with border states we need to adapt the standard semantic definition. CTL is usually interpreted over total structures, while fragments are typically non-total. Furthermore, we need to define the notion of the truth under assumptions associated with border states. An *assumption function* is a partial function that takes a state and a formula and returns **true**, **false** or undefined value $\perp$.

The CTL semantics under assumptions is then defined as a function $\mathcal{C}_M$ that takes an assumption function, i.e. initial assumptions, and returns an assumption function. Intuitively, the returned function contains all information that can be derived from initial assumptions.

The distributed algorithm uses a procedure for performing model checking on each fragment (so called *node algorithm*). Modification of various model checking algorithms can be considered, particularly of an explicit state or of symbolic algorithms (see [7]). Each fragment is managed by a separate process. These processes are running concurrently on each network node.

Each process initializes its assumption function to be undefined on all inputs. After initialization it computes (using the node algorithm) all information that can be gained from current assumptions. Then it sends the results to each process P that may be interested in them (i.e., it sends the part of the assumption function for P's border states) and receives similar information from the other processes. These steps are repeated until a fixpoint is reached ("global" stabilization occurs), i.e. until no new information can be computed.

After stabilization there still may remain a state s and a formula φ for which the validity is undefined. This can happen in the case of the *until* operator. However, if the truth of all subformulas of φ has already been computed in all states in all nodes, then from the fact that the fixpoint has been reached we can conclude that φ does not hold in s. Therefore all processes extrapolate this information and continue to compute. This repeats until the desired result is obtained. The main idea of the algorithm is expressed in Figure 1.

The partition function f that is an input of the algorithm influences the performance of the algorithm heavily. We suggest several ways how to define the partition function. We exploit syntactic information of software programs.

```
proc Distributed Algorithm
  (inputs: total KS M,
        CTL formula ψ,
        partition function f)
  Split M into fragments K_i;
  forall i ∈ {1, ..., n} do in parallel {for all K_i}
    Take the initial assumption function;
    repeat
        repeat
            Compute all you can;
            Send relevant information to other nodes;
            Receive information from other nodes;
        until all processes reach fixpoint;
        Extrapolate additional information;
    until all is computed;
    Return result for the initial state ŝ;
  od
end
```

Figure 1. Main Idea of the Algorithm

We use programs written in the programming language Non-deterministic While Programs (NWP) [15]. A *process* in this language consists of nondeterministic assignments, send and receive commands, and of IF and WHILE constructs. A *program* is parallel compositions of processes. Each process has its own set of local variables and is not allowed to examine variables belonging to other processes or update them.

The Kripke structure describing semantics of the program is created in a classic way. States are tuples containing for each process its program counter *location* and values of its variables. There are two types of transitions – either one of the processes makes a step while the others wait, or a communication between two computers occurs, i.e. one of the processes performs a *send* command, some other process performs a corresponding *receive* command and the rest of the processes wait. Let us denote by $struct(P)$ the Kripke structure for a program P.

NWP encompasses all the essential programming constructs so that it has similar power to "real" programming languages, except that it does not allow dynamic memory allocation.

We consider a modification of *partition graph* as introduced in [9]. A partition graph of a process P is a finite graph representing a decomposition of P into several subprocesses while maintaining the original flow control.

The vertices of the partition graph are Kripke structures with border states, each one representing a sub-process of P or a boolean expression. A vertex representing a subprocess P' is called an *s-vertex* and is defined as $struct(P')$ (which is a fragment in $struct(P)$). A vertex representing a boolean expression is called a *b-vertex*. Let B be the boolean expression evaluated in an IF or WHILE command with a location l. Then the corresponding b-vertex is defined as a fragment in $struct(P)$ containing the set of states with the location l. Edges in the partition graph follow the control flow in P. It holds that there is an edge in the partition graph from n to n' iff there is a border state in n which2 belongs to n' but is not a border state in n'. The notion of partition graph is a generalization of a well-known notion of control flow graph. However, a control flow graph is a partition graph where each s-vertex represents an atomic command. We denote by $pg(P)$ the set containing all possible partition graphs of P, representing different ways of partitioning P into its sub-processes.

Let we have k workstations available. One possible way to define the partition function is to choose an arbitrary partition graph $G \in pg(P)$ with k vertices and make each computer handle exactly one vertex from G. The communication between two computers i and j occurs when the corresponding vertices are connected with an edge in G.

However, for relatively small number of computers and a large program, the b-vertex represents a structure that is typically much smaller than the structure represented by a s-vertex. Thus, another variant is to choose a partition graph with k s-vertices and with an arbitrary number of b-vertices. Each computer i thus handles one s-vertex and several b-vertices. As the communication proceeds along edges of G, the b-vertex should be handled by a computer that handles some of its successors.

This type of partition function has the tendency to place states with subsequent locations (in the sense of control flow, hence syntactical blocks of code), on one computer. In contrast to a random partition of the state space this method reduces the necessary communication.

Let us consider the program $Z = P_1 \| \ldots \| P_n$. The idea of partitioning this state space is to partition each state according to locations (i.e. according to a partition graph) of a process P_a for a fixed $a \in \{1, \ldots, n\}$. Such a partition function preserves some syntactic structure of the program as well.

The above described partition functions split the state space according to locations, i.e. to program counter. The other possibility is to partition according to values of program variables.

Consider P' be a block of a program P corresponding to an s-vertex v_s of an arbitrary partition graph. Suppose x is a variable that is not defined in P', i.e. x neither occurs on left-hand side of an assignment command nor at any *receive* command. Let us suppose we want to split v_s further to k computers. If we place all states with a given value of x in one computer, then there will be no transition of the whole system that leads from one computer to another.

However, the larger the block P' is, the less likely is to find a variable that is not defined there. Thus, we would like to improve this technique using *abstraction* to suitably split the domain D of a variable into several parts or *slicing* to properly determine the parts of the program where a variable is not changed. This is a goal of ongoing research.

The main aim for future is to implement the distributed algorithm and to compare various partitionings and their influence to the performance of the algorithm.

References

[1] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *SPIN'2001*, volume 2057 of *LNCS*, pages 217–234, Toronto, Canada, 2001. Springer-Verlag.

[2] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. Submitted, 2002.

[3] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS'97*, pages 81–102, 1997.

[4] L. Brim, J. Crhová, and K. Yorav. Using Assumptions to Distribute CTL Model Checking. In *PDMC'02*, 2002.

[5] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *International Conference on Concurrency Theory*, pages 168–182, 2000.

[6] O. Burkart and B. Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.

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

[8] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to modular model checking. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 22, 2000.

[9] K. Laster and O. Grumberg. Modular model checking of software. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 20–35, 1998.

[10] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *SPIN'99*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[11] P.-A. Masson, H. Mountassir, and J. Julliand. Modular verification for a class of PLTL properties. In *LNCS*, volume 1945, pages 398–419. Springer-Verlag, 2000.

[12] U. Stern and D. L. Dill. Parallelizing the murφ verifier. In *CAV'97*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.

[13] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science: Background - Computational Structures (Volume 2)*, pages 477–563. Clarendon Press, Oxford, 1992.

[14] Y.-K. Tsay. Compositional verification in linear-time temporal logic. In *FoSSaCS 2000*, pages 344–358, 2000.

[15] K. Yorav. *Exploiting Syntactic Structure for Automatic Verification.* PhD thesis, Technion, Haifa, Israel, 2000.