Distributed Verification with LoLA

Karsten Schmidt

Abstract

We report work in progress on a distributed version of explicit state space generation in the Petri net verification tool LoLA. We propose a data structure where all available memory of all involved workstations can be fully exploited, and load balancing actions are possible at any time while the verification is running. It is even possible to extend the set of involved workstations while a verification is running.

1 Introduction

In explicit state space verification, we verify a system by explicitly enumerating the reachable states and evaluating temporal logic formulas through simple search algorithms on the constructed state space [VW86, CES86]. The well known state explosion problem is tackled by constructing a reduced state space that is smaller than the original one, but preserves by construction the property to be verified. Among the main reduction techniques are symmetries [HJJJ84, Sta91], the partial order reduction [Val88], and, for Petri nets, the coverability graph construction [KM69]. Verifying temporal logic properties on a reduced state space is extremely efficient (linear time with very small coefficients). The mentioned reduction techniques are quite efficient, too. Most people in the area agree that every run of a verification tool ends after few hours—either with a result, or through memory overflow. Since computer aided verification is not necessarily an interactive task, time is not as crucial a resource as elsewhere. Thus, in difference to other application areas, space is the bottleneck ressource in explicit state space verification.

In most environments, many computers linked in a local area network are available, and their resources are hardly exhausted by their dedicated applications. Using the combined ressources of a network of workstations is thus a cheap alternative to buy powerful verification equipment. It is therefore desirable to have algorithms for distributing state space verification to a heterogenious network of workstations.

Previous work on distribution of explicit state space generation was mostly based on using a hash function on the states. The hash value of a state determines the machine that is responsible for storing the state and to generate its successor states. In the algorithm used in the Murϕ tool [SD97], speeding up run time was the main objective. They used a time-expensive procedure to compress states before storing them, and distributed state space generation helped them to compute many of these compressions in parallel thus obtaining significant speed-ups. They did not, however, address the objective of gaining additional memory. Though they reported that they obtained an even distribution of states among the workstations in their examples, it is not clear whether this observation would be equally true if we dealt with hundreds of workstations rather than with tens. In fact, a hash function usually does not exploit particular structural knowledge about the system, so the best behavior we can expect is a random-like (uniformly distributed) assignment of hash values. Unfortunately, in such a stochastical setting, the expected number of states at the moment when the first machine runs out of memory grows much

slower than the number of participating machines. It is not easy to change a hash function once a state space generation is running, so there are only few possibilities to react online to an unbalanced distribution of the state space.

In the sequel, we propose a different distribution scheme, with main emphasize on the full exploitation of memory on all involved workstations. Our data structure offers interesting opportunities for load balancing on-the-fly. Our procedure is able to tolerate adding new processes to a running verification task. Furthermore, we do not want to rely on an expensive local compression procedure in order to decrease the communication bandwidth. On the other hand, we put less emphasize on the run time issue. We must admit that we have only limited experience with our data structures at this time. We must therefore leave final conclusions concerning performance of this technique to future research. We find it nevertheless very promising, particularly for its capabilities to control the load on all participating machines throughout the whole process of state space exploration.

2 Data structure

Our data structure can be seen as a distributed implementation of a binary decision tree. Note that we do not consider symbolic model checking where this tree would appear in a compressed form [BCM92]. With our approach, we cover *explicit* state space verification. This is motivated bt the observation that for some classes of systems (in particular, asynchronously communicating distributed systems) explicit model checking outperforms symbolic model checking. This good performance of explicit model checking is based on the strength of explicit state space reduction techniques such as partial order reduction which can not easily be combined with symbolic verification methods.

We assume that the participating workstations are fully interconnected via point to point message passing. We assume further that the (asynchronous) message passing protocol meets the following specification:

- Messages cannot get lost nor corrupted;
- The order of arrival can be different from the order of sending;

This specification is more strict than the standard protocol UDP, where messages can get lost (and which is a broadcast protocol), but less strict than TCP/IP, where the order of messages is preserved. More relaxed specifications enable more efficient implementations. In the prototype implemenentation of our work in the tool LoLA [Sch00], we built a simple protocol on top of UDP that works more efficiently than TCP/IP and meets exactly the required specification.

A binary decision tree stores a set of boolean vectors of dimension n. A full decision tree is a complete binary tree of depth n where the leafs are labeled with true or false. We attach a depth to each vertex in the tree, starting with depth 0 for the root, and ending in depth n for the leafs, Each boolean vector b corresponds to a particular leaf, in other words a path, in the tree. This leaf can be reached from the root by taking, at depth k, the left successor if $b[k] = \text{false}$, and the right successor if $b[k] = \text{true}$. A decision tree stores the set of vectors (states) the corresponding leaf of which is labeled true. In an actual binary decision tree, only the vertices that can reach a leaf labeled true are present. The size of an actual decision tree is proportional to the size of the represented set of states. In the sequel, we consider actual decision trees only.

We distinguish a logical and a physical distribution of an actual binary decision tree (in the sequel, we call an actual binary decision tree just decision tree). Let $\Pi = \{P_1, \ldots, P_n\}$ be the processes participating in state space generation. Assume

that every vertex in the original decision tree is labeled (coloured) by an element of Π. We say that process P_i *owns a subtree* of the decision tree iff the root of that subtree is coloured P_i. P_i *owns a state* iff the leaf representing that state is coloured P_i. The process owning a state is responsible for computing that state's successors. The owner of a subtree is responsible for storing (and then owning) states belonging to that tree, or forwarding them to other processes if those processes own sub-subtrees. The owner of a subtree can shift parts of its realm to other processes. The owner of the root of the decision tree is called *master*. The actual distribution is obtained by starting a local search in the master process, then shifting repeatedly subtrees to other processes.

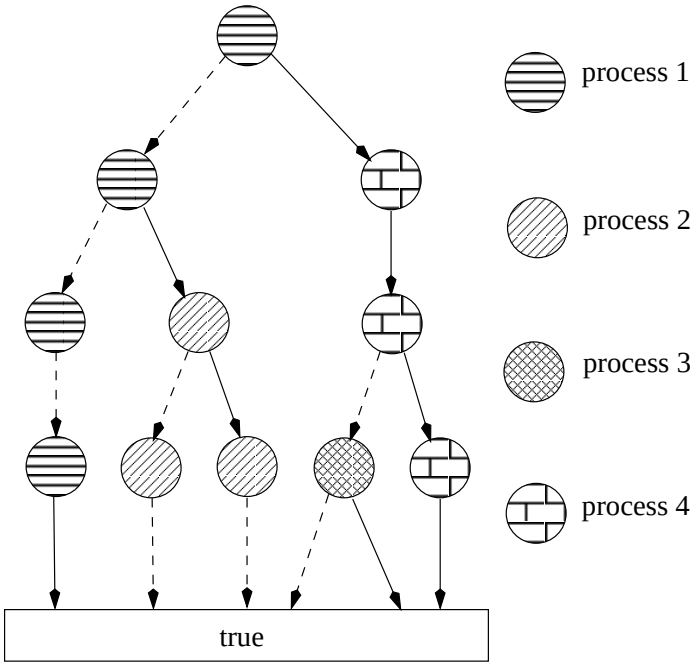

Figure 1: A logically distributed binary decision tree, involving 4 processes. Dashed lines are assumed to be labeled 0, solid lines are labeled 1. Process 1 is master and owns state (0,0,0,1). Process 2 owns states (0,1,0,0) and (0,1,1,0). Process 3 owns (1,1,0,0) and (1,1,0,1). Process 4 owns (1,1,1,1).

We refer to the *physical* data structure as the local data structures present in each process, collectively implementing the logical data structure.

Every process holds as its physical data structure a structure similar to a binary decision tree. As the only difference, there may be inner vertices without successors. As in the logical data structure, every vertex is coloured with a process identifier. However, information stored in each process is only partial. The local tree of P_i consists of the following elements contained in the logical data structure:

- all vertices coloured P_i;
- all vertices on paths that lead from the root to vertices coloured P_i;
- all immediate successors of vertices coloured P_i (if such successor is present in the logical data structure);

The size of a physical data structure is proportional to the number of states owned by P_i. Furthermore, every process has full information about the states that it is owning, so being able to compute its successors.

The colors of vertices not coloured P_i are not necessarily the same in the local tree of P_i as they are in the logical data structure. This reflects the fact that a process has only partial knowledge of the evolution of the logical data structure in other processes. The color of a vertex v coloured $P_j \neq P_i$ in the logical data structure, is P_k iff P_k is the color of the deepest immediate successor of a node coloured P_i on the path from v back to the root in the logical data structure (P_k is the root itself if no nodes coloured P_i are on the path from v back to the root). See Fig. 2. In other words, P_i "knows" only the immediate successors of its own vertices (besides the master), and these immediate successors manage the whole subtree they own.

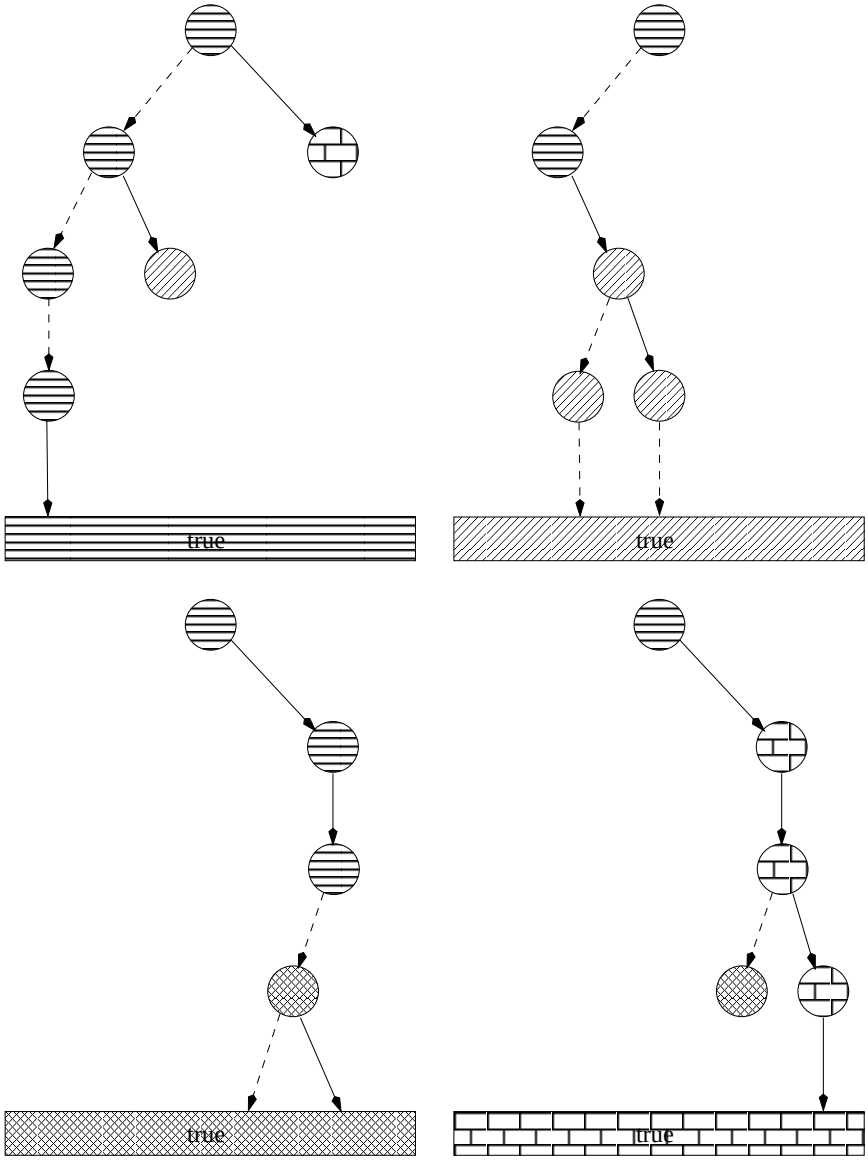

Figure 2: Local data structures of the four processes corresponding to the logical data structure above.

It is immediately clear that the logical data structure can be retrieved from the collection of all physical data structures in the participating processes.

3 Operations on the data structure

The main operation to be implemented on our data structure is search-and-insert. Given a state that has been computed by some state, we need to find out whether the state is already present in the logical data structure. If not, we need to add a new branch to the logical decision tree, assign colors, and let the process that becomes owner of the new state, compute its successors. For the process issuing the search-and-insert request, it is only important whether it is obliged to compute that state's successor or not. There are two possible reasons why a process does not need to compute successors—if the state already exists, or if some other process is responsible for computing those successors. Which of these two cases applies is irrelevant for the continuation of the process issuing the request.

Search-and-insert is implemented by two types of actions in processes. A SEARCH action is parameterized with a state. It can be triggered locally, through the computation of a state in the local state space exploration, or by a message from another process. SEARCH consists of traversing the logical data structure from the root downwards, according to the path defined by the given state. If that path does not end in a vertex owned by the executing process, new messages are triggered. If it does end in a vertex owned by the executing process which is not a leaf, the state is inserted by appending the corresponding branch to the decision tree, or by delegating it to another process, using a specific DELEGATE message.

A DELEGATE action is parameterized with a state and a depth (a number k). It requests the process to assume ownership of the given state, and for the whole subtree defined by that state's prefix up to depth k. DELEGATE is the tool for shifting subtrees to other processes. DELEGATE can be sent only by processes owning the immediate predecessor of the shifted subtree.

We consider first the implementation of a SEARCH action in P_i, triggered by a message or a request in the local search procedure. It starts with a traversal of the local tree, according to the path defined by the given state. This traversal ends in a vertex at some depth k.

Case 1: If $k = n$ (the size of the state), the state exists already, so we do not need to trigger computation of its successors. No modification of any search structure applies, and no further message is issued.

Case 2: If $k < n$, and the final vertex on the traversed path is coloured P_i, then the state is new. In this case, there are two options. First, P_i can decide to take this state. In this case, it adds the remaining vertices for representing the state in its own physical data structure. If the search request is the result of a search message, it adds the state to the queue of states to be locally explored. If the search request has local origin, the search procedure continues exploring successors of the considered state immediately. Logically, we have added a branch in P_i's color to a vertex in P_i's color, so no physical data structures of other processes need to be involved. Second, instead of storing the state locally, P_i can decide to delegate the state, together with some subtree, to another process. In this case, it sends a DELEGATE message to a process $P_j \neq P_i$, using the current state as parameter as well as a number k that is larger than the depth k of the last traversed vertex. Locally, P_i inserts vertices corresponding to all components of the state up to depth k. It colors the new vertex at depth k with P_j, and all remaining inserted vertices P_i. In this case, local search does not compute successors since the receiver of the DELEGATE message becomes responsible for executing this task. The option to pass a subtree to another process is the only available one if a process has run out of memory.

Case 3: If $k < n$, the last vertex on the traversed path is P_j, different from P_i, and the whole traversed path does not contain any vertices labeled P_i then

we trigger a SEARCH message with the given state to P_j. In this case, P_j is the master. Locally, successors of the given state are not computed. The master owns the state or is obliged to dispatch the state to a responsible process which is definitely different from P_i.

Case 4: If $k < n$, the last vertex on the traversed path is P_j, different from P_i, and the traversed path contains vertices labeled P_i then we trigger a DELEGATE message with the given state to P_j. As depth for DELEGATE, we use the depth of the immediate successor of the deepest node coloured P_i on the traversed path. Locally, successors of the given state are not computed.

The fourth case is a tricky one. The fact that we send a DELEGATE message rather than a SEARCH message may be surprising. The reason for this choice is that we rely on a message passing protocol that allows messages to overtake. The local data structure of P_i means that, at some point in the past, P_i has delegated a subtree to P_j, the same subtree as coded in the new DELEGATE message. If we sent a simple SEARCH message in case 4, the new SEARCH message could overtake the original DELEGATE message. In that case, P_j could deal with the message differently, for instance delegate some subtree to a third process, or back to P_i. This would result in inconsistencies of the physical data structures. Using the DELEGATE message, P_j becomes responsible for the same subtree as with the original DELEGATE message, so the order of arrival of the two DELEGATE messages does not matter. All we need to take care of is to let a process tolerate receipt of multiple DELEGATEs for one and the same subtree.

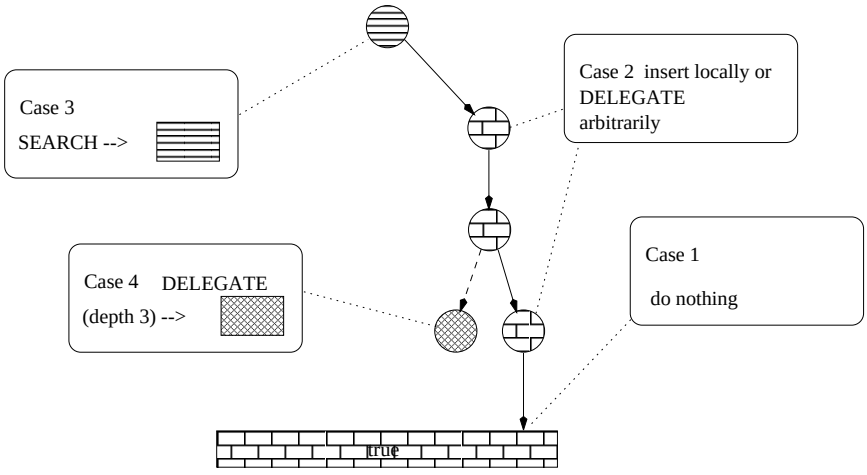

Figure 3: Reaction of a process to a SEARCH request, depending on the deepest matching vertices of the searched state

The SEARCH action, seen as distributed over several processes, terminates in every case. In cases 1 and 2, it terminates locally. In cases 3 and 4, we can assign a termination function. Let $t = 0$ for a sent SEARCH message, and t be the contained depth for a DELEGATE message. This number is strictly increasing with every passed message. In case 3, it is increasing since $t = 0$ in the sending process (by case assumption), and $t > 0$ in any subsequent message sent by the receiving process which is the master and owns at least the root. For case 4, the receiving process owns at least the immediate successor of the last node owned by the sending process, so t is increasing as well, given that execution of DELEGATE terminates.

Execution of a DELEGATE action in P_i starts as well with a local traversal, according to the state parameter.

Case 1. If this traversal ends earlier than the depth parameter k, the last vertex in this traversal must be coloured with an identifier different from P_i. Having an own vertex without successor means that in our SEARCH/DELEGATE protocol only P_i itself can decide the logical color of the immediate successor, in contradiction to the fact that another process claims ownership for the vertex according to the given state at depth $k-1$. In this case, P_i must accept the new subtree. It inserts at least the remaining vertices down to depth k, colors the vertex at depth k P_i, and the remaining inserted vertices with the color of the last traversed vertex. Then P_i decides whether to accept the state itself or to delegate it further, with some depth greater than k. The remainder of this procedure works exactly like case 2 of a SEARCH action. The requirement that P_i needs to accept the subtree is necessary to avoid data structure inconsistencies since the sending process has already marked P_i as the owner of that subtree and does not wait for acknowledgment. Additionally, the fact that a process can respond to a DELEGATE message at most with a DELEGATE message containing a larger depth parameter, guarantees eventual termination.

Case 2. If the traversal ends at a depth $\geq k$ (the depth parameter), then it can be treated as a SEARCH. In that case, P_i is already aware of its ownership of the delegated subtree (otherwise, search traversal could not have ended beneath the passed depth).

It is easy to see that, for every state, the sequence of SEARCH and DELEGATE messages terminates, and the last process involved in that sequence finally owns the state (and is responsible for computing successors).

4 Example

Assume a distributed situation as depicted in Fig. 2. We play first a scenario where the diagonal process produces state (1,1,0,0)—obviously owned by the crossed process in the logical data structures (remember that dashed lines represent 0 and solid lines 1). The local traversal in the diagonal process ends in depth 0. Case 3 applies, and a SEARCH message is sent to the horizontal process (the master). The master conducts a local traversal that terminates in the bricked vertex (the solid line matches the first component of the vector. The master issues a DELEGATE message with depth 1 to the bricked process. Local traversal in the bricked process ends in the crossed vertex (the first 3 components of (1,1,0,0) match). Consequently, a DELEGATE with depth 3 is sent to the crossed process. The crossed process traverses the state completely. None of the processes needs to compute successors of (1,1,0,0) unless that state is still pending in the crossed processes queue.

As a second scenario, assume that the diagonal process initiates a search for (1,0,0,0). As before, it sends a SEARCH message to the master which forwards a DELEGATE with depth one to the bricked process. Local search in the bricked process ends at depth 1 (only the first component of (1,0,0,0) matches. It is now up to the bricked process to decide whether to accept the state. If it does, it inserts a chain representing the remainder of (1,0,0,0) to its local tree and adds (1,0,0,0) to its queue. If not, it issues a DELEGATE, say to the diagonal process with a depth greater than 1, say, depth 3. The diagonal process conducts another local traversal, still ending at depth 0. Case 1 of the DELEGATE protocol applies, and the diagonal needs to insert a new subtree. Assuming that the diagonal process accepts the state, the local data structures of the diagonal and the bricked processes now look as depicted in Fig. 4. The physical data structures of the remaining processes remain unchanged.

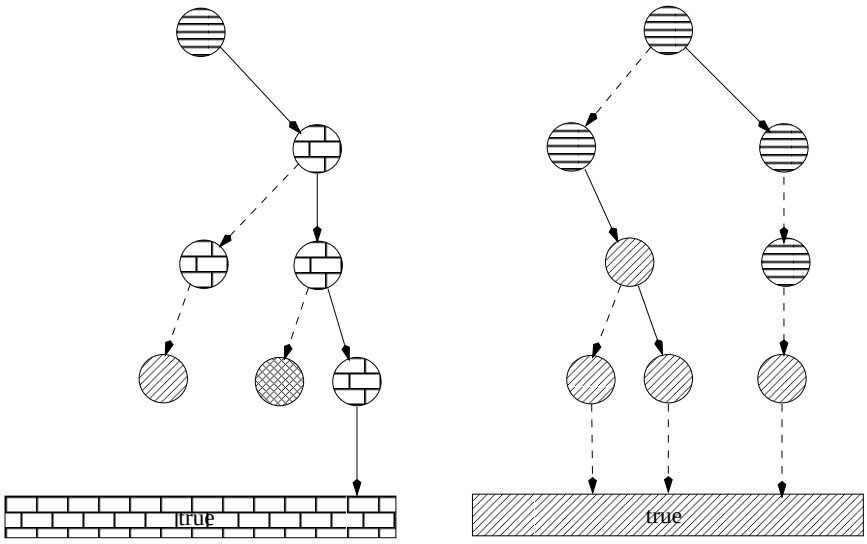

Figure 4: New local data structures after delegation of state (1,0,0,0) from the bricked process to the diagonal process, starting from the situation in Fig. 2. It is a feature and not a bug that the new chain in the diagonal process is mostly coloured horizontally. This color means only that corresponding search requests are sent to the master who forwards them to the bricked process as the real owner of that subtree.

5 Load balancing and reaction at end of memory

At every stage, a process that is about to store a state locally can decide to delegate the state instead to another process. Only delegated subtrees must be stored unconditionally. A process delegates states (and with it whole subtrees) at least if it is about to run out of memory. It can choose a new host arbitrarily. However, processes that are already out of memory should be exempt. This way, a process out of memory is only as long receiving delegated subtrees as it takes other processes to recognize that fact (via, say, broadcast messages containing a processes state sent on a regular basis by every process). In other words, a process can exhaust its local memory, up to a small amount of memory for accepting pending delegated subtrees.

It is also possible to delegate subtrees before running out of memory. Through delegation, the master who begins state space exploration locally, starts to distribute load after having computed a first initial decision tree that consists of a small number of states. Delegation can be used to shift load away from overloaded processes. The decision whether or not to delegate states, and where, is made exclusively by the delegating process, while state space exploration is running. So, in difference to hash techniques, available information about actual load situation in all processes can influence that decision.

Using this data structure, actual computation of successor states differs only little from sequential depth first search. As a minor difference, search and insert are now a monolithic procedure. Other than this there is no change inside a running depth first search since we have seen that a process does not need to distinguish between an existing state and a state it is not responsible for. The main difference is that there is now a queue of pending "initial" states. Whenever a depth first search is completed, a new state is taken from that queue, and a new depth first search is launched. The queue is filled with states that are received through DELEGATE and SEARCH messages from other processes where this process assumes ownership.

Through a specific termination protocol, distributed search ends as soon as all processes have finished their searches and have empty queues.

6 Conclusion

The proposed algorithm is able to compute the set of reachable states. So far, we do not have an efficient solution for storing events, or to compute strongly connected components. Thus, our distributed search algorithm can, at this time, be used only for properties that can be evaluated from the plain set of reachable states, such as reachability properties.

Whether the proposed data structure, hash values, or other techniques are used for distribution, distributed state space search has a major time disadvantage, compared with local state space exploration. Since states are frequently shifted between processes, the advantages of incremental computations of successor states, set of enabled actions etc. are lost. Since our implementation of local state space exploration uses all these techniques, we cannot exhibit experiments where distributed state space exploration runs faster than local exploration. We can, however, show that we are able to solve larger problems than with local search. We found that network bandwidth of a usual local area network with mixed 10MB and 100MB Ethernet connections is sufficient to satisfy the communication requirements with reasonable delays. Compared with distribution based on hash functions, we believe that our distribution scheme requires less states to be shifted to other processes. We argue that a successor state is equal to the original state in most components. Thus, in more cases than in a random setting, the successor state falls into the same subtree (have a common prefix of considerable length) as the original state thus not requiring interprocess communication.

As an example, the state space of a 1000 philosophers system reduced by partial order reduction has 2,997,002 states and 3,997,000 edges. It cannot be verified on a single workstation. On a network of 15 SUN workstations, the state space could be constructed within less than 5 hours. Thereby, several of the 15 involved machine reached their local memory limits.

References

[BCM92] J. R. Burch, Edmund M. Clarke, and Kenneth L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), June 1992.

[CES86] Edmund M. Clarke, E. M. Emerson, and A. P. Sistla. Automatic verification of finite-state concurent systems using temporal logic specifications. *ACMM Transactions on Programming Languages and Systems*, 8(2):244 – 263, April 1986.

[HJJJ84] Huber, A. Jensen, Jepsen, and K. Jensen. Towards reachability trees for high–level petri nets. In *Advances in Petri Nets 1984, Lecture Notes on Computer Science 188*, pages 215–233, 1984.

[KM69] R. M. Karp and R. E. Miller. Parallel programm schemata. *Journ. Computer and System Sciences 4*, pages 147–195, Mai 1969.

[Sch00] K. Schmidt. Lola – a low level analyzer. *Proc. 21th Int. Conf. Application and Theory of Petri nets, LNCS 1825*, pages 465–474, 2000.

[SD97] U. Stern and D.L. Dill. Parallelizing the murphi verifier. *Proc. Int. Conf. Computer Aided Verification, LNCS 1254*, pages 256–267, 1997.

[Sta91] P. Starke. Reachability analysis of petri nets using symmetries. *J. Syst. Anal. Model. Simul.*, 8:294–303, 1991.

[Val88] A. Valmari. Error detection bu reduced reachability graph generation. *Proc. of the 9th European Workshop on Application and Theory of Petri Nets, Venice*, 1988.

[VW86] M.Y. Vardi and P. Wolper. An automate-theoretic approach to to automatic program verification. *Proc. IEEE Symp. Logic in Computer Science*, pages 332–344, 1986.