Distributed LTL Model-Checking in SPIN*

Jiri Barnat[1], Lubos Brim[1], and Jitka Stříbrná[2]

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

[2] Department of Computer and Information Science, University of Pennsylvania
200 South 33rd Street, Philadelphia, PA 19104, USA
jitkas@saul.cis.upenn.edu

Abstract. In this paper we propose a distributed algorithm for model-checking LTL. In particular, we explore the possibility of performing nested depth-first search algorithm in distributed SPIN. A distributed version of the algorithm is presented, and its complexity is discussed.

1 Introduction

Verification of complex concurrent 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. As a matter of fact, in application of such tools to practical verification problems the computational power available (memory and time) is the main limiting factor. Recently, some attempts to use multiprocessors and networks of workstations have been undertaken.

In [UD97] the authors described a parallel version of the verifier Murφ. In their approach, the table of all reached states is partitioned over the nodes of the parallel machine, which allows the table to be larger than on a single node. The explicit state enumeration is then performed in parallel. For the model-checker SPIN [Hol97], a similar approach towards distributed reachability analysis was proposed in [LS99]. This distributed version of SPIN uses different ways to partition the state space than Parallel Murφ. Yet another distributed reachability algorithm was proposed in [AAC87], but has not been implemented. Other recent attempts to use distributed environment of workstations for parallel model-checking are [HGGS00,BDHGS00].

Unlike the original SPIN, the distributed algorithm proposed in [LS99] performs a non-nested, depth-first search of the state space. It carries out reachability analysis, however, it does not implement model-checking of LTL formulas. In this paper, we propose a distributed algorithm based on nested depth-first search, that model-checks LTL formulas. The basic idea is as follows. The algorithm starts to explore the state space in (not necessarily nested) distributed

* This work has been supported in part by the Grant Agency of Czech Republic grants No. 201/00/1023, and 201/99/D026.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 200–216, 2001.

depth-first manner, similarly to the distributed SPIN of [LS99]. When an accepting state of the Büchi automaton (called a *seed*) is visited, it is stored in a special data structure (*dependency structure*). Nested DFS procedures for seeds are initialised separately in an appropriate order 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 synchronisation during its execution. Our aim was to experimentally explore how much will synchronisation influence the overall behaviour of the tool. To our surprise, even using this very simple method we were able to deal with verification problems larger that those that could be analysed with a single workstation running the standard (non-distributed) version of SPIN.

An experimental version of the algorithm has been implemented and a series of preliminary experiments has been performed on a cluster of nine PC based Linux workstations interconnected with a 100Mbps Ethernet, using the MPI library.

The rest of the paper is organised as follows. We start with a section which briefly describes the already existing distributed version of SPIN. The following section explores specific difficulties in direct extention of the distributed SPIN for the purposes of checking LTL formulas, and also proposes a possible solution. Then we describe the additional data structures required by our algorithm and present the pseudo-code. Finally, the complexity and effectiveness of the algorithm are discussed.

2 Distributed SPIN

In this section we briefly summarise the main idea of the distributed version of SPIN as presented in [LS99]. The algorithm partitions the state space into subsets according to the number of network nodes (computers). Each node is responsible for its own part of the state space. When a node computes a new state s, it runs the procedure `Partition(s)` to find out whether or not the state belongs to its own state subset. If the state is local, the node continues locally, otherwise a message containing the state is sent to its owner node. Local computations proceed in the depth-first search manner using the procedure DFV (a slight modification of SPIN's DFS procedure). However, due to distribution of work, the *global* searching does not follow the depth-first order, which is one of the reasons for the algorithm's inadequacy for LTL model-checking. An auxiliary data structure `U[i]` is constructed to hold information about pending requests on a node `i`. The distributed algorithm terminates when all `U[i]` queues are empty and all nodes are idle. A *manager process* is used to detect termination. Each node may send two kinds of messages to the manager. One message will be sent whenever a node becomes idle, a different message will dispatched when it becomes busy. Correct termination requires each node to reconfirm, and the overall number of messages sent and received be equal.

The pseudo-code bellow illustrates the original algorithm used in the distributed version of SPIN.

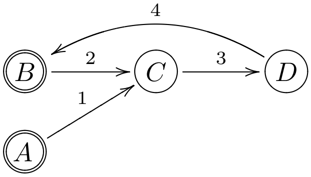
```
procedure START(i, start_state);
begin
  V[i] := {}; { already visited states }
  U[i] := {}; { pending queue }
  j := Partition(start_state);
  if i = j then
  begin
    U[i] := U[i] + start_state;
  end;
  VISIT(i);
end;

procedure VISIT(i);
begin
  while true do
  begin
    while U[i] = {} do begin end;
    S := extract(U[i]);
    DFV(i,S);
  end;
end;

procedure DFV(i, state);
begin
  if not state in V then
  begin
    V[i] := V[i] + state;
    for each sequential process P do
    begin
      nxt = all transitions of P enabled in state;
      for each st in nxt do
      begin
        j = Partition(st);
        if j = i then
        begin
          DFV(i, st);
        end
        else
        begin
          U[j] := U[j] + st;
        end;
      end;
    end;
  end;
end;
```

3 Problems with Extending the Distributed SPIN

When we want to adopt directly the technique of nested DFS approach to distributed computing we encounter two main problems. By simply allowing more nested DFS procedures for different seeds we may obtain an incorrect result, which can be demonstrated by the following example:

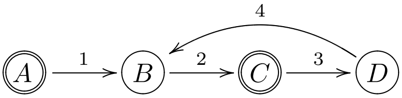


If two nested DFS procedures were run simultaneously on both seeds A and B then the cycle through the state B might not be detected. The outcome depends on the relative speeds of both nested DFS procedures. If the nested DFS procedure initialised from A visits the state C first, the nested DFS procedure starting in B will not be allowed to continue through C and, as a result, the cycle B, C, D, B will not get detected.

In general, whenever the subgraphs generated by two different seeds do not have an empty intersection, there is a possibility that some cycle may not be detected. A simple criterion to determine whether it is possible to run two or more nested DFS procedures in parallel is to find out whether or not the corresponding intersections are empty. However, verifying this condition could result in searching the entire state space.

An obvious solution to this problem is to store for each state the information in which nested DFS procedure it was visited. This would mean to store a nontrivial amount of information because each state might be a seed, in which case the space complexity of the additional storage may turn out to be quadratic with respect to the number of states. This is the reason why we do not allow to run several nested DFS procedures simultaneously.

Another problem is the following. The original distributed version of SPIN does not preserve the depth-first-search order of visited nodes. This is not a problem in case of reachability analysis because the only relevant information is whether or not a given state was visited. However, this may pose a threat to the correctness of the full model-checking procedure, which is shown on the following example:



A correct run through this graph (when DFS procedure goes back we put a dash over the name of the edge, runs of nested DFS are put into brackets), in which the cycle through C is detected, is:

$$1, 2, 3, 4, \overline{4}, \overline{3}, [3, 4, 2, \circ]^C$$

An incorrect run, in which the correct order of seeds is not preserved, is for instance

$$1, 2, 3, 4, \overline{4}, \overline{3}, \overline{2}, \overline{1}, [1, 2, 3, 4, \overline{4}, \overline{3}, \overline{2}, \overline{1}]^A, [3, \overline{3}]^C$$

A possible solution to this problem was proposed in [LS99]. It is based on the original distributed SPIN procedure for reachability analysis and consists of adding synchronisation to the distributed algorithm. The synchronisation is done by sending suitable acknowledgements that will exclude the possibility of a seed being processed before another seed that is "below", thus avoiding incorrect runs where cycles may not be detected. This solution cuts off any parallelism and so in fact does not perform better than the plain sequential algorithm.

What we propose here is an attempt to provide a more subtle solution that makes it possible to effectively tackle larger tasks than standard SPIN within reasonable space limits. The idea is to reduce the necessary synchronisation by using additional data structures with limited space requirements.

Yet another important issue that must be taken into consideration is that any distributed extention of SPIN should incorporate the main memory and complexity reduction techniques available in SPIN, such as state compression, partial order reduction, and bit state hashing. The approach that we present here is compatible with such mechanisms as much as possible.

4 Distributed Model-Checking Algorithm

From the discussion in the previous section it is clear that it is crucial to take care about the order in which individual nested DFS procedures are performed. We need to make sure that the following invariant will always hold for each distributed computation:

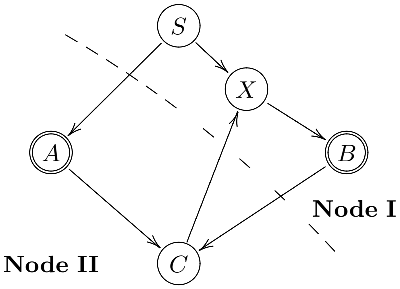
> *A nested DFS procedure is allowed to begin from a seed S if and only if all seeds below S have already been tested for cycle detection.*

Different states, hence also seeds, of the state space are processed on different nodes in accordance with the partition function. Hence, it may occur that for a seed S, a computation of the subgraph generated by S is interrupted and continues on a different node. We need to keep track of such situations.

In order to represent dependencies between states (corresponding to computations transferred to other nodes) we shall build a dynamic structure that will keep this essential information. We need to remember the states which caused the computation to be transferred to other nodes. We call these states *transfer states*. Two border states are involved in the transfer of computation and we need to remember only one of them. A *border state of node m* is a state belonging to the node m whose incoming or outgoing edge crosses to another node. We shall also include all the seeds that appear during the computation in order to ensure the correct order of performed nested DFS procedures. Each node maintains its own structure and we call it *DepS* (Dependency Structure).

4.1 Dependency Structure

Dependency structure (DepS) for node n is a graph whose vertices are either *seeds* of the local state space of node n or *transfer states for node n*. Vertices can be indexed by a set of node names. A transfer state for node n is either a border state of node n whose predecessor is a border state of another node, or a border state of another node whose predecessor is a border state of node n (see Figure 1 for graphical explanation). The starting state of the entire state space is also a vertex in DepS for the node running manager process. Note that a seed (or the starting state) can be a transfer state. In this case it will occur only once in the structure (as a seed *and* and a transfer state at the same time). Indexes in vertices corresponding to transfer states represent the nodes from which the computation was transferred to the transfer state.



S is the starting state

X is a transfer state for node II and I

A is a seed and a transfer state for I and II

B is a seed

C is a transfer state for node I and II

Fig. 1. Transfer States and Seeds

The edges in DepS for node n represent the reachability relation between states (and provide crucial information to perform nested DFS procedures in a "correct order"). The dependency structure is a forest-like structure and is built dynamically during the computation of the DFS procedure. All vertices in the structure are the states actually *visited* by the algorithm during depth-first search. For each vertex its immediate successors contain the states (seeds or transfer states) which the vertex is "waiting for", in the sense that the states must be processed (checked for nested DFS in case of seeds) before the state corresponding to the vertex can be processed.

The structure DepS is changing dynamically as the computation continues. Whenever the DFS procedure reaches an unvisited seed or a transfer state, a new vertex corresponding to the state is added to the structure as an immediate successor of the last visited state which is present in the structure. Moreover, in the case of a transfer state a request to continue with DFS procedure is sent to the owner node of the transfer state (if not already sent before).

During the computation each node receives requests from other nodes to continue a walk through the state space. These requests are remembered in a local queue of pending requests and the queue is processed in the standard FIFO manner. For each request it is first checked whether the state is in the set of visited states. If it is, then it is checked whether there is any vertex in the (local) DepS structure containing the same state. If yes, then the "name" of the node who sent the request is added to its index. If the request is not in the DepS structure and it is in the set of visited states, then an acknowledgement of the request is sent back. If the request is not in the set of visited states (hence not in DepS) a new *root* vertex is added to DepS with "name" of the sender node as its index.

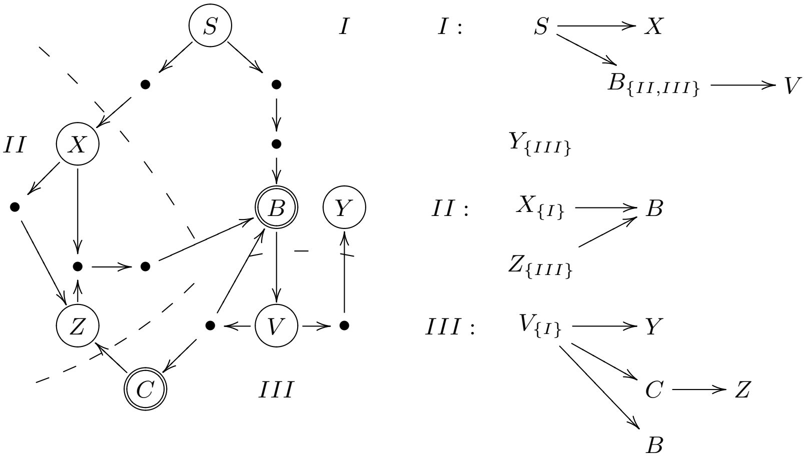


Fig. 2. Example of dependency structures

Removing a vertex from the dependency structure is possible only after the DFS procedure has gone up through the respective state. The removal procedure first checks whether the state associated with the vertex is a seed, in which case the state is added to the global queue of seeds waiting for the nested DFS procedure. For any state all acknowledgements are generated according to the (vertexes) index. Then the vertex is taken out of the structure. Removing any vertex may make its parents' successor list empty and thus cause elimination of the parent and parent's parent, and so on.

Whenever the DFS procedure goes up through a state with a vertex in DepS, it is checked whether the vertex is a leaf of the tree. If it is the case the removal procedure is initiated. Removal procedure can also be initiated by incoming acknowledgements.

In Figure 2, the dependency structures for nodes I, II and III are shown before any vertex has been removed.

4.2 Manager Process

Distributed SPIN uses a manager process that starts the verification program on a predetermined set of network nodes. After it has detected termination, the manager process stops the program, and collects the results. In our version of distributed SPIN we will in addition require that the manager process is in charge of initiating nested DFS procedures, in accordance with the method described in previous sections.

All nodes involved in the computation communicate with the manager process by sending information about their status. The status of each node is determined by: *DFS status, nDFS status, numbers of sent and received requests (DFS and nDFS packets)*. DFS status can be *busy*, *idle-empty-DepS* and *idle-nonempty-DepS*, whereas nDFS status can be only *busy* or *idle*. In this way, the manager process has a local representation of the current state of all the nodes.

Nested DFS Procedures

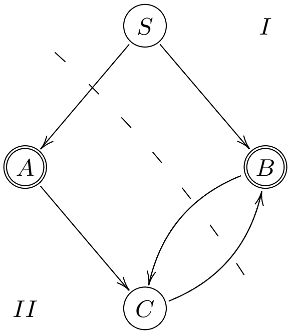
As we have mentioned before, our approach relies on the requirement that only one nested DFS procedure is allowed at a time. This is the reason for sending the seed to the manager process instead of starting a nested DFS procedure immediately during the DFS procedure. The manager process maintains a global queue of seeds waiting for their nested DFS procedures. It starts a nested DFS procedure for the first seed in the queue only if no other nested DFS procedure is running. This will be determined by checking that the nDFS status of all nodes is equal to *idle* and the overall number of all sent nDFS packets equals the overall number of all received nDFS packets, as it was already implemented in [LS99]. After the manager process has started a nested DFS procedure for a seed, it will be removed from the queue.

Termination Detection

The distributed algorithm must terminate when there is no waiting seed in the global queue, no nested DFS procedure is running, the overall numbers of sent and received DFS packets are equal, and all nodes have an *idle-empty-DepS* DFS status.

Termination detection can be handled in a similar way as it is done by distributed SPIN. The only exception is the situation when the overall numbers of sent and received DFS packets are equal, no computer has *busy* DFS status, *but* some node has *idle-nonempty-DepS* DFS status. In this case the manager process asks all the nodes with *idle-nonempty-DepS* DFS status for information about their *DepS* structures. The nodes reply by sending the following elements of their *DepS* structures: $X_Z \xrightarrow{u} Y$, where X is a node from the *DepS* structure which has a nonempty index Z, and Y is the node representing the transfer state for which X is waiting. The edge has a label u that represents the presence of a seed on the path from X to Y, including X and excluding Y, in the original *DepS* structure. So u can be either $1 =$ *with a seed* or $0 =$ *without a seed.*

The manager process constructs a temporary graph out of the received elements, and using standard Tarjan's algorithm [Tar72], it finds a maximal strongly connected component with no outgoing edges in the graph. Note that such a strongly connected component must exist. The manager process checks for the presence of an edge with label 1 in the strongly connected component. If there is no such edge then an acknowledgement is generated for an arbitrary node from the found component, and distributed computation continues in the standard way. In the other case, i.e. when there is a cycle labelled by 1 in the temporary graph, it is clear that a cycle through an accepting state must exist in the original state space and therefore the verified property does not hold. In this case the algorithm terminates immediately. The whole situation is shown in Figure 3.



$$\begin{tikzcd} I: & S \arrow[r] \arrow[dr] & A & \\ & & B_{\{II\}} \arrow[r] & C \end{tikzcd}$$

$$\begin{tikzcd} II: & A_{\{I\}} \arrow[r] & B \\ & C_{\{I\}} \arrow[ur] & \end{tikzcd}$$

Elements sent to the manager process are: $B_{\{II\}} \xrightarrow{1} C, A_{\{I\}} \xrightarrow{1} B, C_{\{I\}} \xrightarrow{0} B$

The constructed graph is: $$\begin{tikzcd} A_{\{I\}} \arrow[r, "1"] & B_{\{II\}} \arrow[r, bend left, "1"] & C_{\{I\}} \arrow[l, bend left, "0"] \end{tikzcd}$$

A strongly connected component without outgoing edges is: $$\begin{tikzcd} B_{\{II\}} \arrow[r, bend left, "1"] & C_{\{I\}} \arrow[l, bend left, "0"] \end{tikzcd}$$

A cycle through an accepting state has been found.

Fig. 3. Example

4.3 The Algorithm

Our algorithm extends the distributed algorithm from [LS99] in the sense that it employs the same distributed method for reachability analysis. This ensures that we do not exclude use of the main memory and complexity reduction techniques available in SPIN.

The underlying idea behind our distributed model-checking algorithm is the following. The whole reachable graph is partitioned into as many regions as the number of network nodes. Each node performs the computation on the states that belong to its own region. When a successor belonging to another region is generated, a message containing the new state is sent to its owner. Received messages are stored in a (remote) queue and processed sequentially. For both DFS and nested DFS procedures, only the yet unvisited states are explored. In contrast with the original algorithm, not all visited states are permanently stored in the set of visited states, only transfer states and seeds. Each node keeps the set of permanently stored states in array `PV[i]`. To prevent cycling through not permanently stored states, the algorithm keeps track of all visited states within the processing of each received request. This temporary set is kept in array `V[i]`, which is initialised to $\emptyset$ before processing another request (state) from the queue of pending requests `U[i]`.

To ensure the right order of nested DFS procedure calls, an additional data structure `DepS` is maintained (see subsection 4.1). This structure is built by procedures `CREATE_IN_DepS(s)` and `ADD_TO_DepS(s1,s2)`, using a temporary pointer `Last_visited`. A root vertex in the dependency structure, which corresponds to state `s`, is created by procedure `CREATE_IN_DepS(s)`. Procedure `ADD_TO_DepS(s1,s2)` creates a vertex corresponding to state `s2`, if it does not exist yet, and adds an edge between the vertices corresponding to states `s1` and `s2`. A vertex representing state `s`, written as ⟨`s in DepS`⟩ in the pseudo-code, is composed of several components (fields): *parent*, *successors*, *state*, *DFS_gone* and *index*. The field *parent* points to the vertex for which this vertex has been created as a successor. The field *index* is a set of node names. The field *DFS_gone* is a flag indicating whether the DFS procedure has already walked through the state up. The meaning of the fields *successors* and *state* is obvious.

Some additional local variables are used in the algorithm. The meaning of `Seed`, `state`, `came_from` and `tmp` is obvious. Variable `toplevel` is used to distinguish whether procedure `DFV` was called recursively from `DFV` or whether it was called from procedure `VISIT`. Variable `Seed_queue` is a global variable which is maintained by `MANAGER_PROCESS`. It represents the queue of seeds waiting for their nested DFS procedures to be started.

All nodes execute the same code. For simplicity we assume that the master node runs the manager process only. The DFS procedure is started by calling procedure `START(i,starting_state)`. The value of `i` is the name of the node (integer). The procedure puts `starting_state` into the queue of pending requests `U[i]` at the proper node `i`. Procedure `VISIT(i)` is called for all nodes except the master node at the end. Procedure `MANAGER_PROCESS` is called at the master node. The task of procedure `MANAGER_PROCESS` was explained in the subsection 4.2.

Procedure `PROCESS_INCOMING_PACKETS` is actually a boolean function which returns `false` if the computation should be stopped for some reason, and returns `true` otherwise. This function plays the role of the client side of the manager process. It updates the numbers of locally received and sent packets and sends this numbers and information about the node status to the manager process. It also processes all incoming packets. The requests are stored in `U[i]` queue, the acknowledgements are collected and procedure REMOVE is called for them. Also all control packets are handled by it.

Procedure VISIT waits for queue U[i] to be nonempty. It collects a request from the queue, and resets variables toplevel and V[i]. In case that the request is a new one (it is not in the set PV[i]), procedure DFV is called. It is necessary to distinguish between the first DFS procedure and the nested DFS procedure. In the case of nested DFS procedure variable Seed must be set, on the other hand in the case of DFS procedure appropriate actions on the DepS structure are performed. The states already processed, which are requested again, are checked for presence in the DepS structure. If the corresponding vertex exists, only its index is updated, otherwise the acknowledgement is generated.

The DFV procedure checks whether the state belongs to the same node. If not, the message containing the state is sent to the node owning the state and the DepS structure is updated. (Note: In the case of nested DFS procedure the seed, for which the nested search is running, is included in the message as well.) When the DFV procedure is called from the VISIT procedure, a new root vertex must be added to the DepS structure, and the state must be stored in the set of permanently visited states PV[i]. In case that state is a seed it is necessary to update the DepS structure. Note that the DepS structure is maintained only for the first DFS procedure and not for the nested DFS procedure. Conversely, the check whether the reached state is Seed is done only in the nested DFS procedure. The CYCLE_FOUND procedure informs the manager process about the fact that a cycle has been found. Before all the successors of state are generated, state is added to the set of actually visited states (V[i]). The unvisited successors are handled by recursive calling of procedure DFV. If a successor is a seed or a transfer state which is already contained in the DepS structure (hence it must appear in PV[i]), the appropriate edge is added to the DepS structure. After all successors of state have been processed and if state is a seed, the check whether there are any more successors of the corresponding vertex in the DepS structure is performed. In case there are no successors, the vertex is removed from DepS by procedure REMOVE.

Procedure REMOVE(vertex) is crucial with respect to maintaining the DepS structure. It is responsible not only for (recursive) deleting of the vertices from the memory but also for sending the seeds to the global queue (Seed_queue), and for sending all appropriate acknowledgements, which is done by procedure ACK(vertex) (see the pseudo–code for details).

Note that the same state can be visited twice, in the DFS procedure and in the nested DFS procedure. To store the information about the fact that the state has or has not been visited in one or both DFS procedures only one additional bit is required (see [Hol91]). We assume that this one additional bit is included in the bit vector representing the state. That is why the bit vectors representing the same state differ in the case of DFS procedure and nested DFS procedure.

The Partition function is used for partitioning of the state space. The choice of a "good" partition function is crucial in the distributed algorithm since a "bad" partition of states among the nodes may cause communication overhead.

This issue was addressed in [LS99], where several partition functions were proposed and tested.

The pseudo-code of our proposed algorithm follows:

```
procedure START(i,start_state)
begin
  DepS := {};
  U[i] := {};
  PV[i] := {};
  Last_visited := nil;
  Seed := nil;
  if (i = Partition(start_state)) then
  begin
    U[i] := U[i] + {start_state};
  end;
  if (i = 0) then
  begin
    MANAGER_PROCESS();
  end
  else
  begin
    VISIT(i);
  end;
end.

procedure VISIT(i)
begin
  while (PROCESS_INCOMING_PACKETS()) do
  begin
    if (U[i] <> {}) then
    begin
      get (state, came_from) from U[i];
      toplevel := true;
      V[i] := {};
      if (state not in PV[i])) then
      begin
        if (Nested(state)) then
        begin
          Seed := state.seed;
          DFV(i,state);
        end
        else
        begin
          DFV(i,state);
          if (<state in DepS>.successors = {}) then
          begin
            REMOVE(<state in DepS>);
          end;
        end;
      end
      else
      begin
        if (state in DepS) then
        begin
          <state in DepS>.index := <state in DepS>.index
                                   + {came_from};
        end
        else
        begin
          ACK(<state in DepS>);
        end;
      end;
    end;
  end;
end.

procedure REMOVE(vertex)
begin
  if Accepting(vertex.state) then
  begin
    Seed_queue := Seed_queue + {vertex.state};
  end;
  ACK(vertex);
  tmp := vertex.predecessors;
  for (i in tmp) do
  begin
    i.successors := i.successors - {vertex};
  end;
  free(vertex);
  for (i in tmp) do
  begin
    if ((i.successors = {}) and (i <> nil)
                            and (i.DFS_gone)) then
    begin
      REMOVE(i);
    end;
  end;
end.

procedure DFV(i,state)
begin
  if (PARTITION(state) <> i) then
  begin
    U[PARTITION(state)] := U[PARTITION(state)] + {state};
    if (not Nested(state)) then
    begin
      ADD_TO_DepS (Last_visited, state);
    end;
    return;
  end;
  if (toplevel) then
  begin
    PV[i] := PV[i] + state;
    if (not Nested(state)) then
    begin
      CREATE_IN_DepS(state);
      Last_visited := state;
    end;
  end;
  if (Accepting(state)) then
  begin
    if (not(toplevel) and (not Nested(state))) then
    begin
      ADD_TO_DepS(Last_visited,state);
      Last_visited := state;
      PV[i] := PV[i] + state;
    end;
  end;
  toplevel := false;
  V[i] := V[i] + {state};
  for (newstate in successors of state) do
  begin
    if (Nested(state) and (Seed=newstate)) then
    begin
      CYCLE_FOUND();
    end;
    if (newstate not in (V + DepS + PV[i])) then
    begin
      DFV(i,newstate);
    end
    else
    begin
      if ((newstate in DepS) and (not in 1stDFS stack)) then
      begin
        ADD_TO_DepS (Last_visited,newstate);
      end;
    end;
    if (Accepting(state) and (not Nested(state))) then
    begin
      Last_visited := <Last_visited in DepS>.parent;
      <state in DepS>.DFS_gone := true;
      if (<state in DepS>.successors = {}) then
      begin
        REMOVE(<state in DepS>);
      end;
    end;
  end;
end.
```

5 Complexity and Effectiveness

We shall try to estimate the overall size of dependency structures constructed during distributed computations. For any node, there are two kinds of states stored in the structure – all *seeds* that are visited by this node, and all states that arise in the computation but are transferred to another node in accordance with the partition function (*transfer states*). The number of the latter is crucial because this can be in the worst case quadratic wrt the overall number of states.

The partition function we employ was originally proposed in [LS99] where it was also shown that with this function, the fraction of transfer states in the global state space is at most $\frac{2}{P}$, where P is the number of processes. The number of transfer states T is bounded by the expression $S \times R$, where S is the number of states and R is the maximum of out-going degrees over all states. R is at most $P \times ND$, where ND is the maximal number of nondeterministic choices of a process. Thus we get that $T \leq (S \times P \times ND)$ and the average number of transfer states is $\frac{2}{P} \times S \times P \times ND = 2S \times ND$. Hence, the number of states stored in the dynamic structure is on average $S + 2(2S \times ND)$ which works out to be $O(S \times ND)$. In most real systems the amount of non-determinism (represented by ND) is limited and small. We may conclude that the memory complexity of the distributed algorithm is on average linear in the size of the state space and the factor given by non-determinism.

We will compare our approach with the simple method of synchronisation proposed in [LS99]. We will look in detail at how the first and nested depth-first-search procedures work. The first DFS searches through the state space and marks accepting states (seeds) with a particular kind of flag. The computation of first DFS is completely asynchronous and never stops before the whole state space is searched trough. The nested DFS searches in a similar, i.e. distributed, way the subgraph rooted in the currently processed seed. If there are seeds ready to be processed, the nested DFS is running in parallel with the first DFS. Therefore our method allows parallel computations which are almost disabled in the simple synchronisation technique.

We conclude the complexity analysis by a brief remark on the time complexity of the algorithm. As we do not permanently store all the visited states, we have to recompute some of them if required. This could in the worst case result in the theoretically not very significant increase of time complexity from quadratic to cubic in the size of the state space. Moreover, for SPIN with its very fast time performance the space (memory) is the real critical resource limitation in practice.

6 Conclusions and Future Research

We have proposed an extension to the existing distributed algorithm used in the verification checker SPIN, which allows to model-check LTL formulas. This problem was suggested in [LS99] and left unsolved. The method used is very simple and requires some synchronisation between nodes.

The experimental version of the algorithm has been implemented and a series of preliminary experiments has been performed on a cluster of nine 366 MHz Pentium PC Linux workstations with 128 Mbytes of RAM each interconnected with a fast 100Mbps Ethernet and using MPI library.

We have compared the performance of our algorithm with the standard sequential version of SPIN running on a single computer. Although the implementation of the *Dependency structure* was far from being optimal, we were able to receive some promising results. Our results show that it is possible to increase the capability of SPIN to check LTL formulas in a distributed manner. For testing we have used some of the scalable problems from the SPIN distribution. The experimental version was mainly used to get a fast feedback on applicability of the method. We intend to perform an extensive testing on a variety of different examples using the new and more sophisticated implementation currently under development.

There are several problems we intend to consider in the future. Firstly, we have used MPI library to get a fast prototype implementation. However, the overhead caused by this communication infrastructure is quite high. We assume that using another communication infrastructure (like TCP/IP) will certainly lead to better performance. Secondly, our algorithm utilises a partition function which is based on the function used by distributed SPIN. We realize that partition function plays a crucial role in the effectiveness of the algorithm and so we want to investigate other techniques that might be better suited to model-checking LTL formulas. Finally, for simplicity reasons we allow only one nested DFS procedure to be performed at a time. Under certain circumstances, which can be effectively recognised from the dependency structure, it is possible to start more than one nested DFS procedure at a time and we want to extend our distributed algorithm by partial parallelisation of nested DFS procedures.

References

[AAC87] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56, Berlin, Germany, August 1987. Springer-Verlag.

[BBS00] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. Technical Report FI-MU-10/00, Masaryk Univeristy Brno, 2000.

[BDHGS00] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *third International Conference on Formal methods in Computer-Aided Design (FMCAD'00), Austin, Texas*, November 2000.

[Dil96] David L. Dill. The murφ verification system. In *Conference on Computer-Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 390–393. Springer-Verlag, July 1996.

[HGGS00] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In Orna Grumberg, editor, *Computer Aided Verification, 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, June 2000.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *SPIN workshop*, number 1680 in LNCS, Berlin, 1999. Springer.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.

[UD97] U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267, Berlin, Germany, 1997. Springer.

[WL93] P. Wopler and D. Leroy. Reliable hashing without collision detection. In *Conference on Computer-Aided Verification (CAV '93)*, Lecture Notes in Computer Science, pages 59–70. Springer-Verlag, 1993.