Distributed Generation of State Space for Timed Petri Nets

I. Rada  
Toronto Software Laboratory  
IBM Canada Ltd.  
Toronto, Canada M3C 1H7

W.M. Zuberek  
Department of Computer Science  
Memorial University of Newfoundland  
St. John's, Canada A1B 3X5

Keywords: Distributed computing, timed Petri nets, state space generation, speedup analysis.

Abstract
A cluster of PC's is used for the generation of state space for timed Petri nets. Disjoint regions of the state graph are generated on different machines. On each machine, the communication is separated from the computation part, and is performed by two concurrent processes: one receiving, and one sending messages to other machines. The implementation is based on PVM (Parallel Virtual Machine) using a modified version of TPN-tools. Experiments performed on a cluster of 32 PC’s show almost linear speedup for some classes of timed Petri nets.

1. INTRODUCTION

Development of complex, real-world systems is usually preceded by detailed studies conducted on formal models of such systems. Formal, mathematical models are used for the verification of system's properties and for the derivation of its performance characteristics [11].

For systems which exhibit concurrent activities, Petri nets are a popular choice of the modeling formalism because of their ability to express concurrency, synchronization, precedence constraints and nondeterminism. Moreover, Petri nets “with time” (stochastic or timed) include the durations of modeled activities and this allows to study performance aspects of modeled system [1, 13, 22].

Three basic approaches to the analysis of Petri net models are known as structural analysis, reachability analysis and, for time-enhanced nets, discrete-event simulation [16, 23]. Structural methods predict the properties of net models on the basis of their structure (i.e., connections between elements). Structural analysis is usually rather simple, but it can be applied only to nets with special properties. Net simulation is based on the fact that a (timed or stochastic) Petri net is a discrete-event system, with events corresponding to the firings (or occurrences) of net transitions. Simulation can be applied to a very large class of nets, but may not capture events which occur very rarely.

Reachability analysis is the preferred method when a detailed analysis of the model’s behavior is needed. The exhaustive generation of all model’s states and transitions between the states allows to answer questions about liveness, boundedness, persistence, absence of deadlocks, etc. [13, 16]. In reachability analysis, the states of the net and all transitions between the states are organized in a directed graph called the reachability graph. For timed and stochastic Petri nets (with deterministic or exponentially distributed firing times), this graph is a Markov chain, whose stationary probabilities can be determined by known methods [18]. These stationary probabilities are used to derive many performance characteristics of the analyzed model [1, 22].

For large net models, the state space can be very large, and can easily exceed the resources of a single computer system. There are two basic methods to cope with this problem [6]: avoidance methods, which use net properties to obtain a smaller state space, and tolerance methods, which accept that the state space is large and use various techniques (in particular, parallel and distributed algorithms) to generate it. The availability of clusters of workstations and portable libraries for distributed computing makes the second approach very attractive; the state space can be constructed using a cluster of communicating processors.

While there have been several papers published on distributed generation of state spaces of systems [14, 15] and on parallel and distributed state space generation for stochastic Petri nets [3, 2, 4, 5, 12], very little information is available on distributed analysis of timed Petri nets [17].

This paper describes a distributed algorithm for the generation of state space for timed Petri nets. The algorithm has been implemented in C++ using the TPN-tools [23], and STL [20] and PVM [10] libraries. It has been tested on a network of PC's and workstations in the Department of Computer Science, Memorial University of Newfoundland. Experimental results show almost linear speedup for some classes of timed Petri nets.

Section 2 recalls the sequential algorithm for state space generation. The proposed distributed algo-
2. STATE SPACE GENERATION

A typical algorithm for the sequential generation of the state graph of a (bounded) net is given below [21]; there are several variations of this algorithm, but the differences are rather insignificant (e.g., a stack can be used instead of the queue [4]).

1. algorithm sequential state graph generation:
2. var \( m_0 \); (* initial marking *)
3. States := \( \emptyset \); (* set of states *)
4. Arcs := \( \emptyset \); (* set of arcs *)
5. unexplored := \( \emptyset \); (* queue of unexplored states *)
6. search_set := \( \emptyset \); (* search tree *)
7. begin
8. for each state in initialStates(\( m_0 \)) do
9. States := States \cup \{state\};
10. insert(search_set, state);
11. insert(unexplored, state);
12. endfor;
13. while nonempty(unexplored) do
14. state := remove(unexplored);
15. for each s in successors(state) do
16. if s \( \notin \) search_set then
17. States := States \cup \{s\};
18. insert(unexplored, s);
19. insert(search_set, s);
20. endif;
21. Arcs := Arcs \cup \{(state.id, s, s, s.prob)\}
22. endfor;
23. endwhile;
24. end.

This algorithm constructs the state graph \( G = (\text{States}, \text{Arcs}) \) for a timed Petri net with an initial marking \( m_0 \). It uses a queue, unexplored, for the unexplored states, and an auxiliary data structure, search_set, for efficient checking if a state has already been generated. The function initialStates(\( m_0 \)) returns a set of initial states corresponding to the initial marking \( m_0 \), and the function successors(s) returns the set of states directly reachable from a state \( s \).

For unbounded nets, a solution for a special class of stochastic Petri nets (with exactly one unbounded place) is given in [9]. Other approaches are based on coverability graphs [19].

3. DISTRIBUTED GENERATION

The state graph of a timed Petri net can be generated by partitioning the (yet unknown) state space into \( n \) disjoint regions, \( R_1, R_2, \ldots, R_n \), which are constructed independently, and then integrated in one state graph if needed. The construction of these regions can be performed by \( n \) identical processes running concurrently on different machines. The entire distributed generation has three phases:

1. the initialization phase during which the system is set up by creating the cooperating processes and exchanging the information necessary for inter-process communication.
2. the computational phase during which the regions of the state space are constructed.
3. the (optional) integration phase during which all the states and arcs of the regions are collected, and integrated into the complete state graph.

Fig.3.1. Distributed generation system.

Three types of processes are used (as shown in Figure 3.1): a process starting the distributed system and initiating the computations, called Spawner; several processes constructing the region graphs, called Generators, and a process collecting and integrating the results, called Collector. Technically, the Collector can be the same process as the Spawner, because they exist in disjoint periods of time; the distinction between them is made here for clarity only.

Physical processes corresponding to these logical processes constitute a "virtual machine." This virtual machine runs on a cluster of computers.

The partitioning of the state space is defined by a partitioning function, which determines, for each state, the region to which it belongs:

\[
\text{region}(s) = \left( \sum_{i=0}^{P} a_i m(p_i) + \sum_{i=0}^{P} \beta_i f(t_i) \right) \mod (n).
\]

where the coefficients \( a_i \) and \( \beta_i \) are integer numbers and \( m \) and \( f \) are the marking and firing components of the state \( s \) [22]. The same partitioning function is used by all processes of the virtual machine.

3.1. State space generation

The distributed state space generation starts with the execution of the Spawner which creates the Collector and spawns \( n \) Generators on the other hosts.
providing them with the addresses of itself and of the Collector; so that they can exchange messages. Each Generator sends its address back to the Spawner as soon as it is ready. The Spawner collects all addresses into a process table, and broadcasts it back to all Generators.

A state \( s \), created by Generator \( r \), is local if region \( (r) = i \) and it is non-local otherwise. A state \( s \) is external to Generator \( r \) if region \( (r) = i \) but \( s \) has been created by another Generator. A non-local state can be generated many times by Generators. Non-local states generated for the first-time are called first-time non-local states, and the corresponding cross-arcs directed to them are called first-time cross-arcs.

Each Generator sends all non-local states with their cross-arcs to the appropriate Generators. When an external state does not already exist in the region of the destination processor, it is inserted there and then processed. External cross-arcs are treated differently because the insertion of cross-arcs into their appropriate regions is not critical for the state space generation; they are sent to their regions when all the states have been generated in all regions, reducing the communication during the state generation phase.

### 3.2. Termination detection

When a Generator finishes processing its states, it waits for states from other processes. In order to prevent a deadlock situation in which all Generators are idle and wait for each other, a global termination detection algorithm is interleaved with the computation. This termination algorithm checks if all processors have finished their computations.

Global termination detection is a classical problem in distributed computing. The algorithm used here is the one proposed by Dijkstra et al. [8]. It assumes that the cluster of processors has a ring topology. \( P_1 \rightarrow P_2 \rightarrow P_3 \cdots P_{n-1} \rightarrow P_n \), \( P_n \rightarrow P_1 \), in which a token is transmitted from one processor to another checking if all processors have terminated their tasks. The token uses two colors, Black and White, to represent two states of the distributed system: the White color corresponds to the situation when all processors are found idle; the Black color corresponds to the situation where some activity existed prior to the moment of checking and, therefore, it cannot be concluded that the system is idle. A processor indicates that its state as idle or active by making its color White or Black, respectively. Whenever a processor induces activity in the system by sending a data message, it also sets its color to Black.

### 3.3. Processes and process interactions

In order to perform computation tasks concurrently with communication, each Generator is composed of three processes (Figure 3.2): the Worker, responsible for the generation of the state space, the Sender, responsible for sending messages to other processes, and the Receiver, responsible for receiving messages from other processes and for the termination detection. When the Spawner creates the Generators, it actually creates Worker processes; as its first step, each Worker creates its Receiver and Sender processes.

![Fig.3.2. The structure of a Generator.](image)

The Worker, Receiver, and Sender of each Generator reside on the same processor. Their communication is based on shared variables. Each Worker communicates with its Receiver using a shared memory segment `recv_buffer`, and with its Sender using a shared memory segment `send_buffer`. The mutual exclusion for accesses to the buffer is controlled by semaphores in a typical producer-consumer scheme. The Sender and the Receiver share the semaphore `sender_ready`. In the initialization phase, the Receiver waits at this semaphore until the Sender performs the signal operation. This ensures that when the Receivers indicate to the Spawner to broadcast the process table, the Senders are ready to receive it.

Also, the machine color (for checking the termination) needs to be accessed by both the Worker and the Receiver. Therefore, it is stored in a memory segment shared by these two processes.

System components residing on different processors communicate by message passing using the PVM (Parallel Virtual Machine) package [10]. Different types of messages are distinguished by marking them with distinct tags.

### 3.4. Algorithms

Some of the presented algorithms use the generic semaphore operations `wait(semaphore)` and `signal(semaphore)`.

The syntax used for receiving messages is `message := receive`; if a processor expects a specific type of messages, then `receive` contains the type of messages as its argument; e.g., in line 21 below, an EXIST message is expected. Similarly, messages are
sent by a send operation which has a process identifier and the information (or a message type) as arguments. Broadcast messages are sent by the broadcast operation. The information exchanged through the shared buffers is sent to the buffer by put(buffer, info) operations, and retrieved from a buffer by get(buffer) function invocations.

The Spawner

The Spawner performs initialization tasks:

1. algorithm Spawner:
2. var host_file; (* virtual machine conf. filename *)
3. net_file; (* Petri net description filename *)
4. net; (* internal net representation *)
5. hosts]; (* virtual machine hosts *)
6. n; (* number of processors *)
7. init_states; (* queue of initial states *)
8. id; (* identifier *)
9. init_id; (* identifier of the initiator *)
10. tid; (* task identifier *)
11. tid_coll; (* Collector’s identifier *)
12. generators; (* all Senders and Receivers *)
13. begin
14. (n, hosts] := read(host_file);
15. net := read(net_file);
16. init_states := initial_states(net);
17. init_id := region(first(init_states));
18. tid_coll := spawn Collector(n) on host_name[0];
19. for i := 1 to n do
20. spawn Worker (init_id, tid_coll, my_tid, net, i, n) on hosts[i];
21. (id, tid) := receive(EXIST);
22. process_table[id] := tid
23. endfor;
24. broadcast(generators, INFO(process_table));
25. while nonempty(init_states) do
26. state := remove(init_states);
27. send[process_table[region(state)],
DATA(state, 0, state.prob)]
28. endwhile
29. end.

The Spawner reads the configuration of the virtual machine and the description of the net model from input files (lines 14,15). Then it determines the initial states of the net (line 16) and establishes the termination detection initiator as the Generator processing the first initial state (line 17).

The Spawner then creates the Collector and n Workers (which are provided with initiator’s identifier, the Spawner’s and Collector’s addresses, and the total number of Generators).

The Spawner collects all the addresses into a process table (line 22) and broadcasts it back (line 24) to all Generators (actually, their Senders and Receivers). Since the Senders need to know where to send their non-local states, and the Receivers need to know their successors in the processor ring used for termination detection, all Senders and all Receivers join a PVM process group called “generators” (line 9 of Sender and line 12 of Receiver).

When the entire virtual machine is set up, the Spawner initiates the space state generation by sending the initial states of the net to their corresponding Generators (lines 25-28). As in [23], a dummy state, with id 0, is used as a common parent of all initial states. Therefore, each initial state is packed into a DATA message containing the state representation, its parent id (i.e., 0) and the corresponding transition probability.

The Worker

Each process Workeri constructs the region Ri = (Statesi, Arcsi) of the state graph. The set of generated states, Statesi, is implemented using the C++ standard class set [20], which guarantees logarithmic search time. Each element of this set is a tuple (state, id) containing a state description and the state id. The set of arcs is a linked list of triples (id1, id2, prob), where id1 and id2 are state identifiers and prob is the state transition probability.

The outline of each Worker is as follows:

1. algorithm Worker; (init_id, tid_coll, tid spun, net, n);
2. var machine_color shared with Receiveri;
3. send_buffer shared with Senderi;
4. recv_buffer shared with Receiveri;
5. Statesi := ∅; Arcsi := ∅; (* states and arcs *)
6. unexplored := ∅; (* queue of states *)
7. id; parent_id; parenti; (* identifiers *)
8. probi; (* arc probability *)
9. cont := true; (* continuation flag *)
10. begin
11. if i = init_id then
12. spawn InitiatorReceiver(tid_coll, tid spun, n) on this_host
13. else
14. spawn OrdinaryReceiver(tid_coll, tid spun, n) on this_host
15. endif;
16. spawn Senderi(n) on this_host;
17. execute mainloop; (* loop is shown below *)
18. cont := true;
19. while cont do
20. (state, (parent, prob)i−1..k) :=
get(recv_buffer);
21. if state = null then
22. cont := false
23. else
24. id := findstate.id(Statesi, state);
25. Arcsi := Arcsi ∪ { (parent, id, prob)i−1..k }
26. endif
27. endwhile;
28. send(tid роли, States_); 
29. send(tid роли, Arcs_); 
30. send(tid роли, FINISHED) 
31. end.

First, each Worker_i creates its Receiver_i (Initiator/Receiver or Ordinary Receiver) and its Sender_i (lines 11-16), on the same host (this host). Then it enters the main loop in which all reachable states and their descendants are determined. This loop is described below.

When the main loop is finished, all Workers have generated all the states in their regions, but they need to collect the remaining arcs leading to their region (lines 18 to 27). When a non-null item \((state, \langle parent_id, prob \rangle)\) is retrieved, the id of the state is looked up in the region, and the arcs are inserted in the list of arcs. This loop continues until a null item is retrieved from the receive buffer (lines 21 and 22).

After completing its region, the Worker sends the states and arcs, clustered in just two messages, to the Collector. A final message FINISHED notifies the Collector that the entire region has been sent.

The Worker deals with states generated locally and states received from other processes. It maintains two queues, unexplored, a queue for local unexplored states, and recv_buffer, a queue for external states.

The main loop is as follows:

1. mainLoop: 
2. while cont do 
3. if empty(recv_buffer) \&\& nonempty(unexplored) then 
4. state := remove(unexplored); 
5. new := true 
6. else 
7. \((state, parent_id, prob) := get(recv_buffer)\); 
8. if state = null then 
9. cont := false 
10. else 
11. new := not state_exists(States_i, state); 
12. if new then 
13. state.id := create_local_id; 
14. States_i := States_i \cup \{state\} 
15. end; 
16. Arcs_i := Arcs_i \cup \{(parent_id, state.id, prob)\} 
17. end; 
18. end; 
19. if cont \&\& new then 
20. for each s in successors(state) do 
21. if region(s) = i then 
22. if state_exists(States_i, s) then 
23. Arcs_i := Arcs_i \cup \{(state.id, s.id, s.prob)\} 
24. else 
25. state.id := create_local_id; 
26. States_i := States_i \cup \{s\}; 
27. Arcs_i := Arcs_i \cup \{(state.id, s.id, s.prob)\}; 
28. insert(unexplored, s) 
29. endif 
30. else 
31. processor.color := Black; 
32. put(send_buffer, \{s, state.id, s.prob\}) 
33. end; 
34. endfor 
35. endif 
36. endwhile;

If recv_buffer is empty but the working queue is not empty, the state to process is taken from the working queue (line 4); otherwise, the function get is called to get an item from recv_buffer (line 7), which will involve waiting if the buffer is empty. This loop is discontinued when a null item is retrieved from the buffer, i.e., when the function get returns a null state (line 9); the Receiver puts this null item in the buffer after receiving the message that the global termination has been detected.

When a non-null datum is retrieved from the buffer, the Worker checks whether the state included in the datum already exists in the region by calling the function state_exists (line 11). If the state does not exist (i.e., the function state_exists returns false), the state must be inserted in the graph with a new, unique id (lines 13, 14). Whether the state exists or not, the corresponding arc is inserted in the set of arcs (line 16).

For each new state, the Worker creates the successor states (line 20) with their probabilities, and determines whether they are local or not (line 21). Local successor states are then processed as in the sequential algorithm; non-local states are deposited into send_buffer (line 32), from where they will be subsequently extracted and sent away by the Sender.

The Worker also changes the color of the Generator to Black (line 31) to indicate that the processing of states is not finished yet.

### The Sender

The Sender is mainly responsible for sending the non-local states and cross-arcs to their corresponding Generators.

1. algorithm $\text{Sender}(n)$; 
2. var recv_buffer shared with Worker_i; 
3. sender_ready semaphore shared with Receiver_i; 
4. state; (* current state *) 
5. sent_states; (* states already sent *) 
6. process_id; (* process addresses *) 
7. cont := true; (* continuation flag *) 
8. begin 
9. join_group(generator_i); 
10. end.
10. signal(sender.ready);
11. processable := receive(INFO);
12. while cont do
13.  (state, parent_id, prob) := get(send.buffer);
14.  if state = null then
15.    send.rest(processable, sent.states, cluster.size);
16.    cont := false
17.  else
18.    if state ∉ sent.states then
19.      send(processable[region(state)],
20.         DATA(state, parent_id, prob));
21.      insert(sent.states, state)
22.    else
23.      add.arc.link(state, parent_id, prob)
24.    endif
25.  endwhile;
26. broadcast(generators, ARCS_EXPORT_DONE)
27. end.

First, the Sender, joins the group of Generators (line 9) in order to receive the table of process identifiers from the Spawner, after notifying the Receiver, that it is ready (line 10); the received table is stored in the process.table (line 11).

Each non-null triple (state, parent_id, prob) retrieved from the buffer is processed by the Sender as follows: if state is a first-time non-local state, then it is stored in a local structure state.sent, and the triple is sent to the appropriate Generator (lines 19, 20). If state has been sent already, then the arc information is attached to the local copy of the state (line 22).

In order to minimize the search time, the structure state.sent is organized as a search tree, using the state representation as the key.

The Sender's loop is terminated when the function get returns a null state, as a consequence of the global termination detection. The Sender then sends the remaining arcs, together with their states (states are needed to determine their ids at the destination), by calling the function send.rest (line 15).

After sending this data, the Sender broadcasts a message ARCS_EXPORT_DONE and terminates.

The Receiver

All Receivers except the one used for initiating termination detection implement the following algorithm.

1. algorithm OrdinaryReceiver (tid, coll, tid.spawn, n);
2. var processor_color shared with Worker;
3. rec.buffer shared with Worker;
4. sender.ready semaphore shared with Sender;
5. message; (* message received *)
6. n; (* number of processors *)
7. processtable[]; (* process addresses *)
8. k := 0; (* termination counter *)
9. begin
10.  wait(sender.ready);
11.  join(group(generators));
12.  send(tid.spawn, EXIST(i, my.Tid));
13.  processable := receive(INFO);
14.  while k < n - 1 do
15.    message := receive;
16.    case message-type of
17.      DATA:
18.        put(rec.buffer, message.s);
19.      CHECK_TERM:
20.        if Worker.idle ∧ Sender.idle then
21.          if processor.color = Black then
22.            message.token := Black
23.            endif;
24.            send(Receiver, message.token);
25.            processor.color := White
26.        endif;
27.      TERMINATE:
28.        begin
29.          interrupt(Sender);
30.          put(rec.buffer, null)
31.        end;
32.      ARCS_EXPORT_DONE:
33.        begin
34.          k := k + 1;
35.          if k = n - 1 then
36.            put(rec.buffer, null)
37.        endif
38.      end
39.  endcase
40. endwhile
41. end.

The initialization part is common for all Receivers; before sending an EXIST message to the Spawner, the Receiver waits for its Sender to be ready (the Sender should be able to receive messages when the process table is sent to it by the Spawner). A semaphore sender.ready is used for this purpose; the Receiver waits at this semaphore until the Sender performs the signal operation, and then it sends, to the Spawner, an EXIST message containing its id and the Receiver's address (line 12). After this the Receiver gets the process table, and stores it in the process.table.

The Receiver reacts to different messages in different ways. For DATA messages, which are sequences of state descriptions with their incoming arcs (state, (parent_id, prob_i) = 1, ..., n), the enclosed k descriptions are extracted from the message and stored in rec.buffer.

In the case of a CHECK_TERM message the Receiver checks whether its processes are idle by calling the logical functions Worker.is.idle and Sender.is.idle. The Worker is idle when it is waiting with empty rec.buffer; i.e., it is suspended on
the semaphore not-empty. Similarly, the Sender is idle when it is suspended on a semaphore not-empty.

If the processes are idle, the Receiver propagates the token (line 25), changing the token’s color if the processor’s color is Black (line 23). Receiver in the ring: \(i \oplus 1 = i + 1\) if \(i < n\) and \(i \oplus 1 = 0\) if \(i = n\).

For a TERMINATE message, the Receiver must end the Worker’s loop and cause the Sender to send the remaining arcs. Therefore, the Receiver adds a special null item to rec Maul buffer (line 31). Upon retrieving this item, the Worker ends processing the states. The Receiver also discontinues its Sender’s loop (line 30). Finally, the Worker receives all the remaining cross-arcs as DATA messages.

All ARCSEXPORT_DONE messages are counted (line 35). After \(n - 1\) such messages, the Receiver does not expect any subsequent data (PVM ensures that the order of sent messages is preserved at the receiver side), so it puts a null item into rec Maul buffer (line 36). When the Worker retrieves this message, it terminates its operation.

4. EXAMPLES

This section presents some experimental results obtained for nets with a fairly large state space sizes. Experiments were conducted for D-timed as well as M-timed nets because it was anticipated that the performance results for these two classes of nets can be quite different; usually D-timed nets generate less arcs (i.e., transitions between states) than comparable M-timed nets, but are more computationally-demanding for state processing than M-timed nets. Consequently, the computation-to-communication ratios for these two classes of nets are quite different.

The performance measure used in the experiments is the speedup of the program. The speedup \(S\) of a distributed program is a function \(S: \{1, 2, 3, \ldots\} \rightarrow \mathbb{R}^+\) defined as the ratio of the program’s execution time on one processor, \(T(1)\), to the program’s execution time on \(n\) processors, \(T(n)\):

\[
S(n) = \frac{T(1)}{T(n)}.
\]

The program’s speedup is influenced by the partitioning function, and in particular by its locality. The locality of a partitioning function is the (arithmetic) average of processor’s localities, and the locality of each processor is the ratio between its number of local states and its region size.

All experiments have been performed on a cluster of 32 diskless Linux PCs, each with 128MB RAM, connected via a 100 Mbps Ethernet.

4.1. D-timed nets

The example is taken from [7]: the net models a parallel MIMD architecture.

Two initial markings were used. For the first marking, the state graph has 14487 states and 26675 arcs (Example 1(a)); while for the second marking it has roughly three times as many states (46729) and more than three times as many arcs (92253) (Example 1(b)).

Figures 4.1 and 4.2 show the speedup for these two cases, for the range of processors from 1 to 32.

![Fig. 4.1. Speedup for Example 1(a).](image1)

![Fig. 4.2. Speedup for Example 1(b).](image2)

It can be observed that the character of the speedup curves for the two cases is very similar, however, the speedup is better for the larger state graphs.

4.2. M-timed nets

This example is also taken from [7]; it is another net model of a parallel MIMD architecture.

The state graph has 27399 states and approximately seven times more arcs (197337).
The speedup obtained in this example is shown in Figure 4.3. It can be observed that the speedup is rather modest in this case (close to 2 for 25 processors), and it changes very slowly with the number of processors.

4.3. Discussion

The experiments indicate that the program's execution time depends on: (1) the average time needed for processing a single state, (2) the average number of arcs per state, and (3) the locality of the partitioning function.

If the average processing time per state is sufficiently long, the computation time dominates the (total) execution time, and this time is significantly reduced with each additional processor added to the virtual machine.

If the average number of arcs per state is large, the number of messages in the system grows with each additional processor, thus affecting processor's computation time (the searching of external states), and its waiting time.

If the locality decreases with additional processors, the number of messages in the system increases, and, on average, each processor's waiting time increases.

For nets with large number of arcs per state and small state processing time, the communication time seems to dominate the execution time, so only moderate speedups can be obtained. Faster communication medium would improve the results in such cases.

Distributed processing is quite efficient for nets with high state processing time and small number of arcs per state (like D-timed nets).

5. CONCLUDING REMARKS

This paper proposes a distributed algorithm for the generation of state space for timed Petri nets. The algorithm is built on top of the software package TPNTools [23]. Similar research has been conducted for stochastic Petri nets but there are significant differences between these two classes of nets with respect to states and state transitions, which result in rather different models of the same systems. The algorithm presented in this paper differs from others in the following aspects:

- The proposed algorithm totally separates the computational aspect from the communication, by the use of the three concurrent processes running on each machine of the cluster.

- Each Generator gives priority to external states over local ones preventing redundant incoming states from accumulating in the memory. All other algorithms give priority to local states. They wait to receive external states only after the available local states have been processed.

- The construction of the state graph is performed in two consecutive stages: during the first stage, all states are sent to their regions (i.e., processors), but cross-arcs directed to already created non-local states are not sent, but stored locally. In the second stage, all stored cross-arcs are sent in one message. This delayed sending has two consequences: (1) it reduces the traffic in the network, so that the external states which are needed to complete the construction of the region, can be transferred with reduced delays, improving the performance, and (2) no matter how many cross-arcs a processor creates for one non-local state, all arcs will be sent in at most two messages.

Experimental results suggest that the performance of the algorithm is influenced by both the structure of the model (the average state processing time, the average number of successors per state), and the choice of the partitioning function (which establishes the number of cross-arcs).

For the class of D-timed Petri nets with a high average state processing time and a small average number of arcs per state, the distributed implementation gives almost linear speedup. The algorithm allows the generation of relatively large state spaces (an order of $10^6$ states) in very reasonable run times. For this class of nets, a "better" behavior can be noted when increasing the state space size, which can be attributed to large and more uniformly distributed numbers of states assigned to each of the processors.

Finally, the developed distributed implementation of the state space generation can be used as a starting point for a distributed steady state solution of
the state graph a “natural” next step in quantitative
analysis of timed net models.

Acknowledgement

The Natural Sciences and Engineering Research
Council of Canada partially supported this research
through grant RGPIN-8222.

This work has been done when I. Rada was at
Memorial University of Nfld.

REFERENCES

[1] M. Ajmone Marsan, G. Balbo, and G. Conte,
nets for the performance evaluation of systems”; 
ACM Transactions on Computer Systems vol. 2,
no. 2, pp. 93–122.

Parallel graph generation algorithms for shared
and distributed memory machines”; in: Parallel
Computing: Fundamentals, Applications and New

shared-memory state-space exploration in
stochastic modeling”; Solving Irregularly Structured
Problems in Parallel (LNCS 1253), pp. 207–218.
Springer-Verlag.

1997. “State space construction and steady
state solution of GPSN on a shared memory
multiprocessor”; Proc. IEEE Int. Workshop
Petri Nets and Performance Models (PNPM’97),
pp. 112–121.

approaches to the numerical transient analysis
of stochastic reward nets”; in: Application
and Theory of Petri Nets 1999 (LNCS 1639),

Petri nets”; Proc. of the IEEE Conf. on
Petri Nets and Performance Models (PNPM’95),
pp. 2–9.

concurrent architectures with mesh topology”;
Fourth Int. Workshop on Petri Nets and Performance
Models (PNPM’91), pp. 280–289.

[8] E. Dijkstra, W. Feijen, and A. van Gasteren,
1983. “Derivation of a termination detection algo-
rithm for distributed computations”; Information

bounded stochastic Petri nets: ergodic criteria
and steady-state solutions”; Journal of Systems
and Software, vol. 6, no. 1-2, pp. 103–118.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang,
R. Manchek, and V. Sunderam. 1994. PVM: Par-
allel Virtual Machine. A Users’ Guide and Tu-
torial for Networked Parallel Computing MIT
Press.


“Analysis of large GSPN models: a distribu-
ted solution tool”; Proc. IEEE Int. Workshop
Petri Nets and Performance Models (PNPM’97),
pp. 122–131.

sis, and applications”; Proceedings of the IEEE,
vol. 77, no. 4, pp. 541–580.

parallelization of discrete state-space genera-
tion”; Journal of Parallel and Distributed Com-

space generation of discrete state stochastic
models”; INFORMS Journal on Computing vol. 10,
no. 1, pp. 82–92.

eling of Systems. Prentice Hall.

space for timed Petri nets”; M.Sc. Thesis, De-
partment of Computer Science, Memorial Uni-
versity of Newfoundland, St. John’s, Canada A1B 3X5.

Solution of Markov Chains. Princeton University
Press.

graphs for a class of synchronously executed un-
bounded Petri nets”; Journal of Parallel and
Distributed Computing vol. 10, no. 4, pp. 253–
260.

Language (3rd ed.). Addison-Wesley.

space for timed Petri nets”; Proc. of ACM 16th
Annual Computer Science Conference, pp. 239–
248.

tions, properties, and applications”; Microelec-
tronics and Reliability, vol. 31, no. 4, pp. 627–
644.

nets: basic concepts and properties” (Lecture
notes for the course “Modeling and Analysis of
01. Department of Computer Science, Memorial
University of Newfoundland, St. John’s, Canada
A1B 3X5.