34-th Annual Simulation Symposium, Seattle, WA, April 22-26, 2001, pp.93-98.
Copyright © 2001 IEEE (DOI 10.1109/SIMSYM.2001.922120).

Modeling and Analysis of Distributed State Space Generation for Timed Petri Nets

W.M. Zuberek
Department of Computer Science
Memorial University of Nfld
St.John's, Canada A1B 3X5
wlodek@cs.mun.ca

I. Rada
Toronto Software Laboratory
IBM Canada Ltd.
Toronto, Canada M3C 1H7
irinar@ca.ibm.com

Abstract

The performance of distributed generation of the state space for timed Petri nets is rather sensitive to the type of analyzed nets. In order to analyze the performance of such an application, the distributed generation is represented by a timed Petri net and the behavior of this net is studied, using simulation techniques, for different combinations of modeling parameters.

1. Introduction

Implementation of complex, real-world systems is usually preceded by extensive studies of their formal models. For systems which exhibit concurrent activities, Petri nets are a popular choice of 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 the performance aspects of the modeled system [1, 7, 12].

Three basic approaches to the analysis of Petri net models are known as structural analysis, reachability analysis and discrete-event simulation [8, 13]. 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. Reachability analysis is based on an exhaustive generation of all reachable states; reachability analysis is the most suitable method when a detailed analysis of the model's behavior is needed [7, 8]. Net simulation is based on the fact that a (timed or stochastic) Petri net is a discrete event system, where the events correspond to the firings (or occurrences) of net transitions; simulation can be applied to a large class of temporal nets, but may be unsuccessful or very inefficient in capturing events which occur rarely.

In reachability analysis, the states of the model and the transitions between states are organized in the reachability graph which is used for verifying the required qualitative properties (such as absence of deadlocks or liveness). For timed and stochastic Petri nets, this graph is a Markov chain, whose stationary probabilities can be determined using known numerical methods [10]; these stationary probabilities are used to derive many performance measures of the model [1, 2, 12].

For large net models, the state space can easily exceed the resources of a single computer system. The availability of clusters of workstations and portable message passing libraries makes distributed generation of the state space an attractive possibility.

The purpose of this paper is to investigate the significant differences in the speedup that have been observed [9] during distributed analysis of different timed Petri nets; in some cases the speedup curve is an almost linear function of the number of processors, n_p, as illustrated in Fig.1.1; in other cases, this curve reaches saturation and even decreases for large number of processors, as shown in Fig.1.2.

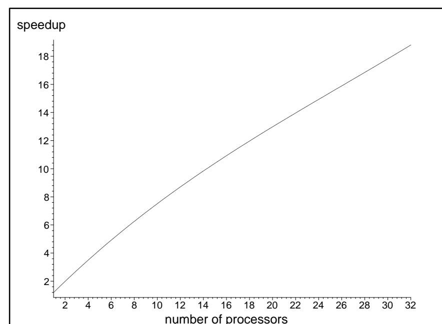

Fig.1.1. Speedup as a function of n_p; case 1.

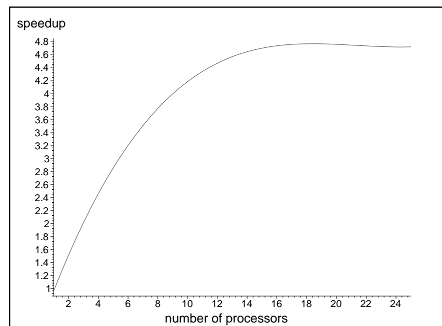

Fig.1.2. Speedup as a function of n_p; case 2.

Section 2 recalls the basic sequential algorithm for reachability analysis. Section 3 presents its distributed version, that was used to generate the results shown in Fig.1.1 and Fig.1.2. A timed Petri net model of a cluster of processors connected by a switch is presented in Section 4, Section 5 discusses the results obtained from the presented model, and Section 6 concludes the paper.

2. Sequential generation

A typical algorithm for the sequential generation of the state graph of a (bounded) net is given below [11]; there are several variations of this algorithm, but the differences are rather insignificant.

```
1. Sequential_state_graph_generation:
2. var m0;                          (* initial marking *)
3.     States := ∅;                 (* set of states *)
4.     Arcs := ∅;                   (* set of arcs *)
5.     unexplored := ∅;             (* queue of states *)
6. begin
7.     for each state in initial_states(m0) do
8.         States := States ∪ {state};
9.         insert(unexplored, state)
10.    endwhile;
11.    while nonempty(unexplored) do
12.        state := remove(unexplored);
13.        for each s in successors(state) do
14.            if not found(States, s) then
15.                States := States ∪ {s};
16.                insert(unexplored, s)
17.            endif;
18.            Arcs := Arcs ∪ {(state.id, s.id, s.prob)}
19.        endfor
20.    endwhile
21. end.
```

This algorithm constructs the state graph $G = (States, Arcs)$ for a timed Petri net with an initial marking m_0. It uses a queue, *unexplored*, for the unexplored states. The function $initial_states(m_0)$ returns the set of initial states corresponding to the initial marking m_0, the function $successors(s)$ returns the set of states directly reachable from the current state s, and the logical function *found(States,s)* returns **true** if the state s exists in the set of states *States*.

3. Distributed generation

In distributed generation of the state graph, the (yet unknown) state space is partitioned into n disjoint regions, $R_1, R_2, ..., R_n$, and these regions are constructed independently by n identical processes running concurrently on different machines. At the end, the regions can be integrated in one state graph if needed.

The disjoint regions of state graphs are determined by a partitioning function, which maps each state into the region to which it belongs. This partitioning function is similar to the one used in [6], but, for timed nets, it also takes into account the firing transitions:

$$region(s) = [\sum_{i=0}^{|P|} \alpha_i \ m(p_i) + \sum_{i=0}^{|T|} \beta_i \ f(t_i)] \bmod (n)$$

where $|P|$ is the cardinality of the set of places P, $|T|$ is the cardinality of the set of transitions T, the coefficients α_i and β_i are integer numbers, and m and f are marking and firing components of a state s [12].

Distributed generation of the state space can be performed by three kinds of (logical) processes [9]: a process starting the distributed system and initiating the computations, called *Spawner*; several processes constructing the regions of the state space, 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 distributed generation starts with the execution of the *Spawner* which creates the *Collector* and spawns n *Generators* on the other hosts; it also organizes the addresses of all created processes in a *proc_table*, which it broadcasts to all processes so they can send messages to each other:

```
1. Spawner:
2. var m0;                    (* initial marking *)
3.     n;                     (* the number of hosts *)
4.     proc_table[];          (* processor identifiers *)
5. begin
6.     input virtual machine and model descriptions;
7.     spawn Collector on this_host;
8.     for i := 1 to n do
9.         proc_table[i] :=spawn Generator on host[i]
10.    endfor;
11.    broadcast(proc_table);
12.    for each s in initial_states(m0) do
13.        send(proc_table[region(s)], ⟨s, 0, s.prob⟩)
14.    endfor
15. end.
```

Each $Generator_i$ determines all successors for each state belonging to region R_i. A successor state can be in the same region (in which case it is called an internal state and the connecting arc is an *internal arc*) or in a different region (in which case it is called an external state and the connecting arc is called a *cross–arc*).

Each *Generator* sends all external states, with the appropriate cross-arcs, to the *Generators* determined by the partitioning function. In order to perform state processing concurrently with communication, each *Generator* is composed of three processes (Figure 3.1): the *Worker*, responsible for the processing of states, the *Sender*, responsible for sending states and

arcs to other processes, and the *Receiver*, responsible for receiving states and arcs from other processes and for the termination detection. When the *Spawner* creates the *Generators*, it actually creates *Worker* processes. As the first step, each *Worker* creates its *Receiver* and *Sender* processes.

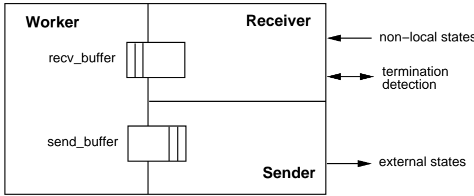

Fig.3.1. The structure of a *Generator*.

The *Worker*, *Receiver*, and *Sender* of each *Generator* reside on the same processor. Their communication is based on shared variables: the *Worker* communicates with the *Receiver* using a shared memory segment *recv_buffer* in the standard producer–consumer scheme; similarly, the *Worker* and the *Sender* communicate via a shared memory segment *send_buffer* also using the producer–consumer scheme.

Each *Generator* processes the states from the internal queue *unexplored* (local states) and from *recv_buffer* (non–local states), with non–local states taking priority over local ones. External cross-arcs are not critical for the state space generation, so they are collected at the generating hosts and sent to the appropriate regions after the termination of the state space generation. Such a solution reduces the traffic in the connecting links during state space generation.

```
1. Worker:
2. var States_i := ∅;              (* set of states *)
3.     Arcs_i := ∅;                (* set of arcs *)
4.     unexplored := ∅;            (* queue of states *)
5.     cont := true;               (* continuation flag *)
6. begin
7.     spawn Receiver, Sender on this_host;
8.     while cont do
9.        if empty(recv_buffer) ∧
                     nonempty(unexplored) then
10.           state := remove(unexplored);
11.           new := true
12.       else
13.           ⟨state, id⟩ := get(recv_buffer);
14.           if state = null then
15.               cont := false
16.           else
17.               new := not found(States_i, state);
18.               if new then
19.                   States_i := States_i ∪ {state}
20.               endif;
21.               Arcs_i := Arcs_i ∪ {⟨id, state.id,
                                      state.prob⟩}
22.           endif
23.       endif;
24.       if cont ∧ new then
25.           for each s in successors(state) do
26.               if region(s) = i then
27.                   if not found(States_i, s) then
28.                       States_i := States_i ∪ {s};
29.                       insert(unexplored, s)
30.                   endif;
31.                   Arcs_i := Arcs_i ∪ {⟨state.id,
                                      s.id, s.prob⟩}
32.               else
33.                   put(send_buffer, ⟨s, state.id⟩)
34.               endif
35.           endfor
36.       endif
37.   endwhile
38. end.
```

An important aspect of distributed applications is the termination condition. When a *Generator* runs out of unexplored states, it waits for states from other processes. In order to prevent a “wait forever” situation in which all *Generators* are idle and wait for each other, a global termination detection algorithm, proposed by Dijkstra [3], is interleaved with the computation. This termination algorithm checks if all processors have finished their computations.

When the construction of all regions is completed, each *Generator* sends the states and arcs to the *Collector* and then terminates.

Physical processes residing on different hosts constitute a “virtual machine”; they communicate by message passing using the popular PVM (Parallel Virtual Machine) package [4].

4. Petri net model

The experiments illustrated in Fig.1.1 and Fig.1.2 were performed on a cluster of computers connected to a switch by a 100 Mb Ethernet. An outline of such a cluster is shown in Fig.4.1 (for 4 machines; in the original experiments there were 32 machines).

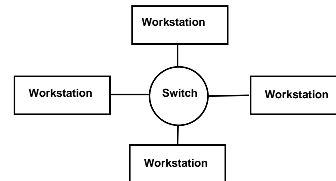

Fig.4.1. An outline of a cluster of 4 workstations.

A Petri net model of a single machine connected to a switch, with independent processes for state generation and for message passing to other processors, is shown in Fig.4.2.

The processing of states is represented by transition t_{ip} with the average time of processing one state assigned to it as the firing time. The queue of states waiting for processing is represented by place p_{ir}, which, in Fig.4.2, contains 2 tokens (i.e., 2 states).

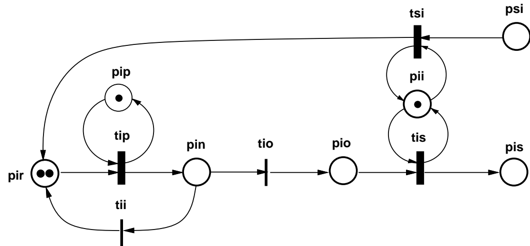

Fig.4.2. Petri net model of a processor and its link.

Transitions t_{is} and t_{si} represent message passing to and from the switch, respectively. Since the messages share the same link, place p_{ii} is a shared resource (the link) which can be used either for sending (t_{is}) or for receiving (t_{si}) a message.

For a 4–machine cluster (Fig.4.1), the switch connecting the machines is shown in Fig.4.3; each of messages incoming from four directions (p_{1s}, p_{2s}, p_{3s} and p_{4s}) has a free–choice structure which forwards the message to one of the other hosts connected to the switch. It is assumed that the states are uniformly distributed over the regions, so all free–choice probabilities associated with the selections within the switch are equal, and, in Fig.4.3, are equal to 1/3.

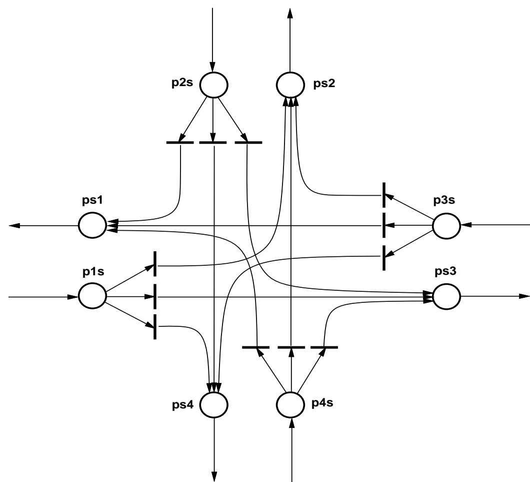

Fig.4.3. Petri net model of a switch.

In Fig.4.2, the result of processing a state is a new state, which, after termination of t_{ip}'s firing, is inserted into place p_{in}. If this new state is local, it is forwarded, by firing t_{ii}, to the waiting queue p_{ir} for further processing; if the new state is external, it is sent to p_{io} by firing t_{io}, and then to its target host by firing first t_{is} and then t_{sj} of the selected host j (the selection is within the switch).

Place p_{in} is a free–choice place, and the selection of local or external state is described by free–choice probabilities associated with t_{ii} and t_{io}; for a cluster of n_p machines, assuming uniform distribution of states over regions, these free–choice probabilities are $1/n_p$ for t_{ii} and $(n_p - 1)/n_p$ for t_{io}. Moreover, within the switch, the free–choice transitions connected to places p_{is} have the free–choice probabilities equal to $1/(n_p - 1)$, so all hosts are selected with the same probabilities equal to $1/n_p$.

5. Analysis of the model

The performance of distributed applications depends upon several factors, which include the granularity of the tasks and the balancing of the workload among the processors, and also the amount of communication which is needed for the execution of tasks distributed among the processors. It appears that a ratio of computation to communication time, $r_{comp/comm}$, is one of important factors characterizing the performance of distributed applications. For distributed state space generation, this ratio characterizes the amount of computation and communication needed for processing a single state.

In order to represent the steady–state performance of distributed state space generation, it is assumed that each analyzed state generates one new state. The workload of the system is controlled by the initial marking of places p_{ir} (Fig.4.2), and it is assumed that the initial marking is uniformly distributed over the processors (otherwise the initial transient behavior would occur to remove the differences).

The utilization of processors, for different values of the computation to communication time ratio and for several numbers of processors is shown in Fig.5.1 for small workload (the initial marking of p_{ir} equal to 2), in Fig.5.2 for medium workload (the initial marking of p_{ir} equal to 4), and in Fig.5.3 for high workload (the initial marking of p_{ir} equal to 8); the figures show the utilization of processors as a function of two variables, the number of processors n_p (from 2 to 16) and the value of time ratio $r_{comp/comm}$, with the values from 0.25 to 2.5.

All plots have similar character, and it can be observed that the utilization increases with the increase of the workload, although Fig.5.3 shows some “saturation effects” for larger values of $r_{comp/comm}$. Moreover, for communication times greater than the computation time (i.e., for $r_{comp/comm}$ less than 1.0), processor's utilization is rather poor, below 50%, and almost linearly tends to 0 with the value $r_{comp/comm}$.

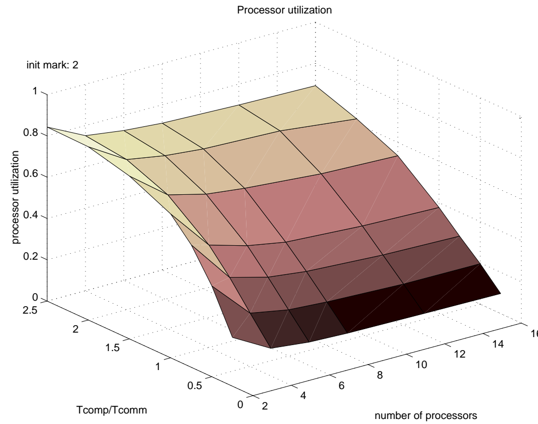

Fig.5.1. Utilization of processors for $m_0 = 2$.

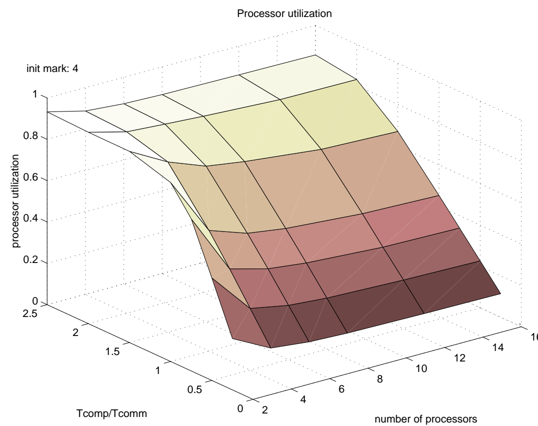

Fig.5.2. Utilization of processors for $m_0 = 4$.

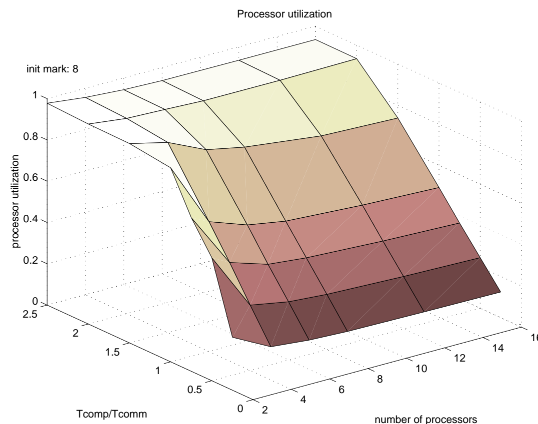

Fig.5.3. Utilization of processors for $m_0 = 8$.

Processor utilization can be used to determine the speedup as a function of the number of processors, n_p. The speedup $S(n_p)$ of an n_p–processor system is usually defined as the ratio of execution time of an application on one processor, $T(1)$, to the application's execution time on n_p processors, $T(n_p)$:

$$S(n_p) = \frac{T(1)}{T(n_p)}.$$

For the ideal, uniform distribution of workload on all processors, the execution time on n_p processors can be expressed as:

$$T(n_p) = \frac{T(1)}{n_p} \frac{1}{u_p(n_p)}$$

where $u_p(n_p)$ is the utilization of each processor in an n_p–processor system, and then the speedup is simply:

$$S(n_p) = n_p u_p(n_p).$$

Fig.5.4 shows several speedup curves corresponding to the processor utilization shown in Fig.5.2 (i.e., for medium workload). Although almost linear speedup is obtained in all cases, the actual value of the speedup depends in a significant way on the value of the ratio $r_{comp/comm}$; the larger is this values, the closer is the real speedup to the ideal one.

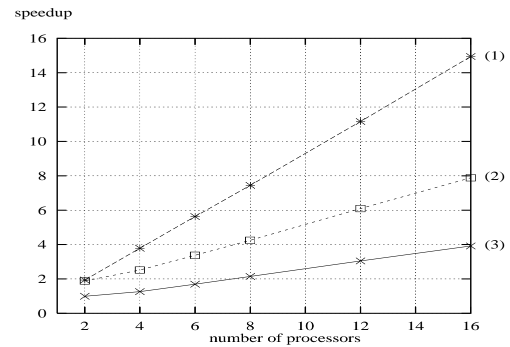

Fig.5.4. Speedup as a function of n_p; (1) $r_{comp/comm} = 2$, (2) $r_{comp/comm} = 1$, (3) $r_{comp/comm} = 0.5$.

The results shown in Fig.5.4 are obtained with the assumption that there is sufficient workload for all processors used. If this is not the case, and if the same workload is divided among the processors, the increasing number of processors will result in decreasing workload assigned to each of the processors, which decreases the utilization of the processors.

Fig.5.5 shows the speedup characteristics corresponding to this situation, for two cases, when the ratio is equal to 0.5 and when it is equal 2.0.

It should be observed that the speedup values in Fig.5.5 are significantly smaller than the ones in Fig.5.4.

6. Concluding remarks

The paper presents a very preliminary study of the performance of distributed generation of the state

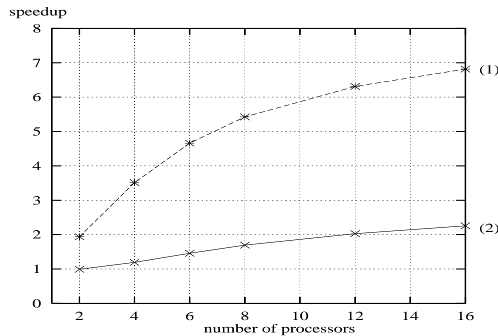

Fig.5.5. Speedup as a function of n_p;
(1) $r_{comp/comm} = 2$, (2) $r_{comp/comm} = 0.5$.

space for timed Petri nets. A timed Petri net model is used to represent the essential features of the distributed application. Evaluation of the this model (by simulation techniques) provides an insight into the influence of major modeling parameters on the performance of the application, and the speedup that can be obtained on n_p processors.

The presented approach uses many assumptions to simplify the model as much as possible and still capture the essential aspects of the distributed behavior. For example, any realistic application cannot be characterized by the ideal distribution of the workload; the steady–state behavior is also an idealization of the real behavior. However, the obtained characterizations are consistent with experimental results, and it is expected that further refinements of the model will provide further insight into the behavior of distributed applications.

Petri net models of distributed systems contain many 'regularities' which can be used for model reduction. For example, in colored Petri nets [5], tokens are associated with attributes (called colors). An immediate application of such attributes is to represent different processors (or nodes) of a system by different colors and "fold" the model of a distributed system into a single processor, significantly simplifying the model, but not its analysis.

The presented approach is not restricted to the discussed state space generation; it can be used to analyze the behavior of other applications which are based on reachability analysis, e.g., model checking or verification of discrete systems.

Acknowledgement

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

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance evaluation of systems"; *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 93–122, 1984.

[2] F. Bause, and P. Krinzinger, *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg 1996.

[3] E. Dijkstra, W. Feijen, and A. van Gasteren, "Derivation of a termination detection algorithm for distributed computations"; *Information Processing Letters*, vol. 16, no. 5, pp. 217–219, 1983.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial.* MIT Press 1994.

[5] K. Jensen, "Coloured Petri nets"; in: *Advanced Course on Petri Nets 1986* (Lecture Notes in Computer Science 254), Rozenberg, G. (ed.), pp. 248–299, Springer–Verlag 1987.

[6] P. Marenzoni, S. Caselli, and G. Conte, "Analysis of large GSPN models: a distributed solution tool"; *Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pp. 122–131, 1997.

[7] T. Murata, "Petri nets: properties, analysis, and applications"; *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[8] J. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice Hall 1981.

[9] I. Rada, "Distributed generation of state space for timed Petri nets"; M.Sc. Thesis, Department of Computer Science, Memorial University of Nfld, St.John's, Canada A1B 3X5, 2000.

[10] W. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press 1994.

[11] W.M. Zuberek, "On generation of state space for timed Petri nets"; *Proc. of ACM 16th Annual Computer Science Conference*, pp. 239–248, 1988.

[12] W.M. Zuberek, "Timed Petri nets, definitions, properties, and applications"; *Microelectronics and Reliability* (Special Issue on Petri Nets and Related Graph Models), vol. 31, no. 4, pp. 627–644, 1991.

[13] W.M. Zuberek, "Petri nets and timed Petri nets: basic concepts and properties" (Lecture notes for the course "Modeling and Analysis of Computer Systems"); Technical Report #2000–01, Department of Computer Science, Memorial University of Nfld, St. John's, Canada A1B 3X5, 2000.