Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

AN APPROACH TOWARDS DISTRIBUTED SIMULATION OF TIMED PETRI NETS

Devendra Kumar
Saad Harous

Department of Computer Eng. and Science
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT

We present a model of Timed Petri Nets which is more general than known models in terms of modeling convenience. The model consists of simple but fairly general modules. This would result in simpler and more modular codes for simulation of these systems, as compared with the known models of Timed Petri Nets. After discussing this model, we present an approach towards its distributed simulation. The well known distributed simulation schemes for discrete event systems do not directly apply to these systems due to non-autonomous nature of *place* nodes in Timed Petri Nets. Moreover, in our approach we incorporate several ideas to increase the degree of concurrency and to reduce the number of overhead messages in distributed simulation.

1. INTRODUCTION

Petri Nets are useful for modeling parallel systems; using them synchronization among processes can be captured in a precise and simple way. The classical model of Petri Nets is briefly described in section 2. Essentially, in this model the system is represented by a graph with two kinds of nodes: *places* and *transitions*. The state of a place node at any point consists of the number of *tokens* it has. An event in the system is represented by a "firing" at a transition; any such firing would destroy some tokens and generate some tokens at certain place nodes. States of various nodes in the system determine when various transitions can fire.

In the classical definition of Petri Nets no facilities are provided to model the duration of system activities, e.g., what is the duration between removal of tokens at the start of a firing and the time when new tokens start getting generated, or when these new tokens are fully available to other transitions in the system to determine if those transitions can fire, etc. This makes it impossible to predict the actual times at which various events in the system would take place (relative to the time when the system starts). Moreover, since no time constraints are placed on the system behavior, the model may predict that certain events *can* take place even when those events are impossible to occur due to time constraints. Similarly, time constraints in an actual system may dictate that certain events will never take place concurrently, whereas the model may predict them to possibly take place concurrently. Therefore having the notion of time in the model would be more informative in modeling and simulation of systems by Petri Nets.

Several researchers have proposed various forms of extensions to the classical definition of Petri Nets in order to capture the notion of time, e.g., [Alanche et al. 1985, Coolahan 1983, Ramchandani 1974]. Such extended models of Petri Nets are called *Timed Petri Nets*. As pointed out in [Garg 1985], it is easy to represent the behavior and to analyze the performance characteristics of the logic of many systems using Timed Petri Nets. Timed Petri Nets have in fact been used to model and analyze many systems [Alanche et al. 1985; Carlier et al. 1984; Chretienne 1983; Coolahan 1983; Ozsu 1985a, 1985b; Ramchandani 1974]. In particular, Timed Petri Nets are a simple and elegant tool for representing distributed systems because of their capability to clearly describe concurrency, conflicts and synchronization among processes [Garg 1985]. This is especially true for parallel or distributed systems where the notion of time is critical, such as embedded real-time systems. In section 3 we review the two major ways in which Timed Petri Nets have been defined in the literature.

In the above two approaches, time is associated with either places or transitions, but not both. Even though the two approaches are equivalent in the sense that a model in one approach can be represented by a model in the other approach, this may be quite inconvenient to the user since it would often require defining artificial places or transitions in the Timed Petri Net model of an actual system. In our model of Timed Petri Nets, we associate time with both places and transitions, thereby simplifying the modeling process in such cases. The above models for Timed Petri Nets follow as special cases of our model. Also, we further decompose the place or transition nodes to consist of certain modules; these modules are not only fairly simple, but also quite general to allow for a wide variety of ways in which time can be associated with places or transitions. Having place or transition nodes to consist of such simple modules would result in simple and modular codes for the simulation of these systems. The nature of a place or transition node can be modified by simply changing one or more of its modules without affecting its other modules. Our model of Timed Petri Nets is discussed in section 4.

Having defined our model of Timed Petri Nets, we next consider its *distributed simulation*. Distributed simulation is a simulation where the simulator is a distributed network of processes, where usually each process simulates a part of the system being simulated. The communication among these processes is via messages. In distributed simulation there is usually no global simulation clock as in traditional sequential simulation. Synchronization among processes is accomplished by appending to each event message the time of occurrence of this event.

The motivation for distributed simulation is that sequential discrete event simulation usually requires an enormous amount of execution time. Distributed simulation is a potential alternative to improve the performance of simulation, since the various processes of a distributed simulator could be progressing in parallel to simulate various parts of the actual system. In Timed Petri Nets, indeed such concurrency often exists; for example two transitions that do not conflict with each other could be simulated concurrently if they are both ready to fire.

In section 5 we briefly review the major distributed simulation schemes [Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982] available in the literature. These schemes are defined for a large subclass of discrete event systems.

In section 6 we discuss the problems in applying these schemes to the simulation of Timed Petri Nets and our approach towards distributed simulation of these systems. The most notable problem is that Timed Petri Nets do not satisfy a fundamental assumption made in these schemes, namely, a simulating process can *autonomously* predict its output message history up to time t provided it knows its input message histories up to time t. In Timed Petri Nets, if a place is connected to several output transitions, then a token at that place may go through any one of the output transitions. Such a place node can not autonomously determine which output transition should be given the token. Due to this conflict among output transitions of a place node, and the inability of the place node to autonomously resolve this conflict, these schemes [Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982] do not apply to Timed Petri Nets in a simple manner. In section 6, we discuss how to solve this problem of non-autonomous nature of a place process.

Another serious problem in applying these schemes (or simple variations of these schemes that handle the problem mentioned above) to Timed Petri Nets is that potentially a large number of overhead messages may be generated during the simulation. Overhead messages are ones that do not represent actual events in the system being simulated, but they are needed to ensure progress of the simulation. In any distributed simulation of any system, overhead messages are often needed to avoid potential deadlock situations that may arise in the distributed simulation. As discussed in section 6, we avoid the problem of deadlock by using a variation of ideas in [Chandy and Misra 1979]. Specifically, we use *NULL* messages with a slightly different meaning than the ones in [Chandy and Misra 1979].

The schemes discussed in section 5 are designed to be correct for a large subclass of discrete event systems. Due to this generality, they often result in a large amount of overhead when directly applied to the simulation of arbitrary systems. In this paper, we are concerned only with the simulation of Timed Petri Nets, and hence we exploit special nature of these systems to develop ways of reducing the amount of overhead and ways of increasing the degree of concurrency in the simulation algorithm (e.g., relaxing restrictions as to when an output message can be sent out from a simulating process). These ideas are also discussed in section 6.

Finally, section 7 gives concluding remarks.

2. DEFINITION OF PETRI NETS WITHOUT NOTION OF TIME

A Petri Net [Peterson 1981] is a graph with two types of nodes: *places* and *transitions*. An *edge* (also called an *arc*) exists only from a place to a transition or from a transition to a place. No multiple edges are allowed from a given node to another given node. If there is an arc directed from a place P to a transition T, then P is called an *input place* of transition T, and T is called an *output transition* of place P. Similarly, we define an *input transition* of a place and an *output place* of a transition corresponding to an arc from a transition to a place. At any moment, a place may have zero or more *tokens*. Graphically, places, transitions, arcs, and tokens are represented respectively by: circles, bars, arrows, and dots. Figure 1 below shows an example Petri Net. Here the places are P1, P2, P3, and P4. The transition are T1, T2, and T3. P1 is an input place of T1, and T1 is an output transition of P1. Similarly, T1 is an input transition of P2, and P2 is an output place of T1. At the current point, the places P1 and P3 have exactly one token each, and places P2 and P4 have no tokens. For an in-depth treatment of Petri Nets, the reader is referred to [Peterson 1981].

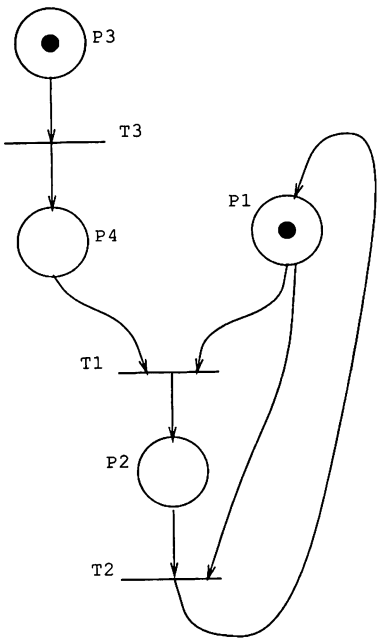

Figure 1. An Example Petri Net

A transition node is said to be *ready* if and only if there is at least one token at each of its input places. At any moment, zero or more transitions are chosen to "fire". Each of these chosen transitions must be ready at that moment. Also, for each such chosen transition, there should be an association of input tokens (one from each of its input places) such that no token is associated with two or more different chosen transitions. When a transition fires, its associated tokens at its input places are removed, and a new token is deposited at each of its output places. (If a place is an output place for several chosen transitions, then one token should be deposited on behalf of each of these transitions). The firing of a transition is instantaneous. At any moment, the choice of transitions to fire is non-deterministic, i.e., the set of transitions that can be chosen to fire together at the current moment may not be unique. For example consider a system with only three transitions T1, T2, and T3 which share the same input place P and have no other input places. Also suppose currently this place P has two tokens. Then the set of transitions that are chosen to fire could be any two of them, or any one of them, or none of them. (However, in general it is assumed that some reasonable progress would be made eventually, if currently one or more transitions are ready).

Two transitions are said to be *neighbors* if they share at least one input place. Note that two neighboring transitions can fire simultaneously only if each of their common input places has at least two tokens.

3. TIMED PETRI NETS PROPOSED IN THE LITERATURE

A Timed Petri Net is a Petri Net with time associated

with places or transitions or both. This time would represent the time duration of certain activities in the system, e.g., the time duration between when a transition starts firing and when the new tokens generated by this firing have become available at the corresponding output places, or the time duration between when a new token has appeared at a place node and when it is available to the output transitions of this place to determine if they can fire. These associated times may be deterministic or probabilistic.

Models proposed in the literature associate time with either places or transitions, but not both. The two approaches, and the meaning of time usually assumed in these approaches, is briefly summarized next.

3.1 Approach 1 (Time Associated With Transitions Only)

In this approach, with each transition T a time δ is associated, called the *execution time* of transition T. The usual meaning of this time δ is that when transition T starts firing at time t, a token is removed from each input place of T at time t, and a token is placed at each output place of T at time t + δ [Ramchandani 1974]. Note that no time is associated with a place node in this approach; thus whenever a token is placed at a place node, it is immediately available to the output transitions of this place node.

3.2 Approach 2 (Time Associated With Places Only)

In this approach, with each place P a time δ is associated. The usual meaning of this time δ is that when a token is placed at the place P at time t, it remains unavailable to output transitions of place P until time t + δ [Alanche et al. 1985; Coolahan 1983; Ramchandani 1974]. In this approach no time is associated with transitions. Thus whenever a transition fires, the corresponding input tokens are removed immediately and the corresponding output tokens are placed at its output places immediately.

3.3 Equivalence Of The Two Approaches

As discussed in [Alanche et al. 1985; Ramchandani 1974], these two models of Timed Petri Nets are equivalent. Given a model in approach 1, one may arrive at an equivalent model in approach 2 by simply replacing each transition node T with associated time δ by a serial network consisting of transition T, a new place P, and a new transition U, where the transitions T and U now have no time associated with them and the new place P is associated with the time δ. All the place nodes that existed in the original model are now associated with time zero. As further simplification, if $\delta = 0$ then nodes P and U need not be created at all.

Similarly, given a model in approach 2, one may arrive at an equivalent model in approach 1 by simply replacing each place node P with associated time δ by a serial network consisting of place P, a new transition T, and a new place Q, where the places P and Q now have no time associated with them, and the new transition T is associated with time δ. All the transition nodes that existed in the original model are now associated with time zero. As before, the obvious simplification can be made when $\delta = 0$.

4. OUR MODEL OF TIMED PETRI NETS

As an overview, in our model the above two approaches to defining Timed Petri Nets are generalized in two important ways:

1. We associate a time δ_i with each place and each transition node i in the system.
2. the time δ_i associated with a place or transition node is no more a simple delay, but rather in our model arbitrary queuing mechanisms can be associated with this time δ_i.

Our model is discussed in detail next. Each place node consists of three subnodes as shown in Figure 2, and describe below.

1. The first subnode is called a *Pmerge* process. A *Pmerge* process has one or more input lines, and one output line. Whenever a token arrives along one of its input lines, it sends it out via its output line after zero delay. If occasionally two or more tokens arrive at the same time, they are sent out simultaneously via the output line after zero delay. This process is almost the same as the *merge* process defined in [Chandy and Misra 1981; Kumar 1989b], which is often used in queuing networks; the only differences are that (i) a Pmerge process can possibly have only one input line whereas a merge process has two or more input lines, and (ii) if several tokens arrive at the input of a Pmerge process simultaneously, then they are sent out as separate tokens, rather than as a single, composite token as in case of simultaneous input messages at a merge process (obviously, the two views are equivalent, but at the simulation level they may result in different simulation code and different simulation messages, if the simulation messages are to closely match the messages in the model).
2. The second subnode is an arbitrary *queue* process with the associated service time δ_i. The queue may be defined in several different ways. For example, it may be a FCFS queue with a single server with service time δ_i, or a FCFS queue with infinite number of servers with service time δ_i, etc.
3. The last subnode is a place node with no time associated with it, as in the usual Petri Nets without association of times. In our model we call this node (or process) a *zero-place* process.

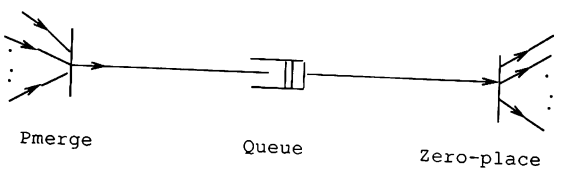

Figure 2. Subnodes in a Place Node

The transition node in our model is also broken down into three subnodes as shown in Figure 3. These subnodes are described below.

1. The first subnode is a transition node with no time associated with it, as in the usual Petri Nets. In our model we refer to this process as a *zero-transition* process. Note that it has only one output line; thus it generates only one output token in firing.

2. The second subnode is an arbitrary *queue* process defined in the same way as above in the context of a place node.
3. The last subnode is a *Tfork* process which is slightly different from the *fork* processes defined in [Chandy and Misra 1981; Kumar 1989b]. A *Tfork* process has one input line, and one or more output lines. Whenever a token arrives along its input line, it sends out an identical copy of that token along each of its output lines without any time delay. (In contrast, a fork process would send its input to *only one* of the output lines).

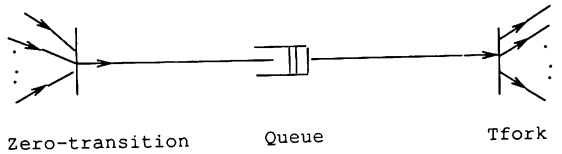

Figure 3. Subnodes in a Transition Node

In our simulation we will assume that the moment a zero-transition process T becomes ready, it will fire immediately unless the other zero-transitions chosen to fire at that moment make it *impossible* for T to fire at that moment (i.e., if T has an input place P such that all the tokens at P have become associated with other chosen transitions). Thus, for example, we preclude a scenario where a zero-transition would simply keep idling (i.e., not firing) even when it is ready and has no conflicts with other transitions. Also, several assumptions similar to [Chandy and Misra 1979] are needed to ensure progress and termination of the simulation, we skip these details here.

Our model of Timed Petri Nets offers several advantages over the other models discussed in section 3:

1. Generality, and Modeling Convenience: The model has more expressive power because the queue processes that are part of various place or transition nodes could be arbitrary. A queue process may follow any of the job scheduling disciplines and it may have any number of servers. In contrast, the two models of Timed Petri nets defined in section 3 assume, in effect, a FCFS discipline with infinite number of servers (which is the same as a delay process that simply delays an input job by the time δ).

Note that in our model if we restrict the queues to follow only the FCFS discipline with infinite number of servers, then our model becomes equivalent to the previous two models. However, even with this restriction our model is still more convenient to use since in the previous two models one has to define artificial place or transition subnodes to represent an original node which has time associated with it. In our model, to represent any node with an associated time, one needs to create three subnodes such that one of these subnodes is the same Petri Net node without an associated time, and the other two subnodes are simple and often familiar queuing processes. Thus creating or interpreting a Timed Petri Net in our model would be intuitively simpler.

2. Modularity: In our model of Timed Petri Nets, a node consists of simple subnodes that capture various aspects of behavior of that node. For example, the timing aspects are delegated to simple queuing processes and the logic of a *zero-place* or a *zero-transition* subnode does not have to deal with timing issues. The *zero-place* process does not have to deal with the issue of merging input tokens arriving along various input lines. The *zero-transition* process does not have to know where the output tokens generated in firing will have to be placed. Such modularity does not exist in previous definitions of Timed Petri Nets. For example, in the approach that associates time with a transition, the logic of when to start firing, when to create output tokens, where to place output tokens — is all combined in a single process.

Due to this modularity, our model offers several advantages. It simplifies the simulation code (in either sequential, distributed, or parallel simulation). Simple changes in the system being modeled would result in relatively simple changes in its model and the code simulating that model. For example, if the queuing discipline to be followed by a node is to be changed then only the corresponding queue process and its code needs to be changed, not the processes *zero-place* or *zero-transition*. Similarly, if the outgoing edges from a transition are changed then only the code for the corresponding *Tfork* process would change.

5. REVIEW OF DISTRIBUTED SIMULATION ALGORITHMS

Borrowing terminology from [Chandy and Misra 1979], in the following a system to be simulated is called a *physical system* (in our case it is a Timed Petri Net). The physical system consists of a network of physical processes (or *pps* for short) - in our case the *pps* are the subnodes defined above. Each *physical system* is simulated by a distributed simulator called a *logical system*. A *logical system* is a collection of *logical processes* (or *lps* for short), each one simulating a corresponding physical process.

Deadlock is a major problem in designing an algorithm for distributed simulation. Several algorithms for distributed simulation have been developed [Bryant 1977; Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982; Peacock et al. 1979a, 1979b]. These algorithms use overhead messages to ensure progress of simulation.

The different distributed simulation algorithms available in literature are divided mainly into 3 classes depending on the method used to handle the deadlock problem — these methods are (i) deadlock avoidance, (ii) deadlock detection and recovery, and (iii) rollback. In the first two methods, a logical process will not progress until it is sure that it is safe to do so. On the other hand, in the third method, the process progresses with the assumption that its input messages are correct and have arrived in the right order at the input port, until it detects that a message has arrived out of order — then it rolls back to an earlier state in order to correct the above assumption. Below we briefly summarize these three methods.

Avoidance methods: The algorithms which are based on the avoidance approach [Bryant 1977; Chandy and Misra 1979; Peacock et al. 1979a] require that a process sends out some kind of control message along the output lines indicating an estimate of the lower bound on the time stamp of its next event message. These overhead messages allow the receiver processes to advance their clocks, thereby preventing deadlock from occurring. For example, in the scheme proposed by [Chandy and Misra 1979] deadlock is avoided by using overhead NULL messages. A NULL message (t,NULL) sent from *lp i* to *lp j* informs *lp j* that no further messages would be sent on line *(i,j)* up to time t. The receiving *lp* would update the input history and advance the input line clock accordingly.

Deadlock detection and recovery methods: [Chandy and Misra 1981] presents a distributed simulation

scheme based on deadlock detection and recovery. In this scheme, a distributed algorithm proposed in [Dijkstra and Scholten 1980] is used for deadlock detection. Deadlock is broken in a distributed manner by determining "next message" to be sent in the logical system. Determining "next message" is similar to determining "next event" in the event-list mechanism in sequential simulation.

Rollback method: The Time Warp simulation method [Jefferson and Sowizral 1982] attempts to improve performance of distributed simulation by developing a new scheme based on the assumption that most of the messages will arrive in the right order. An *lp* will process the message it has on hand assuming that no message will arrive in the future with an earlier time stamp. Later, if a message with earlier time stamp arrives, it will roll back to an earlier state. During this rollback phase, one or more anti-messages will be generated to inform other processes about this rollback and thus to put back the simulation on the right track.

The number of overhead messages is usually large. For a detailed survey on distributed simulation schemes, we refer the reader to [Misra 1986]. Since the publication of this survey, several new works have appeared. In particular, [Reed 1983, 1985; Reed et al. 1988] points out some negative performance results regarding the schemes in [Chandy and Misra 1979, 1981]. Several researchers have looked into variations of the general purpose schemes or distributed simulation of specific classes of systems, e.g., [Lubachevsky 1989; Fujimoto 1989].

6. OUR APPROACH TOWARDS DISTRIBUTED SIMULATION OF TIMED PETRI NETS

As in [Chandy and Misra 1979, 1981] each *pp* (physical process), i.e., a subnode, is simulated by a corresponding *logical process* (or *lp*). We assume infinite input buffers at the input port of any *lp*. Also, we assume that each communication line in the logical system is FIFO (first-in-first-out) with arbitrary but finite communication delays, and error free. An *lp* can send out a message whenever it has one without waiting for the receiver *lp* to be ready to receive it. Messages sent on a line are stored at the input port of the receiver in the order they are sent. Later the receiver *lp* receives them in FIFO order for any given line. In the following discussion, it is assumed that whenever a message is sent from an *lp i* to an *lp j*, the corresponding communication line does exist.

Note that the actual events in the system are the generation, removal, or transfer of various tokens in the system. More specifically, the events are: (i) the transfer of a token from the input of a Pmerge or queue process to its output, (ii) the firing of a *zero-transition*, i.e., removing input tokens and generating an output token, and (iii) removal of a token from input of a Tfork process and generation of corresponding output tokens. In addition to the corresponding event messages, we will need various overhead messages. Next we discuss the various kinds of problems that one has to consider in the distributed simulation of these systems, and our approach to handle them.

6.1 How To Handle Non-Autonomous Nature Of A Zero-place

To start with, we first point out a basic problem in applying the well known distributed simulation schemes [Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982] for the distributed simulation of these systems. Consider the system shown in Figure 4.

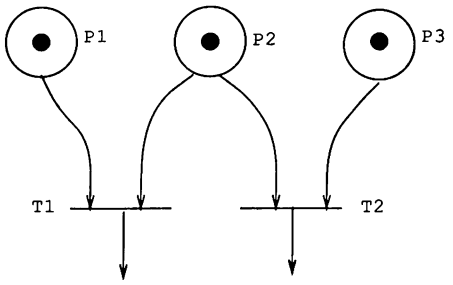

Figure 4. To Show Non-Autonomous Nature of Place Nodes

Here, we have two transitions T1 and T2 in the system. As shown in the figure, P1 and P2 are input places of T1, and P2 and P3 are input places of T2. Suppose the corresponding zero-place *lps* P1, P2 and P3 currently hold one token each with time stamps equal to t_1, t_2 and t_3 respectively where $t_2 < t_1$ and $t_2 < t_3$. If $t_1 < t_3$ then the corresponding zero-transition *pp* T1 would fire, but if $t_3 < t_1$ then T2 would fire. (If $t_1 = t_2$ then the choice between the two transitions is arbitrary). Therefore, a process simulating the zero-place P2 cannot decide on its own as to whether to give out its token to transition T1 or to transition T2. This violates the assumption used in the schemes [Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982] that a simulating process can *autonomously* determine its output message histories until time *t* if it knows its input message histories until time *t*.

To avoid the above problem, one has to use certain overhead messages among the neighboring zero-transitions and their input zero-places so as to determine which transition should fire next. We take the following approach. When a token is available at a zero-place *lp i* with time stamp *t*, it sends a message *tokavail(t)* to its output zero-transitions. These zero-transition *lps* will communicate with each other to determine if and when one of them fires. When a zero-transition *j* fires, it sends a *grabbed* message to all its input zero-places and to all its neighboring zero-transitions, and simulates its firing. Each zero-transition that receives a *grabbed* message from the zero-transition *j* comes to know that the corresponding tokens at input zero-places shared with *j* have been grabbed, and therefore it assumes that any such token is no longer available. (It is possible that this zero-transition has not yet received the *tokavail(...)* messages corresponding to some of these input tokens; it has to remember for future that indeed these tokens have been grabbed). Once a zero-place *i* gets the *grabbed* message, it would work for its next available token if any, as before.

How do the neighboring zero-transitions determine when and which one of them fires? When a transition u has a *tokavail(...)* message from each of its input zero-places, it needs to know whether it would be ready to fire *before* its neighbors (or at the same time, as discussed below). To this end, for each zero-transition *lp i* we define CT_i, its *logical clock*, as follows. (As a convention in this paper, we use a subscript *i* to refer to the value of a local variable or a quantity known locally at a process *i*). Intuitively, CT_i represents the earliest time known to the zero-transition *lp i* when it can possibly fire. Let $CP_i[k]$ be the minimum time when input zero-place *k* would have a token available next, as known at zero-transition *i*. Thus if zero-transition

i has received a message *tokavail(t)* from zero-place k, then it would set $CP_i[k]$ to t. (Later, we will see *NULL* messages that also affect the value of $CP_i[k]$). The value of CT_i for any zero-transition i is defined to be the maximum value of $CP_i[k]$ over all its input zero-places k. Obviously, if zero-transition i has received *tokavail(...)* messages from its every input zero-place and if $CT_i < CT_j$ for its every neighboring zero-transition j, then zero-transition *pp* i will fire at the time CT_i (i.e., the zero-transition *lp* i will fire with time stamp CT_i). What if $CT_i = CT_j$ for one of the neighbors j? We break the tie in this case by comparing the ids i and j. In other words, we first define a total ordering among ordered pairs of numbers by $(x, y) < (u, v)$ if and only if (i) $x < u$ or (ii) $x = u$ and $y < v$. Then we require that a zero-transition i will fire if it has received *tokavail(...)* messages from its every input zero-place and if $(CT_i, i) < (CT_j, j)$ for its every neighboring zero-transition j.

How does a zero-transition i know about the clock values CT_j? One possibility is that each zero-transition i transmits its current CT_i value to all its neighbors, whenever this value changes. However, that may cause too many such overhead messages, since they are being sent even when the receiver *lp* does not need them. Therefore, we use a "demand-driven" strategy — whenever i has tokens at each of its input zero-places, it sends a message *request(CT_i)* to all its neighbors. Any zero-transition j that is a neighbor of i keeps a variable $CT_j[i]$ that remembers the clock value of i as is currently known at j. For notational simplicity, we use the expression $CT_j[j]$ to be the same as CT_j. On receiving the message *request(CT_i)* from i, j would update $CT_j[i]$. Subsequently, j would send a message *answer(CT_j)* to i whenever (currently or later) j finds that $(CT_j[j], j) > (CT_j[i], i)$. At most one such *answer(...)* message is sent from j to i for a given request. (As a minor optimization, if i already knows that $CT_i[j] > CT_i[i]$, then it does not have to send a *request* message to j).

6.2 How To Handle The Deadlock Problem

Another important problem one has to deal with in any distributed simulation in general, is the possibility of deadlocks. To see the possibility of deadlocks in our case, consider the Timed Petri Net shown in Figure 1. (The times associated with nodes, and the queuing disciplines are not shown in the figure. Also, each node is assumed to consist of the corresponding subnodes as discussed in section 4).

Suppose at time zero, the zero-place nodes P1 and P3 have a token. After the firing at T3 suppose zero-place P4 has a token at time 10 available for the zero-transition T1. Obviously at time 10, transition T1 should fire. However, following our approach above, T1 would send a message *request(10)* to zero-transition T2. T2 would not send back any *answer(...)* message to T1 since it has no information from P2. Obviously there will be a deadlock cycle {zero-transition T2→ entire place node P2→ entire transition node T1→ zero-transition T2}.

To resolve the deadlock problem, we use a slight variation of NULL messages defined in [Chandy and Misra 1979]. In our approach, a NULL message on a line with time stamp t means that the next message on this line will have a time stamp greater than or equal to t. To understand the use of NULL messages in our approach, consider again Figure 1 where a token is available at the output of zero-place P1 at time zero, and at the output of zero-place P4 at time 10. In our approach, the zero-transition T1 would send out a message NULL(10) to its output queue *lp*. The queue *lp* would subsequently send out a message NULL($10+\delta_1$) to its Tfork process where δ_1 is the service time of the corresponding queue *pp*. This would be followed by a NULL($10 + \delta_1$) message sent to Pmerge P2, a NULL($10 + \delta_1$) message to queue P2, a NULL($10 + \delta_1 + \delta_2$) message to zero-place P2 where δ_2 is the service time of queue *pp* P2, and finally a NULL($10 + \delta_1 + \delta_2$) message to zero-transition T2. At this point the zero-transition T2 updates its value of CT[T2] to be $10 + \delta_1 + \delta_2$ and sends a message *answer($10 + \delta_1 + \delta_2$)* to zero-transition T1. Using assumptions similar to those in [Chandy and Misra 1979], we will have $\delta_1 + \delta_2 > 0$, and therefore the zero-transition T1 will fire.

In general, in our approach a zero-transition i would send out the message NULL(CT_i) to its output queue process when its CT_i value is greater than the time component of the last message sent to this queue process. On receiving a NULL(t) message from an input zero-place k, the value of $CP_i[k]$ is updated to t, and CT_i is updated to t if t is greater than the current value of CT_i. A zero-place *lp* receives and sends NULL messages in the obvious way. However, after a zero-place *lp* has sent out *tokavail(...)* messages for a token in hand, it will not send out any further *tokavail(...)* or NULL(...) message until that token has been grabbed.

Other processes receive and generate NULL(...) messages in the obvious way similar to that in [Chandy and Misra 1979; Kumar 1989]. In particular, note that a queue process may generate an output NULL(...) message even without receiving a corresponding input NULL(...) message. The number of NULL messages can be reduced by incorporating a few simple strategies, e.g., (i) if an input NULL message is followed by another input message then ignore this NULL message, and (ii) do not send out a NULL(t) message on an output line if the previous message sent on the line has time component = t.

Note that the meaning of NULL message in our approach is a bit different from that in [Chandy and Misra 1979]. In [Chandy and Misra 1979] a NULL message on a line with time stamp t means that the next message on this line will have time stamp "strictly greater than" t, whereas in our approach it will be "greater than or equal to" t. To understand the motivation why we chose to define NULL messages with a different meaning, consider again Figure 1 where a token is available at the output of zero-place P1 at time zero, and at the output of zero-place P4 at time 10. In the approach of [Chandy and Misra 1979], the zero-transition T1 cannot send out a NULL(10) message (since it does not know whether it would fire at time 10); it would have to choose a time t<10 and send out a NULL(t) message. If this chosen t is too small, i.e., if $t + \delta_1 + \delta_2 < 10$ where δ_1 and δ_2 are the service times at the queues at T1 and P2, then deadlock will not get resolved.

In general, in our simulation we do not require that two successive messages *tokavail(...)* or NULL on a given line carry strictly increasing time component (which is required in [Chandy and Misra 1979]). This has several advantages: (i) It makes coding easier, since in the approach of [Chandy and Misra 1979] if a Pmerge process receives several input tokens with the same time component (on the same or different input lines), then it has to combine them together into a single composite token message. Thus all *lps* have to deal with composite token messages. In our approach any message involving tokens would carry information about only one token, and therefore these *lps* would deal with one input token at a time, simplifying the code. (ii) It is somewhat more efficient, since more information is being carried in NULL messages as discussed in the previous example.

6.3 Further Considerations In Our Approach

Note that in a direct application of [Chandy and Misra 1979], a transition would look at all its input and output lines and then advance its clock value to the minimum of the line clocks. Input message histories known

beyond this clock value are ignored in the computation of the output messages. This approach would normally generate too many NULL messages. For example, consider a zero-transition which has two input lines. Suppose that it receives a token message with time stamp 10 on line 1, and a NULL message with time stamp 5 on line 2. In the approach of [Chandy and Misra 1979], the zero-transition *lp* in this case will send out a NULL message with time stamp 5. Suppose next it receives a NULL message with time stamp 7 on line 2. Then it will send another NULL message with time stamp 7. Similarly, more NULL messages would be sent if more NULL messages arrive on line 2 with time stamp less than 10. On the other hand, in our approach the time clock of the process simulating a zero-transition is advanced to the *maximum* clock of the input lines. Thus, after receiving the token message with time stamp 10 on line 1 and the NULL message with time stamp 5 on line 2, the zero-transition *lp* will advance its clock to time 10 (and not to time 5 as in the approach of [Chandy and Misra 1979]) and will send out a NULL message with time stamp 10. Subsequently, on receiving the NULL message with time stamp 7 on line 2, no further NULL message will be generated. Thus our approach of using *maximum* of input line clocks reduces the number of NULL messages.

Our approach of using the *maximum* of input line clocks also increases the degree of concurrency. For example, if a zero-transition has received an input token on each of its input lines, with time stamps 10 and 5 respectively, and if it has no conflict with its neighbors, it can go ahead and fire with simulation time 10. In the approach of [Chandy and Misra 1979] this *lp* would have to wait and receive more messages until the clock value of the second input line also reaches time$\geq$10. This would be inefficient and would also make the code more complex since now more messages must be sent, received, and explicitly stored by the receiver *lp* for the second line.

Similarly, in [Chandy and Misra 1979], there are other unnecessarily rigid rules as to when an *lp* is allowed to send or receive tuples. In particular, an *lp* sends a tuple on a line only when the line clock for that line becomes minimum among all clock values of all the lines adjacent to the *lp*. This can obviously affect the degree of concurrency. For example, consider a delay *pp* that processes incoming jobs in the FCFS order [Kumar 1989b] which received a tuple (10,m) and sent out a tuple (15,m). Suppose next it receives a tuple (13,m). Now this *lp* is not allowed to process and send this job until the input clock becomes 15. This affects the degree of concurrency between this *lp* and the *lp* connected to its output.

Also, the scheme [Chandy and Misra 1979] is based on the synchronous message communication of Hoare's CSP [Hoare 1978]. This can cause a minor performance degradation when implemented on an asynchronous system.

In summary, our approach differs from [Chandy and Misra 1979] in several ways: (i) Handling non-autonomous nature of a zero-place process and conflict resolution among neighboring zero-transitions, (ii) Use of NULL messages with slightly different meaning, and not requiring *strict* chronology of time-stamps on a line, which would make coding simpler and would be more efficient, (iii) The time clock of the process simulating a transition is incremented to the *maximum* clock of the input lines, resulting in less number of NULL messages, more concurrency, and simpler code, (iv) An *lp* is allowed to send messages whenever possible, rather than having to wait for inputs simply because the current output line clock is higher than the minimum of input line clocks, (v) Message communication in the logical system is assumed to be asynchronous, and (vi) Infinite input buffers are assumed at the input port of any *lp*, to provide higher degree of concurrency.

7. CONCLUDING REMARKS

In this paper we presented a model of Timed Petri Net and an approach towards its distributed simulation. In this model time can be associated with either a place or a transition or both. The model consists of several simple modules (i.e., the processes *Pmerge, queue, zero-place, zero-transition* and *Tfork*) resulting in modular code for simulation. This modularity makes it easy to modify the code in the future. Also our model is fairly general in the sense that the *queue* processes can be defined in many ways.

As mentioned earlier, the distributed simulation schemes of [Chandy and Misra 1979, 1981; Jefferson and Sowizral 1982] do not apply to the simulation of these systems directly. We presented an approach for the distributed simulation of these systems. In this approach, neighboring transitions communicate with each other to determine which of them grabs a token that arrived at a common input place. The possibility of deadlocks is avoided by using a slight variation of the idea of NULL messages as proposed in [Chandy and Misra 1979]. We exploited special properties of Timed Petri Nets to reduce the number of NULL messages and to increase the degree of concurrency, as discussed in section 6.

REFERENCES

Alanche, P., K. Benzakour, F. Dolle, P. Gillet, P. Rodrigues, and R. Valette (1985), "PSI: A Petri Net Based Simulator For Flexible Manufacturing Systems", *Lecture Notes in Computer Science* 188, 1-14.

Andrews, G.R. and F.B. Schneider (1983), "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, 15, 1.

Baik, D. and B. Zeigler (1985), "Performance Evaluation of Hierarchical Distributed Simulators", *1985 Winter Simulation Conference Proceedings*, 421-427.

Bhargava, B. (1982), "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking", *Proceedings of the 2nd International Conference on Distributed Computing Systems*, Miami, FL, 508-517.

Bryant, R.E. (1977), "Simulation of Packet Communication Architecture", M.S Thesis, Department of Computer Science, Massachusetts Institute of Technology, Boston, MA.

Bryant, R.E. (1984), "A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE trans. on Computers*, C-33, 2, 160-177.

Carlier, J., P. Chretienne and C. Girault (1984), "Modeling Scheduling Problems with Timed Petri Nets", *Advances in Petri Nets*, 62-82.

Chandy, K.M. and J. Misra (1979), "Distributed Simulation: A Case Study In Design And Verification of Distributed Programs", *IEEE Transactions on Software Engineering* 5, 5, 440-452.

Chandy, K.M. and J. Misra (1981), "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM* 24, 4, 198-205.

Chandy, K.M. and J. Misra (1984), "The Drinking Philosophers Problem", *ACM Transactions on Programming Languages and Systems*, 6, 4, 632-646.

Chretienne, P. (1983), "Les Resaux de Petri Temporises", These d'etat, University de Paris VI, Paris, France.

Coolahan, J.E. (1983), "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets", *IEEE Transactions on software Engineering* 9, 5, 603-615.

Davidson, D. and P. Reynolds (1983), "Performance Analysis of Distributed Simulation Based on Active Logical Processes", *1983 Winter Simulation Conference Proceedings*, 267-268.

Dijkstra, E.W and C.S. Scholten (1980), "Termination Detection for Diffusing Computations", *Information Processing Letters*, 11, 1.

Fujimoto, R.M., J.J. Tsai, and G.C. Gopalakrishnan (1988a), "Design and Performance of Special Purpose Hardware for Time Warp", *Proceedings of the 15th Annual Symposium on Computer Architecture*, 401-408.

Fujimoto, R.M. (1988b), "Lookahead in Parallel Discrete Event Simulation", *Proceedings of the 1988 International Conference on Parallel Processing*, 34-41.

Fujimoto, R.M. (1989), "Time Warp on a Shared Memory Multiprocessor", *Proceeding of the 1989 International Conference on Parallel Processing*, 242-249.

Gafni, A. (1988), "Rollback Mechanisms for Optimistic Distributed Simulation Systems", *In Distributed Simulation*, Society for Computer Simulation, 61-67.

Garg, K. (1985), "An Approach to Performance Specification of Communication Protocols Using Timed Petri Nets", *IEEE Transaction on Software Engineering*, 11, 10, 1216-1225.

Hoare, C.A.R. (1978), "Communicating Sequential Processes", *Communications of the ACM*, 21, 8, 666-777.

Groselj, B. and C. Tropper (1987), "Pseudosimulation: An algorithm for distributed simulation with limited memory", *Int. J. Parallel Programming*, 15, 5, 413-456.

Jefferson, D.R. and H.A. Sowizral (1982), "Fast Concurrent Simulation Using The Time Warp Mechanism, Part I: Local Control", Technical Report, The Rand Corporation, Santa Monica, CA.

Jefferson, D.R. Jefferson and H.A. Sowizral (1985), "Fast Concurrent Simulation using the time Warp mechanism", in *Distributed Simulation 1985, The 1985 Society for Computer Simulation Multiconf.*, San Diego, CA.

Jefferson, D.R. (1985), "Virtual Time", *ACM Transactions on Programming Languages and Systems* 7, 3, 404-425.

Joyce, J., G. Lomow, K. Slind, and B. Unger (1987), "Monitoring Distributed Systems", *ACM Transactions on Computer Systems*, 5, 2.

Kravitz, S.A., R.E. Bryant, and R.A. Rutenbar (1989), "Massively Parallel Switch-Level Simulation: A Feasibility Study", 26^{th} *ACM/IEEE Design Automation Conference*, 91-97.

Krishnamurthy, M., U. Chandra, and S. Sheppard (1985), "Two approaches to the implementations of a distributed simulation system", *Proceedings of the 1985 Winter Simulation Conference*, San Francisco, CA, 435-443.

Kumar, D. (1989a), "Systems Whose Distributed Simulation Requires Low Overhead", 8^{th} *Annual IEEE International Phoenix Conference on Computers and Communications*, Scottsdale, Arizona.

Kumar, D. (1989b), "Correctness Proof of a Distributed Simulation Scheme", *First Annual IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 49-56.

Kumar, D. (1989c), "An Approximate Method to Predict Performance of a Distributed Simulation Scheme", 18^{th} *International Conference on Parallel Processing*, St. Charles, IL, 259-262.

Kumar, D. (1990), "An Algorithm for N-Party Synchronization Using Tokens", 10^{th} *International Conference on Distributed Computing Systems*, Paris, France.

Lamport, L. (1978), "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21, 7, 558-565.

Lonow, G. and B. Unger (1982), "Process View of Simulation In ADA", in *1982 Winter Simulation Conference*, 77-86.

Lubachevsky, B.D. (1987), "Efficient parallel simulations of asynchronous cellular arrays", *Complex Systems*, 1, 6, 1099-1123.

Lubachevsky, B.D. (1988), "Bounded lag distributed discrete event simulation", *Proceeding of the 1988 SCS Multiconference*,19, 3, 183-191.

Lubachevsky, B.D. (1989), "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communication of the ACM*, 32, 1, 111-123.

Madisetti, V., J. Walrand, and D. Messerschmitt (1988), "WOLF: A Rollback Alogorithm for Optimistic Distributed Simulation Systems", *1988 Winter Simulation Conference Proceedings.*

Misra, J. (1986), "Distributed Discrete-Event Simulation", *ACM Computing Surveys* 18, 1, 39-65.

Ozsu, M.T. (1985a), "Modeling and Analysis of Distributed Database Concurrency Control Algorithms Using an Extended Petri Net formalism", *IEEE Transactions on Software Engineering*, 11, 10, 1225-1240.

Ozsu, M.T. (1985b), "Performance Comparison of Distributed vs. Centralized Locking Algorithms in Distributed database System", *In Proc. 5th International Conference on Distributed Computing Systems*, Eds. IEEE, Piscataway, NJ, 254-261.

Ozsu, M.T. (1987), "Distributed Simulation using Petri Nets", *19th Annual Summer Conference On Computer Simulation*, 3-8.

Peacock, J.K., J.W. Wong and E.G. Manning (1979a), "Distributed Simulation Using A Network of Processors", *Computer Networks* 3, 1, 44-56.

Peacock, J.K., J.W. Wong and E.G. Manning (1979b), "A Distributed Approach To Queuing Network Simulation", in *Proc. 4th Berkeley Conf. on Distributed Data Management and Computer Networks*, Berkeley, CA, 237-259.

Peterson, J.L. (1981), "Petri Net Theory and Modeling of Systems", Prentice-Hall.

Ramchandani, C. (1974), "Analysis of Asynchronous Concurrent Systems by Petri Nets", Technical Report 120, MAC, MIT, Boston, MA.

Reed, D.A. (1983), " A Simulation Study of Multimicrocomputer Networks", *Proceedings of International Conf. on Parallel Processing*, 161-163.

Reed, D.A. (1985), "Parallel Discrete Event Simulation: A Case Study", *Record of Proceedings: 18th Annual Simulation Symposium*, 95-107.

Reed, D.A., A.D. Malony, and B.D. McCredie (1988), "Parallel Discrete Event Simulation Using Shared Memory", *IEEE Transactions on Software Engineering*, 14, 4, 541-553.

Reynolds, P. (1982), "A Shared Resource Algorithm for Distributed Simulation", *Proceedings of the 9th International IEEE Architecture Conference*, Austin, TX.

Sauer, C.H. and E. A. MacNair (1983), Simulation of Computer Communication Systems, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Schneider, F.B. (1982), "Synchronization in Distributed Programs", *ACM Transactions on Programming Languages and systems*, 4, 2, 125-148.

Seitz, C.L. (1985), "The cosmic cube", *Communication of ACM*, 28, 1, 22-23.

Wieland, F. and D. Jefferson (1989), "Case Studies in Serial and Parallel Simulation", *Proceeding of the 1989 International Conference on Parallel Processing*, St. Charles, IL, 255-258.