Design and Implementation of Distributed Time Petri-Nets (TPNs) Simulator in the RAS/Tools Framework

Xiaomao Liu, Yi Deng, Xudong He
School of Computer Science
Florida International University
Miami, FL 33199
e-mail: {xiaomao, deng, hex}@cs.fiu.edu

ABSTRACT

The Real-time Architectural Specification (RAS) based on Time Petri-Nets (TPNs) formal method provides a modeling approach to support design and development of reliable real-time distributed systems. In this paper, we present a distributed TPNs simulation algorithm and its clustering implementation for RAS model simulation. The distributed algorithm and implementation can observably simplify the complexity of RAS model structure and improve the performance of system simulation. Two issues focus on the distributed TPNs simulation algorithm: (1) how to get an efficient distributed simulation algorithm of the Time Petri-Nets, and (2) how to design the time agreement of multiple TPNs simulations in distributed system. In addition, we propose a gang schedule strategy that effectively maps multiple simulation threads onto NT-cluster. The distributed simulation algorithm, as a core module of the RAS/Tools framework, is implemented by Java class (Jbuilder2.0) and distributed CORBA class (VisiBroker2.0) in an NT workstations-cluster.

Keywords: Time Petri-Nets (TPNs), Time Warp, Distributed Simulation, Real Time Architectural Specification (RAS), Gang Scheduling Scheme, CORBA, and NT-Cluster Computing

1 INTRODUCTION

Petri-Nets simulation is an audio-visual and straightforward approach for modeling, especially when exhaustive analysis is unusable due to the size and complexity of the model. But, checking a large Petri nets simulation is probably quite difficult. So, some researches have been done [1-3] to apply distributed simulation technology to Petri nets or its extended mode, i.e. it allows to partition a given model into several small function components that is easy to analyze and execute these chunks on a distributed system.

Component technology is an important means to alleviate the costly rediscovery and re-invention of complex software. Components are useful for documenting recurring micro-architecture, which are abstractions of common and existing software model that expert developers apply to solve more complex software problems. By studying and applying components, developers can often avoid traps and pitfalls that have traditionally been learned only by prolonged trial and error. On the other hand, components are basic concurrent execution units for the logic individual processing nodes of a distributed system. The components executed in concurrent or distributed manner can potentially simplify the complexity of model structures and improve the performance of model simulation experiments for large system models.

Real-time Architectural Specification (RAS) approach developed in FIU [4-8] is a formal technique to support design and development of highly reliable real-time distributed systems. RAS model is built on top of Time Petri-Nets. A large and complex RAS model is a Time Petri-Nets of hierarchical and multiple components that describe the core behaviors, processing functions and requirements for real-time system. We have used RAS to define and analyze an intelligent manufacturing system [4], a control and command system [5], and several well-known communication connectors [6].

Chiola and Ferscha [1-3] are among the first ones to introduce the distributed simulation and Time Warp technology into Ramachandani’s Timed Petri-Nets [12]. But, according to Menasché’s analysis, transitions in a Timed Petri-Nets must fire as soon as they are enabled. This may cause some ambiguities when an enabled transition is disabled by the firing of another one [10]. Among Petri-Nets and its extended models, Merlin’s Time Petri-Nets (TPNs) [11] is a most widely used model for real-time system specification and implementation as it can model real-time
constraints in a more natural way. Compared with Timed Petri-Nets, TPNs’ timing constraints are expressed in terms of minimum and maximum amount of time elapsing between the enabling and the execution of model’s each action. It is well suited for a compact representation of the state space and an explicit modeling of real-time system [7].

In this paper, we present a distributed TPNs simulation algorithm for RAS model simulation. Two issues are critical in the algorithm: first, how to define a sole component simulation that is allowed parallel and distributed handling efficiently on distributed system; and second, how to solve the time synchronization of the multi-component simulations running simultaneously on distributed system. With JBuilder (Java class) and VisiBroker (CORBA-based software) we have implemented the RAS/Tools environment to support the distributed TPNs simulation algorithm on NT-cluster. Some practical RAS models, such as C4I and FMS system, have shown to improve the complexity of model analysis and performance of system simulation in the RAS/Tools system.

The rest of the paper is organized as follows: an introduction of system model, including TPNs and RAS model, is given in Section 2. In Section 3, we propose a distributed TPNs simulation algorithm that includes the component simulation, co-ordinator, and rollback recovery and roll-forward algorithms. In Section 4, we introduce an implementation of the distributed algorithm in RAS/Tools. Finally, we give a discussion on future work of the distributed algorithm in Section 5.

2 SYSTEM MODEL

This section gives a definition of Time Petri-Nets and introduces a notion about RAS model [8-11], which is the analysis basis of the distributed TPNs simulation algorithm.

2.1 Time Petri-Nets Model

Merlin’s TPNs is a pair \((PN, SIM)\), where \(PN\) is a tuple \((P, T, B, F, M_0)\) defining a conventional Petri-Net. The follow notations are used for any given TPNs model.

- \(P\): a finite nonempty set of places.
- \(T\): a finite nonempty, set of transitions \(t_i\), it is convenient to view it as an orders set \(\{t_1, t_2, \ldots t_n\}\).
- \(B\): the backward incidence function
  - \(B: T \times P \rightarrow N\)
  - Where \(N\) is the set of nonnegative integers.
- \(F\): the forward incidence function
  - \(F: T \times P \rightarrow N\)
- \(M_0\): the initial marking function
  - \(M_0: P \rightarrow N\)
- \(SIM\): a function that associates an interval of non-negative real numbers \([a_i, b_i]\) to each \(t_i \in T\).

The interval \([a_i, b_i]\) is called Static Firing Interval of the transition \(t_i\). There are no restrictions to the upper and lower limits of this interval, except the fact that \(a_i \leq b_i\). \(a_i\) is named the static Earliest Firing Time (EFT) and \(b_i\) is named the static Latest Firing Time (LFT).

Transitions of TPNs model are enabled the same way as in conventional Petri nets, but the firing of an enabled transition \(t_i\) will only happen in a time \(SIM(t_i)\) within limits defined by its Static Firing Interval, relative to the moment \(t_i\) was enabled. The firing of a transition in a TPNs is instantaneous and has the same effect as in a conventional Petri nets.

2.2 RAS: A Hierarchical and Multi-Components TPNs Model

We define an RAS model as a structure that consists of such basic elements: TPNs component models and inter-component connections. The component is a TPNs model that describes the real time behavior, processing functions and communication interface of the components. The connections are TPNs transitions that specify how the components interact with each other and, in turn, form the system composition model.
(e.g., port7) and output ports (e.g., port6), and (2) a TPNs that describes the time-dependent, operational behavior of the component, that is, it defines the semantics associated with the ports. In fact, the port is a TPNs place that is merely located on the component’s edges. The input ports connect to component’s internal time transitions and the output ports connect to component’s external time transitions. The communication between a component and its environment is solely through the ports. The component’s external time transitions (link transitions) are defined as inter-component connections or components channel. Through these transitions a component implements to connect with other components.

RAS model is a hierarchical and multi-component structure. One major characteristic of the hierarchical and multi-component TPNs model is that it can integrate several existing functional components to form a more complex real-time system model, which increases the reuses of the components. Another useful feature is that it is suitable for distributed simulation implementation, which decreases the complexity of entire model and improves the system performance when a simulation’s state space becomes larger.

3 DISTRIBUTED TPNs SIMULATION ALGORITHM

A large RAS system model is statically partitioned into several layers (sub-systems) that include groups of smaller components (processing function descriptions). According to RAS modeling approach there are no any data-dependent relationship among model’s layers. All layers’ simulations can be executed independently or simultaneously without conflicts. So the distributed TPNs simulation algorithm we present here is localized one layer’s simulations.

3.1 Component Simulation Algorithm

As showed in Figure 1, in general, one RAS layer consists of several components. We define simulation of component is the basic computation region of distributed TPNs simulation. The component simulation algorithm comprises the following steps:

(1) Initialize three transition sets: Disabled Transition Set (DTS), Enabled Transition Set (ETS), and Modified Transition Set (MTS), put all transitions into MTS, let DTS and ETS be empty. Then, initiate Local Clock (LC) of components and Last Firing Time (LFT) variable. Next, save the initial marking of TPNs component to M₀.

(2) Check all of the transitions t(i) (i=1,2,…n) in MTS and ETS. If t(i) is enabled then put it into ETS, otherwise put it into DTS; if t(i) is in ETS then update its firing interval [a,b] to [max{0,a-LFT}, b-LFT], if not, reset its firing interval value.

(3) Check M_k. If M_k = M₀ then stop. Next, send the message COMPONENT_END to the co-ordinator (time arbitrator).

(4) For t(i) ∈ ETS, generate a random firing time δ_i (a_i ≤ δ_i ≤ b_i), here [a_i, b_i] is the Static Firing Interval (SFI) of t(i). Find a transition T₀ that has the minimal δ_i. Set LFT = Min(δ₁, δ₂,…δ_k).

(5) Update Local Clock: LC = LC + LFT, and let MTS be empty.

(6) For all in output places of the transition T₀, generate an event TPKEN_PASSED_EVENT using for rollback recovery or roll-forward procedure and put the event into an event queue QUEUE. Then put the output transitions of these places into MTS.

(7) Firing the enabled T₀ at time LFT leads to a new making for all places p.

\[ M_k(p) = M_k-1(p) - B(T₀, p) + F(T₀, p). \]

If the output of T₀ is a port, then send the message TOKEN_REQ to the co-ordinator. Go to (2).

Obviously, LC is same as Local Virtual Time (LVT) and QUEUE same as Event List (EVL) in Ferscha’s Time Warp algorithm [1-3]. But, Ferscha defines a Global Virtual Time (GVT) in his optimistic strategy and simulation engine. Why don’t we introduce a global absolute clock or Global Virtual Time in the component simulation algorithm? A conclusion will be got in next section.

3.2 The Co-ordinator Algorithm

Every thread implemented the component simulation procedure solely keeps a Local Clock (LC) which records its time processes and contains the time-stamps of last handled event. Because of several components running in one RAS layer, the system simulation is a concurrent execution of multi-threads. The co-ordinator, a global time arbitrator, is chosen to coordinate the execution order of these simulation threads, and a master thread is allocated to the co-ordinator for checking time agreement and being in charge of token exchanges among the components. There are two types of consistency checking in the co-ordinator: one is the input-input checking that evaluates the time agreement among the input components of link transition; the other is the input-output checking that evaluates the time consistency between the input and output components of link transition.

3.2.1 Input-Input Checking

As show in Figure 2, there are n input components linking to transition t through its outer ports, let the enabled clock of t be the synchronous time of arriving tokens for all of input components. Assume the tokens of input components reach its outer ports linked to t at Local Clock \( T_i, i=1,2…n \)
respectively. We use $S$ to denote the enabled clock of $t$. To meet the time consistency of arriving tokens for the input components in $t$, $S$ must satisfy the following constraint:

$$S \geq \text{Max } \{T_1, T_2, T_3...T_n\}$$

Moreover, if there are several output components linking to $t$ in Figure 3, extend the algorithm to compare the Local Clocks of output components with $T_1$ one by one, and then inform each output component whether the rollback or roll-forward procedure should be invoked. Finally, $t$ will be fired when all of synchronous conditions are satisfied for current Local Clocks.

In Ferscha’s algorithm he doesn’t use roll-forward iterations and time arbitrator. The co-ordinator is a global time arbitrator among Local Clocks of simulation threads. According to the algorithms mentioned in previous sections, based on roll-forward/rollback recovery and time arbitrator, it is straight that the algorithm always finds a suited time agreement among Local Clocks of components when simulation running. So, certainly, we don’t need to set Global Virtual Time (GVT) in the distributed TPNs simulation algorithm.

### 3.3 Distributed TPNs Simulation Algorithm

The distributed TPNs simulation algorithm consists of two parts: (1) the component simulation with roll-forward and rollback iterations (clients), and (2) the co-ordinator (server), which implemented with Java threads in client/server mode. Besides, a middle-ware encapsulated VisiBroker CORBA API is applied to support gang threads scheduling and data communication between the clients and server.

Figure 4 illustrates the communication protocol of tokens passing. Here, thread $P_i$ acts as the co-ordinator. $P_i$ is the $i^{th}$ component thread and $P_j$ the $j^{th}$ component thread. If $P_i$ requests to send tokens to $P_j$, it sends $M_1$ (TOKEN_REQ) to $P_0$ first. When $P_0$ gets $M_1$, it sends $M_2$ (CLOCK_REQ) to $P_i$ for asking its Local Clock $T_i$. Then, $P_i$ replies $M_3$ included the LC to $P_0$. $P_0$ makes a computation to decide the rollback, roll-forward or stop action will be executed at $P_i$ when it receives $M_4$, and then sends $M_5$ (NEED-ROLL-BACK, NEED-ROLL-FORARD or STOP) to $P_i$. After receiving $M_6$, $P_j$ invokes the Rollback recovery or Roll-forward

Figure 2. Synchronization of Input-Input Components

The earlier arriving tokens will wait at $t$ until last one occurs. But, whether $t$ is fired immediately or not, it still depends on the time agreement with its output components. We will present a brief analysis for it in the next section.

### 3.2.2 Input-Output Checking

As shown in Figure 3, for simply, assume the transition $t$ connect to an input component $P_i$ and an output component $P_j$, and the Local Clock of component $P_i$ be $T_i$ ($i = 1, 2$). In general, $t$ should be enabled at the local clock $T_i$ when a token just arrives at the outer port of $P_i$. But, $t$ can’t be fired immediately because we don’t know whether the LC is matchable between $P_i$ and $P_j$. Otherwise, the tokens of $P_i$ can’t be past to $P_j$. To assure time synchronization in the input and output components of $t$, the firing time of $t$ should satisfy the following synchronous conditions:

$$T_2 + EFT \leq T_1 + \theta \leq T_2 + LFT.$$  
$$EFT \leq \theta \leq LFT.$$  

We set such iteration in the co-ordinator:

```java
while (T2 doesn’t satisfy synchronous condition) do {
    if (T1 + LFT < T2) then inform P2 to call rollback Recovery;
    else if (T1 + EFT > T2) then inform P2 to call roll-forward iterations;
}

Send the STOP message to P2, then t is fired at T1 + \theta;
```

Figure 3. Synchronization of Input-Output Components
procedure if necessary, then \( M_3 \) (CONTINUE) is sent back to \( P_0 \). It causes \( P_0 \) to repeat previous steps. Finally, \( M_5 \) (START_SEND) is sent to \( P_i \) to start tokens passing, and \( M_7 \) (COMPONENT_END) is sent to \( P_0 \) when \( P_j \) receives the tokens from \( P_i \).

Two running restrictions are made additionally in protocol implementation. First, the co-ordinator or component simulation threads deliver messages within a limit time (< 200 ms, message propagation time); and second the co-ordinator or component simulation threads also respond the message within a limit time (< 300 ms, message-handling time). And, one sending message causes a confirmed ACK replied by receiver. If overtime occurs in sending or waiting data packages the packages will be asked to deliver again.

3.4 Roll-forward Iterations and Rollback Recovery

Because several threads execute the \textit{component simulation algorithm} simultaneously, it certainly appears a time difference in Local Clock of components. Some simulators run fast and some slow. We need to adjust the Local Clock of components continuously in order to maintain the time agreement when token exchange occurs among component simulators.

Roll-forward iterations are applied here if a component’s Local Clock lags the another component’s and the “fast” component just prepare to send tokens to the “slow” one. Assume the component that sends or receives tokens is \textit{sender} or \textit{receiver} respectively. The \textit{co-ordinator} checks the two components’ Local Clock and informs the \textit{receiver} whether it needs to call the roll-forward procedure or not. The \textit{roll-forward iterations} that continues to run forward for several steps in the “slow” \textit{receiver} is same as the \textit{component simulation algorithm} except those updates the former algorithm’s step (5) as following:

(5) Update Local Clock: \( LC = LC + LFT \). And, set MTS to empty. Then, send Local Clock (LC) to the co-ordinator.

After receiving the Local Clock the co-ordinator compares it with the \textit{sender}’s Local Clock, then it replies to \textit{receiver} and informs the roll-forward iterations should continue or stop.

On the contrary, if the \textit{sender}’s Local Clock lags the \textit{receiver}’s, the rollback recovery is invoked at the \textit{receiver}. The \textit{rollback recovery} algorithm depends on the event \textit{TOKEN_PASSED EVENT} and its queue QUEUE. An event is a checking-point that keeps the tracing info for the backtracking of simulation program. We define the event data structure as the following 4-tuple:

\[
\text{EVENT (source place/transition pairs, target place/transition pairs, number of token passed, current LC)}
\]

The \textit{rollback recovery} works in reversing iterations of the \textit{component simulation algorithm}. QUEUE records the entire token markings of component according to the execution order of simulation iterations. So we can restore continuously previous token marking of the component by setting tokens back from target places to source places with QUEUE. During iterations, component thread processes event pop from QUEUE, and rolls back to its previous token marking according to current event’s info. Then, it sends Local Clock to the co-ordinator. The iterations will continue until time synchronous condition is satisfied in the co-ordinator.

4 IMPLEMENTATION DETAILS IN RAS/TOOLS

In the following section we describe the implementation of the \textit{distributed TPNs simulation algorithm} and its mapping procedure in NT-cluster.

4.1 The RAS/Tools Framework

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{architecture.png}
\caption{Architecture of the RAS/Tools}
\end{figure}

4.2 Implementation of Distributed TPNs Simulation Engine

RAS/Tools, its architecture showed in Figure 5, is a framework that implements the simulation and analysis tools (model validation and model checking) of RAS model. The system offers a user-friendly interface (GUI) to build and access TPNs models, simulation engine and analysis tools through multi-compositions/components windows. Model database stores basic TPNs models, simulation data, results of model validation and analysis, and so on. The core of the simulator engine is a TPNs simulation iteration algorithm. RAS/Tools provides reusable implementations of the following software tasks:

- Multi-windows GUI
- Simulation engine
- Models database (MS InterBase)
- Inter-threads communication and management
- Barriers synchronization
- Threads gang scheduling
- Analysis tools

The RAS/Tools environment supports two kinds of simulation engine, which include the traditional TPNs simulation and distributed versions.

We have implemented the distributed TPNs simulation algorithm with Java thread and VisiBroker (CORBA) functions in the simulator engine. Java multi-thread application only supports the concurrent execution of multiple threads in one machine. But, by employing the VisiBroker’s Smart Agent to support remote thread communication, Java threads can be distributed actively at NT workstations-cluster, i.e. any thread supported by such VisiBroker CORBA methods can be as a distributed object to execute remotely by other workstations in NT-cluster.

The Smart Agent (osagent) is a dynamic, distributed directory service that provides facilities used by both client programs and server implementations. As showed in Figure 6, the client program that registers with the Smart Agent can connect to server object on different local networks and migrate from one workstation to another. When the client program invokes the VisiBroker method bind (...) in a thread, the Smart Agent is automatically consulted. The Smart Agent locates the specified server so that a connection can be established between the client and the server. The communication with the Smart Agent is completely transparent to the client program.

![Figure 6. Running Java threads on NT-cluster](image)

We develop distributed TPNs simulation program as the following distributed programming (clients-server mode) style:

1. Write a specification for the co-ordinator (as server) and component simulators (as clients) using Interface Definition Language (IDL). For example, the contents of simulation engine in the Simulation.idl file is created as below:

```java
interface Co-ordinator {
    float getClock();
    Thread create();
    in receive();…
};
```

The Thread interface defines the methods used in component simulation. The Co-ordinator interface manages the thread sessions (such as creates a Thread for the user if one doesn’t already exist) and specifies the operations used in communication with the simulation threads.

2. The IDL interface specification is compiled by VisiBroker idl2java to generate stub code for the client program (simulation) and skeleton code for the server program (co-ordinator) respectively.

3. Write the simulator (client-side) code that implements the client application. The above example’s client program performs these steps:
   - Initializes the Object Request Broker (ORB).
   - Binds to the co-ordinator object.
   - Send a connection message to the co-ordinator.

4. Write the co-ordinator (server-side) code. The server objects implement the server application. The above example’s server program does the following:
   - Initializes the ORB.
   - Creates a Portable Object Adapter (POA) with the required policies.
   - Creates the co-ordinator object.
   - Activates the POA manager (and the POA).
   - Waits for client’s incoming requests.

Here, a POA is the intermediary between the implementation of an object and the ORB. In its role as an intermediary, a POA route requests to servers and, as a result may cause servers to run and create child POAs if necessary.

5. Compile the simulator and server code. To create the TPNs simulation program, compile the client-side program code with the client stub; and to create the co-ordinator program, compile the server-side code with the server skeleton.

6. Start the process co-ordinator and then run the simulators on NT-cluster.

### 4.3 Gang Schedule Scheme

As mentioned in previous sections, there are several concurrent component simulation threads running on one RAS layer. A Java thread is indicated to carry out one
component simulation in simulation engine. We need to map threads to NT-clusters in a way balancing the system load if thread numbers exceed cluster nodes. Otherwise, the loading balance problem will be a system performance “bottleneck”. In user-level we have updated a “gang-scheduling” strategy [14] for allocating simulation threads onto an NT-cluster. “Gang scheduling on NT-cluster” refers to all of a model’s threads being grouped into a gang and currently scheduled on NT-cluster. Let M be the cluster nodes and POOL the global shared thread pool included a wait-queue and a running-queue. We simply describe the gang-scheduling algorithm as the following steps:

1. Get first M threads from running-queue and co-assign them into the M nodes of NT-cluster respectively.
2. After a time slice (1500ms), pause current threads running, and switch them into wait-queue, then move M threads from wait-queue into running-queue.
3. Go to (1) unless no thread in POOL, SUSPENDING.

The gang mapping procedure is active when the following scheduler code is running at NT-cluster:

\[
\begin{align*}
N &= \text{thread numbers}; \\
\text{if}(N \neq 0) \{ & \quad // \text{gang size } = M \\
L &= \text{int}(N \mod M); & \quad // \text{number of gang } = L \\
\text{for}(i = 0; i < L; i++) \{ & \quad \\
\text{Switch current gang threads to waiting-queue; } \\
\text{Pick up new gang threads from running-queue; } \\
\text{Map new gang threads onto NT-cluster; } \\
\text{Sleep (TIME_SLICE); }
\}
\end{align*}
\]

Gang scheduling of a job’s multi-threads has been shown to improve the efficiency of both the individual distributed program and the system [14]. Comparison with the same models built by traditional TPNs algorithm in RAS/Tools, as shown in the following table, the distributed method evidently improves the performance of model simulation, especially on token moving.

<table>
<thead>
<tr>
<th>Time Unit: sec</th>
<th>Non-Distributed</th>
<th>Distributed (2 nodes)</th>
<th>Distributed (4 nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution time</td>
<td>251.20</td>
<td>189.00</td>
<td>101.00</td>
</tr>
<tr>
<td>speed up</td>
<td>1.329</td>
<td>1.104</td>
<td>1.760</td>
</tr>
</tbody>
</table>

*: S-C2 program  
#: S-FMS program

Table. Some Experiment Results on NT-cluster

But, in some practical simulations of RAS models (such as C1T and FMS model), we may synchronize the actual token exchange into visualizing it on GUI. Because these simulations are no-stop loops, faster token may cause it is difficult to catch its trace up. To slow down token moving, we have to add sleep function or step-by-step mechanism in simulation engine.

5 FUTURE RESEARCH

Workstations-clusters have become important parallel and distributed platforms for large-scale scientific and simulation computation. But, most of current workstations-clusters use UNIX as its workstation’s OS environment. We have described in this paper a distributed TPNs simulation algorithm for RAS model and shown an implementation in NT-cluster. Comparison with UNIX-cluster the NT-cluster is highly cost effective, widely available, and easy to implement program’s GUI. In addition, the advantages of the distributed TPNs simulation algorithm includes (1) it is suitable for the RAS hierarchical partitioned component structure; (2) it decreases the complexity of model analysis because of using the hierarchical and multi-component partition; and (3) it improves system performance.

At the moment, the size of component has strong impact onto the performance of the distributed simulation algorithm. How to get an appropriate decomposition of RAS model into a set of components, which appears to be a hard problem for general models. Otherwise, unbalanced loads of components could result in a low performance for system simulation. Optimal RAS model simulation with load balance method is an aim of our current work, which combines or divides automatically the components into a suitable size one in order to decrease the cost of synchronization and system communication operations.

Research work will be also in progress to map and port the distributed TPNs simulation algorithm to other high efficient parallel and distributed computer architectures, such as SMP, Hypercube-connection or Mesh-connection MPP machines systems.

APPENDIX: A DISTRIBUTED TPNs ALGORITHM (PSEUDO-CODE)

(1) Component Algorithm

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do in distributed manner} \\
&\text{/* Each iteration is assigned to a Java thread, then the thread} \\
&\text{grouped into a gang is allocated to a workstation with gang} \\
&\text{scheduling */} \\
&\text{Call component simulation procedures in the } i^{th} \\
&\text{component thread } P_i. \\
&\text{while (exchange tokens among components) do} \\
&\text{/* The code described below is the client procedure of} \\
&\text{component thread } P_i \text{ */}
\end{align*}
\]
if (input-input synchronization occurs in \( P_i \), \( i = 1, 2, \ldots \)) \{  
    \( P_i \) sends current clock \( T_i \) to co-ordinator.  
    \( P_i \) waits until co-ordinator’s message arrives.  
    Get current transition’s enabled clock \( T \).  
    \( P_i \) break.  
\}  
/* The code described below is the client procedures of \( P_i \) and \( P_j \) */  
if (input-out synchronization occurs in \( P_i \) and \( P_j \)) \{  
    if (in \( P_i \))  
      \( P_i \) sends request message to co-ordinator.  
    \( P_j \) waits until co-ordinator replies a message.  
    while (in \( P_i \) and returned message \( \neq \) STOP) \{  
      Call rollback recovery or roll-forward iteration in \( P_j \).  
    \}  
    \( P_i \) and \( P_j \) continue.  
\}  

(2) The Co-ordinator Algorithm  
/* The co-ordinator can be run on any workstation in NT-cluster */  
while (true) do \{  
    Sleeping until receives an asking message from component thread.  
    Get the enabled time \( T_t \) of the current transition \( t \) and it’s firing time interval [EFT, LFT].  
    switch (case) \{  
      case 1: input-out synchronization occurs in \( P_i \) and \( P_j \) \{  
        if (\( P_i \)’s Local Clock \( T_j \) doesn’t satisfy current synchronous condition with \( T_t \)) \{  
          if (\( T_j + LFT < T_t \)) inform \( P_j \) to call the rollback recovery.  
          else if (\( T_j + EFT > T_t \)) inform \( P_j \) to call the roll-forward iteration.  
        \}  
        else send STOP to \( P_j \).  
        Generate random number \( \theta \). /* EFT \leq \theta \leq LFT */  
        \( t \) is fired in \( T_t + \theta \).  
      \}  
      case 2: input-input synchronization occurs in \( P_i \), \( i = 1, 2, \ldots \) \{  
        Waiting at a barrier until all the components’ local clock \( T_1, T_2, \ldots T_m \) arrives.  
        Computing current transition’s enabled clock \( S = Max( T_1, T_2, \ldots T_m ) \).  
      \}  
    \}  
  
REFERENCES


