A Provably Correct, Non-Deadlocking Parallel Event Simulation Algorithm

Meng-Lin Yu
AT&T Bell Laboratories
Holmdel, NJ 07733

Sumit Ghosh
Brown University
Providence, RI 02912

Erik DeBenedictis
nCUBE Corporation
Beaverton, OR 97006

Abstract

This paper first summerizes and then presents a formal proof to a new conservative deadlock-free algorithm, YADDES [1], for asynchronous discrete event simulation. The proof not only makes the algorithm complete but helps us better understand the seemingly complicated algorithm. YADDES constructs a special acyclic data-flow network from the network of simulation models to keep track of the run time data dependencies, which permits a model to be correctly executed as far ahead in time as possible. The data-flow network also uses the asynchronous parallel discrete event driven technique and runs concurrently with the network of simulation models. This paper also reports a preliminary implementation of the algorithm and discusses the algorithm's limitations.

1. Introduction

Discrete event simulation has been widely accepted as a more efficient simulation method than time-based simulation for simulating many physical systems with sparse activities such as digital circuit, queueing networks, telephone networks, and simulated warfare. In this paper, we will concentrate on discussions and examples of digital circuits as our physical processes. In discrete event simulation, a simulation model representing an entity of the circuit remains idle except when excited by an external stimulus. In addition, only changes in a model's response are propagated to other models that are connected to its output. The changes are usually stored in an event queue according to their assertion times. However, this global data structure, event queue, as well as the concept of the monotonically increasing global time make it difficult to run discrete event simulation on a multiprocessor, where processors have limited resources and mechanisms to communicate with each other. Since it is generally accepted that a straightforward synchronous approach, which requires centralized control and resynchronization of all processors at the end of current activities to make all processors agree to the global time, does not scale well with large number of processors due to excessive overhead, most research efforts were made in the area of asynchronous distributed discrete event simulation.

Asynchronous distributed discrete event simulation studies usually start with a time-order preservation assumption. The assumption says that two events A and B, with B happens later than A, coming out of the same communication channel have to preserve their timing order, i.e., event B can only be received after event A. Most multiprocessor architectures support this assumption. Based on this assumption, in order to eliminate the need of a central event queue, a basic asynchronous discrete event simulation algorithm [2] requires that each model has its own queue(s). Whenever there is a change of model X's output due to model execution, an event is sent to input port queues of all models connected to the model X's output. To eliminate the need of global time, a basic asynchronous discrete event simulation algorithm requires that the assertion time is sent along with an event. In this manner, a component (simulation model) can be scheduled for simulation up to time T if each of its input ports has at least an event with time greater than or equal to T. However, the basic asynchronous discrete event algorithm is sometimes too conservative. For example, a particular input to a component may not have incoming events for an extended period of time, yet the component cannot simulate regardless how many events are queued at the other input ports of the component. This leads to the starvation problem which may cause performance degradation. For cyclic circuits, a more serious deadlock occurs when two or more

0272-4715/91/0000/0100$01.00 © 1991 IEEE

components await events from each other in order to continue the simulation. The conservative nature and the inability to determine whether the absence of information at an input port of a model means unchanged signal values or the incoming events yet to arrive cause the deadlock.

Various approaches proposed to resolve the deadlock problem fall into two major categories depending on how the absence of information at an input port is interpreted. In the deadlock-free optimistic rollback approach [3][11], the absence of information is interpreted as an unchanged signal value. If a subsequent message contradicts this interpretation, the simulation system restores (rolls back) to a previous state by sending anti-messages. In the conservative Chandy-Misra approach [2][8][12], the absence of information is interpreted as incoming events to arrive in the future. Techniques were designed to avoid or handle deadlocks associated with this interpretation. Since YADDES is a conservative approach, we will focus on the discussion of the conservative approach variations. Readers interested in the optimistic approach can read [4].

Two schools of thought are used to solve the deadlock problem. In the deadlock detection and recovery approach [2] [5] [6], the entire simulation system is permitted to execute until it results in a deadlock, i.e., when the entire simulation prematurely halts. The deadlock state can be detected by a distributed circular marker algorithm or by a time out mechanism [2] [6]. The recovery can then be done by distributed or centralized time information collection and restart control. However, the deadlock detection scheme fails for certain systems. For example, a system consisting of oscillatory components will never go into a complete deadlock state yet the non-oscillatory components of the system cannot execute due to local deadlocks. Besides, the potentially very high overhead for deadlock detection and recovery limits its applications.

In the conservative deadlock-free null message method [2] [7], null messages are used to carry time information to avoid deadlock when a model simulation results in no output changes. Null messages are otherwise treated as normal events. Its principal limitation is its inefficiency in situations where an external signal to a cyclic circuit remains unchanged for extended time. For example, if the external inputs to an idle cyclic circuit do not change, say, for a million units of the total loop delay, a million useless null messages are generated and transmitted with this method.

The above mentioned problems lead to the design of a new approach, YADDES[1], to avoid deadlocks. The next section summerizes the YADDES algorithm. Detailed examples and descriptions of the algorithm can be found in [1].

2. The YADDES Approach

In this section, a new conservative approach YADDES is summerized. Instead of blindly generating null messages at a fixed interval, YADDES constructs an acyclic data-flow network from the original circuit for computing incremental changes of lookahead or "defined-up-to" information. Both the data-flow network computation and the model simulation employ the event driven technique and run concurrently. The "defined-up-to" information can be considered as optimal lookahead information, dynamic data dependency information, or intelligent null messages. This information guarantees that the simulation will never deadlock and each model can simulate as far ahead in time as permitted by the true data dependency at run time. It also says that any execution further in time may generate incorrect simulation results. The way to construct a data-flow network is described below.

2.1 Data Network Construction - Preprocessing

A circuit to be simulated is first preprocessed for constructing an acyclic data-flow network. The preprocessing consists of the following steps.

Feedback Loops Removal
All subcircuits that constitute cyclic directed graphs are first identified. Other entities of the system that constitute acyclic graphs are simulated as the basic asynchronous discrete event simulation. A feedback arc set [9] S = $\{E_1, E_2, ..., E_n\}$ of a directed graph is then determined such that the graph becomes acyclic following the removal of all of the edges E_1 through E_n. While the size of this cut set affects the overall data-flow network complexity, this approach does not need to find a minimal feedback arc set which is an NP-complete problem. For each E_i for all $i \in \{1,2,...,n\}$ in the original directed graph, a new

acyclic directed graph (reduced acyclic graph) is reconstructed by replacing E_i with two edges E_i^{in} and E_i^{out} as illustrated in Figure 1.

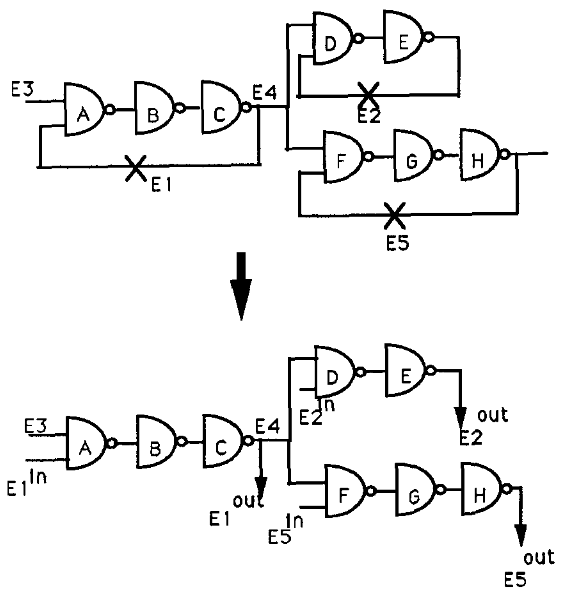

Figure 1: Reducing a Cyclic Directed Graph to an Acyclic Directed Graph.

Primed and Unprimed Reduced Acyclic Graphs

A data-flow network is constructed by interconnecting two identical copies of the reduced acyclic graph side-by-side through a crossbar switch as shown in Figure 2. A component X of the left-hand side reduced acyclic graph is named pseudo **primed** component, labeled X'. A component X of the right-hand side reduced acyclic graph is named pseudo **unprimed** component, also labeled X. The term "pseudo" is used to distinguish it from the real simulation models of the original circuit. Every input port of a pseudo primed component X' that has a label of the form E_i^{in}, where E_i was the feedback arc chosen to break the loop containing X, has a fixed value of ∞.

Crossbar Switch Construction

An output port of every component X' that has a label E_j^{out} is linked to every input port of any pseudo unprimed component Y that has a label of the form E_k^{in}, for all k. These links form the crossbar switch. Every link between E_i^{out} of a pseudo primed component X' to an input port E_j^{in} of a pseudo unprimed component Y has a weight whose value is the minimum propagation delay from components X to Y in the original cyclic graph. If the component X may not affect Y at all, the weight is infinity and the link is considered non-existent. In addition, at the right hand side of the link, all the links coming out of the crossbar switch into a particular E_j^{in} is reduced to 1 by taking the minimum of all those incoming values. For the cyclic graph in Figure 1a, the corresponding data-flow network is shown in Figure 2.

2.2 Definitions of Lookahead Variables

Definition of U: Associated with every real simulation model X is an event list that contains the yet to be consumed events. U_X is defined to the the smallest time value of the events currently in the list. If the list is empty at an instant, the value of U_X is set to be ∞. All U_X for all X are, therefore, initialized to ∞.

Definition of W: The quantity W_X is associated with the output of a pseudo unprimed component X in the data-flow network. $W_X \equiv$ minimum(U_X+d, W_1+d, W_2+d,..., W_n+d), where W_1, W_2, ..., W_n refers to the W value at the input ports 1,...,n of X and "d" refers to component X's propagation delay. The value of W_X for every X is initialized to 0 and simulation is complete when each U_X and W_X for all X in the system is at ∞.

W_X can be interpreted as the earliest possible time of the next message at the output of X. It implies that the output of the model X is defined up to t = $W_{\overline{X}}$. Following the definition, we notice that W is defined recursively and that W can also be computed using discrete event technique. Changes of W or W' are thus computed incrementally through the data-flow network and concurrently with the model simulation whenever there is a "defined-up-to" value change. Since the data-flow network is acyclic, the computation is straightforward, always goes from left to right, and is free from deadlock.

Definition of W': The quantity W'_X is associated with the output of the pseudo primed component X' in the data-flow network. $W'_X \equiv$ minimum(U_X+d, W'_1+d, W'_2+d,.., W'_n+d), where $W'_1, W'_2, ..., W'_n$ refers to the W' values at the input ports 1,...,n of X' and "d" refers to X's propagation delay. It represents an intermediate value in the computation of Ws and may be

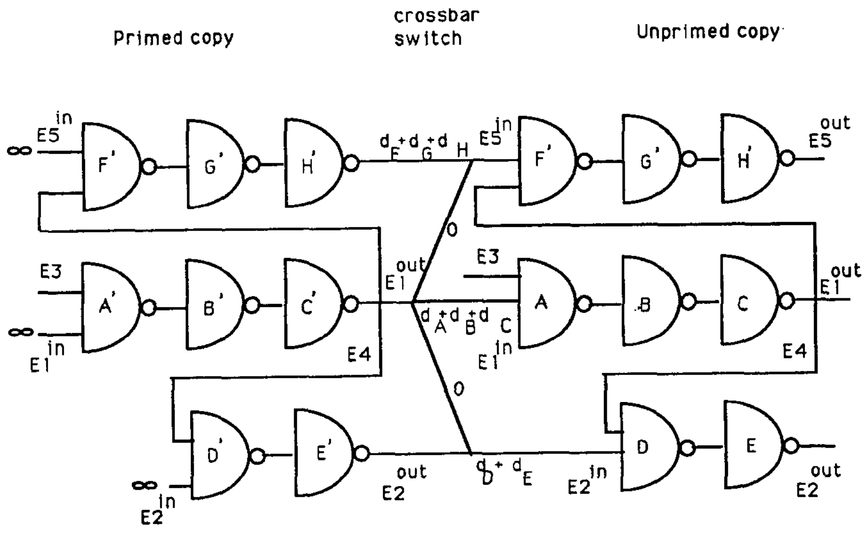

Figure 2: Data-Flow Network Constructed for Cyclic Circuit in Figure 1a.

understood as an optimistic estimate of next message time at the output of simulation model X.

Definition of K: The quantity K_X is associated with the real simulation model X. $K_X \equiv$ minimum of $\{W_i\}$, for all input port i of X.

Given the data-flow network and above definitions, the simulation system would consist of two elements, real simulation models, and the data-flow network along with its Ws and W's. The interactions of these two elements in the simulation process go as follows. A real simulation model X is excited by signal transition messages received at its input ports. It then queries the data-flow network by supplying information of the smallest time (U) in its current event queue. The data-flow network computes and replies with the information of "defined-up-to" (Ws) for each input port. An event in component X's event queue can then be executed if its event time is smaller than the smallest "defined-up-to" (K) among all the input ports, regardless whether every input port has received an event or not.

2.3 Algorithm - Simulation and Lookahead Variables Computation

The YADDES algorithm is described in Figure 3 and Figure 4. The dynamics of the simulation system can be summerized as follows. When a signal transition is asserted at an input port of real simulation model X, it may alter the U value of X or leave it unchanged. In case the U value is altered as a consequence of the incoming signal transition (which has a smaller event time), the simulation model X initiates the pseudo primed components X' of the data-flow network for W-computation and waits for acknowledgement from X'. X' computes W'_X and if its value is unchanged, an acknowledgement signal is sent back to the real simulation model X. If the new value of W'_X is different, X' initiates a chain reaction i.e., it propagates the new W'_X value to every pseudo component Y' connecting to its output port and waits for acknowledgements from each of those pseudo components. Every subsequent pseudo component, Y', then repeats this process. This chain process of computation and propagation of new W' or W continues until the driving force i.e., a change in the value of W' or W at the output of a pseudo component, dies out or when the change may not propagate any further at the rightmost primed pseudo components. Since the data-flow network is acyclic and bounded in size, the event driven W-computation is guaranteed to terminate in finite bounded time. Upon receiving all acknowledgements later on, the pseudo component Y' then sends an acknowledgement back to X' and X' will send an acknowledgement

```
Real simulation model X:
        loop forever {
          if (K > U) { /* model execute */
            execute simulation model and generate output signal;
            if (output signal changed) {
              send output event to all models in the fanout;
              wait till acknowledgement signals received from event receivers;
            }
            update event queue and consequently new U value;
            initiate pseudo component X' to update Ws, $W'$s;
            /* No acknowledgements are needed but acknowledgements */
            /* may still be used for ease of implementation.      */
          }
          read in events at input ports  /* input events */
          if (there is a new event) {
            update event queue and order events according to time;
            if (new event alters U value) {
              initiate pseudo components X' to update Ws, $W'$s;
              wait till an acknowledgement received from X';
            }
            send an acknowledgements back to the event sender;
            compute K = minimum of all W values at input ports of model X;
          }
        }
```

Figure 3: Operations of a Simulation Model X.

back to the real simulation model X when all the acknowledgements are received to indicate the termination of W-computation. The real simulation model X can now acknowledge the event originator, which can now and only now change its U value.

The real simulation model X accesses the W values of each of its input ports and computes the minimum value K_X. If K_X is larger than U_X, the model X may be executed for the event corresponding to the U value. In case no new signals are generated at the output port of X, the event corresponding to the current U is removed from the event queue and the new value of U reflects the currently known next event time in the queue. If the queue is empty, U is set to infinity. If a signal transition is generated at an output port as a consequence of execution of the simulation model X, an event is sent to every simulation model connected to the output of X. The real simulation model X waits for acknowledgements coming back and then the event corresponding to the current U value is removed from the event queue and the value of U is updated.

3. Proof of Correctness of the Algorithm

The proof of correctness of YADDES algorithm requires proofs of the execution of events in the correct order, the absence of deadlock, and the termination of simulation. Since the basic asynchronous discrete event simulation, whose correctness was proved in [2], is used as the backbone of YADDES, we will only show the proof of critical parts unique to the YADDES algorithm.

3.1 Execution of Events in the Correct Order

Consider that two events E_1 and E_2 with assertion times t $= t_1$ and t $= t_2 (t_2 > t_1)$ respectively are sent by a simulation model X. The time-order preservation assumption says that where X sends E_1 and E_2 at real time t $= t_3$ and t $= t_4$ $(t_4 > t_3)$ respectively to the same receiver, the receiver is guaranteed to receive E_1 prior to E_2. Based on this assumption and definitions of Us and Ks, to prove the event executions in correct order, we only need to show that all Ks increase monotonically such that no two events can be executed in reverse order.

```
Pseudo Component X' (or X):
/* An implementation uses acknowledgements for all W-computation */
          loop forever {
            read in command from real simulation model X (for $X'$ only),
              new W' (or W) value from left, and acknowledgements from right;

            if (command from model X is update_W) {
              compute W' ;
              if (W' value remains unchanged) {
                send an acknowledgement signal back to model X;
              }
              if (W' computes to a new value) {
                propagate new W'(W) value and wait for acknowledgements from the receivers;
              }
            }
            if (new W' (W) value at input is read) {
              compute the output W' (W) value;
              if (output W' (W) value is unchanged) then
                send an acknowledgement back to sender;
              else if (output W' (W) value is changed) {
                propagate new W'(W) value and wait for acknowledgements from the receivers;
              }
            }
            if (all acknowledgements are back from each receiver) {
              if (acknowledgement is for itself) send done signal to simulation model X;
              if (acknowledgement is not for itself) relay it towards the original requester;
            }
          }
```

Figure 4: Operations of a Pseudo Component X' (Primed) or X (Unprimed).

We will first prove that W_X indeed represents the define-up-to information, that is, no events can come out of X with assertion time smaller than W_X. Let's consider a general circuit with feedback whose data-flow network is shown in Figure 5. For simplicity, assume all external input events have already been inserted in the proper event queues and all U values have already accounted for them.

For any pseudo primed component Y', by definition,
$W'_Y = \min\ (U_Y + d_Y,\ W'_{A_1} + d_Y,\ W'_{A_2} + d_Y,\ W'_{A_3} + d_Y, \ldots)$, where each A_i', for all i, is a fanin component of Y'.

Also by definition, $W'_{A_i} = \min\ (U_{A_i} + d_{A_i},\ W'_{B_1} + d_{A_i},\ W'_{B_2} + d_{A_i},\ W'_{B_3} + d_{A_i}, \ldots)$, where each B_j, for all j, is a fanin component of A_i'.

By substitution, we have

$$\begin{aligned} W'_Y = \min\ (&U_Y + d_Y,\\ &U_{A_1} + d_{A_1Y},\ U_{A_2} + d_{A_2Y},\ U_{A_3} + d_{A_3Y}, \ldots,\\ &W'_{B_1} + d_{A_1Y},\ W'_{B_2} + d_{A_1Y},\ W'_{B_3} + d_{A_1Y}, \ldots,\\ &W'_{B_1} + d_{A_2Y},\ W'_{B_2} + d_{A_2Y},\ W'_{B_3} + d_{A_2Y}, \ldots,\\ &W'_{B_1} + d_{A_3Y},\ W'_{B_2} + d_{A_3Y},\ W'_{B_3} + d_{A_3Y}, \ldots,\\ = \min\ (&U_Y + d_Y,\\ &U_{A_1} + d_{A_1Y},\ U_{A_2} + d_{A_2Y},\ U_{A_3} + d_{A_3Y}, \ldots,\\ &W'_{B_1} + D_{B_1Y},\ W'_{B_2} + D_{B_2Y},\ W'_{B_3} + D_{B_3Y}, \ldots), \end{aligned}$$

where d_{AY} means the minimum delay from input of A' to output of Y' or $d_Y + d_A$ and D_{BY} means the minimum delay from B' output to Y' output or $D_{BY} = \min\ (d_{iY})$ for all i connected to output of B. Note that $d_{BY} \equiv D_{BY} + d_B$.

By repeated substitution until we reach the leftmost pseudo components $\{F_i'\}$, we have,

$$\begin{aligned} W'_Y = \min\ (&U_Y + d_Y,\\ &U_{A_1} + d_{A_1Y},\ U_{A_2} + d_{A_2Y},\ U_{A_3} + d_{A_3Y}, \ldots, \end{aligned}$$

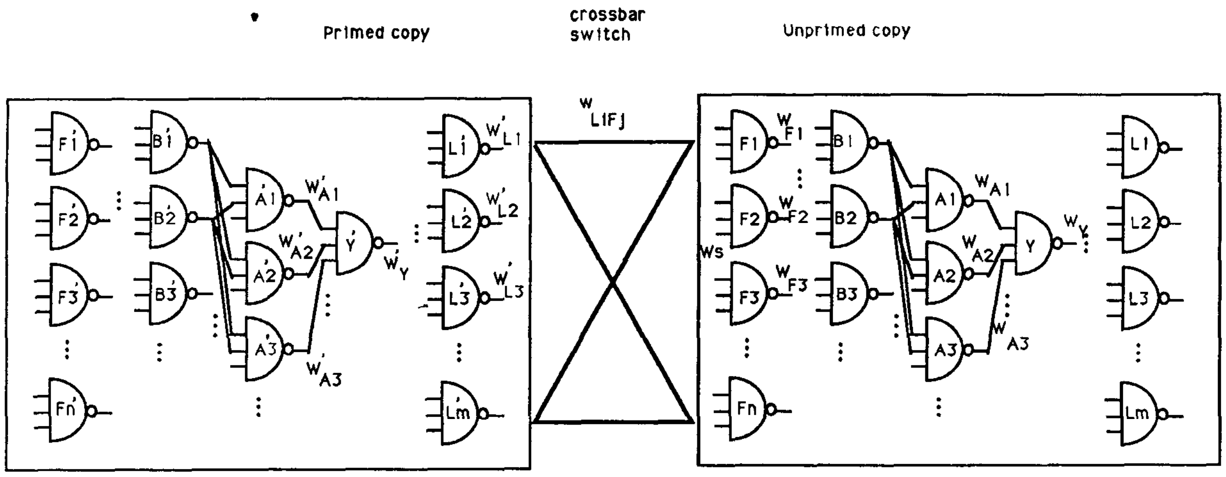

Figure 5: A General Circuit to show that W value increases monotonically.

$U_{B_1} + d_{B_1Y}, U_{B_2} + d_{B_2Y}, U_{B_3} + d_{B_3Y}, \ldots,$
$U_{C_1} + d_{C_1Y}, U_{C_2} + d_{C_2Y}, U_{C_3} + d_{C_3Y}, \ldots,$
$\ldots\ldots,$
$W'_{F_1} + D_{F_1Y}, W'_{F_2} + D_{F_2Y}, W'_{F_3} + D_{F_3Y}, \ldots),$
------- (0)

where $\{A_i'\}$, $\{B_i'\}$, $\{C_i'\}$ ... $\{F_i'\}$ are the components reachable from Y' to the left and are usually referred to as components in Y's input cone.

Since all inputs of the leftmost pseudo primed components have the value of ∞, $W_{F_i} = \min(\infty + d_{F_i}, U_{F_i} + d_{F_i}) = U_{F_i} + d_{F_i}$. We then have

$W'_Y = \min(U_Y + d_Y,$
$U_{A_1} + d_{A_1Y}, U_{A_2} + d_{A_2Y}, U_{A_3} + d_{A_3Y}, \ldots,$
$U_{B_1} + d_{B_1Y}, U_{B_2} + d_{B_2Y}, U_{B_3} + d_{B_3Y}, \ldots,$
$U_{C_1} + d_{C_1Y}, U_{C_2} + d_{C_2Y}, U_{C_3} + d_{C_3Y}, \ldots,$
$\ldots\ldots,$
$U_{F_1} + d_{F_1Y}, U_{F_2} + d_{F_2Y}, U_{F_3} + d_{F_3Y}, \ldots).$

Now let Y be L_i, a rightmost pseudo prime component before crossbar switch, we have

$W'_{L_i} = \min(U_{L_i} + d_{L_i},$
$U_{A_1} + d_{A_1L_i}, U_{A_2} + d_{A_2L_i}, U_{A_3} + d_{A_3L_i}, \ldots,$
$U_{B_1} + d_{B_1L_i}, U_{B_2} + d_{B_2L_i}, U_{B_3} + d_{B_3L_i}, \ldots,$
$U_{C_1} + d_{C_1L_i}, U_{C_2} + d_{C_2L_i}, U_{C_3} + d_{C_3L_i}, \ldots,$
$\ldots\ldots,$
$U_{F_1} + d_{F_1L_i}, U_{F_2} + d_{F_2L_i}, U_{F_3} + d_{F_3L_i}, \ldots).$
------- (1)

Since the circuit is a connected graph, any pseudo component X' at least belongs to the input cone of some L_i', whose W'_{L_i} will have a term $U_X + d_{XL_i}$ in the minimum expression (1) or itself is an L_i', whose W'_{L_i} will have a term $U_X + d_X$ in the minimum expression (1).

Now since between all primed L's and unprimed Fs there is a fully connected cross-bar switch with minimum operators at the F side, the W_{S_k} immediately after the switch for input k of F_i is
$W_{S_k} = \min(W'_{L_1} + w_{L_1F_i}, W'_{L_2} + w_{L_2F_i}, W'_{L_3} + w_{L_3F_i}, \ldots, W'_{L_m} + w_{L_mF_i}),$ -------- (2)
where $w_{L_jF_i}$, is the weight of the crossbar link or precomputed minimum delay between real simulation models L_j and F_i. Note that in practice, many terms will be ∞ and disappear in the minimum expression (2) due to the lack of static data dependency. Their corresponding links also disappear in the crossbar switch.

Now for a pseudo unprimed component X, we have an expression for X similar to expression (0).

$W_X = \min(U_X + d_X,$
$U_{A_1} + d_{A_1X}, U_{A_2} + d_{A_2X}, U_{A_3} + d_{A_3X}, \ldots,$
$U_{B_1} + d_{B_1X}, U_{B_2} + d_{B_2X}, U_{B_3} + d_{B_3X}, \ldots,$
$U_{C_1} + d_{C_1X}, U_{C_2} + d_{C_2X}, U_{C_3} + d_{C_3X}, \ldots,$
$\ldots\ldots,$
$W_{F_1} + D_{F_1X}, W_{F_2} + D_{F_2X}, W_{F_3} + D_{F_3X}, \ldots),$
-------- (3)

where $\{A_i\}$, $\{B_i\}$, $\{C_i\}$ ... $\{F_i\}$ are the components in X's input cone, and their events will affect X without having to go through the feedback arc set or without having to go through the crossbar switch. For the components indirectly affecting X through the crossbar switch, they can be accounted for by substituting (1) (2) and $W_{F_i} \equiv \min(U_{F_i} + d_{F_i}, W_{S_1} + d_{F_i}, W_{S_2} + d_{F_i}, ...,)$ into (3). We then have

$$\begin{aligned} W_X = \min\,(& U_X + d_X, \\ & U_{A_1} + d_{A_1X}, U_{A_2} + d_{A_2X}, U_{A_3} + d_{A_3X}, ..., \\ & U_{B_1} + d_{B_1X}, U_{B_2} + d_{B_2X}, U_{B_3} + d_{B_3X}, ..., \\ & U_{C_1} + d_{C_1X}, U_{C_2} + d_{C_2X}, U_{C_3} + d_{C_3X}, ..., \\ & ......, \\ & U_{F_1} + d_{F_1X}, U_{F_2} + d_{F_2X}, U_{F_3} + d_{F_3X}, ..., \\ & U_i + d^{switch}{}_{iX}, \text{ for all components } i \text{ not in } \\ & \{A_k\} \cup \{B_k\} \cup \{C_k\} \cup ... \cup \{F_k\}\ ), \end{aligned}$$

where $d^{switch}{}_{iX} = \min(d_{iL_1} + w_{L_1F_j} + d_{F_jX}, d_{iL_2} + w_{L_2F_j} + d_{F_jX}, d_{iL_3} + w_{L_3F_j} + d_{F_jX}, ..., d_{iL_m} + w_{L_mF_j} + d_{F_jX})$, for all j.

Note that all the terms in W'_Z for component Z' in the input cone of X' do not appear in the above expression (eliminated by the minimum operator), since they are greater than the corresponding terms in X's input cone by the value of $\min(\{ d_{XL_i} + w_{L_iF_j} + d_{F_jZ}\})$, for all i and all j.

The above can be rewritten as
$W_X = \min\ (U_X + d_X,\ U_j + d_{jX}$, for every component j in X's input cone, $U_i + d^{switch}{}_{iX}$, for each i not in X's input cone.) -------- (4)

Expression (4) is a correct description of data dependency of component X. In essence, it says that model X's output could have an event (value change) at time no smaller than (a) the current smallest unconsumed event time in its queue plus delay of X, (b) the smallest of all of the current unconsumed event time in queue Y of X's input cone component Y plus the delay from Y input to X output, or (c) the smallest of all of the current unconsumed event time in any other component Z plus the delay from Z input to X output through the feedback arc of the original circuit. That is, any event in the event queue of any component cannot reach (affect) component X with an assertion time smaller than W_X. Therefore the it is correct in stating that the output of X has been defined up to W_X time.

Lemma 1: W_X is the earliest time that X output can change value.

We now show that the all Ws (not W's) increase monotonically. First we examine expression (4) statically. Note that some U_i may decrease due to receiving a new event. Let's examine only those cases since U_i going up will not cause W value to decrease in expression (4). U_i can decrease only because of another event firing with a smaller assertion time, say U_j. Let α_{ij} denotes the minimum delay between i and j, that is, either d_{ij} or $d^{switch}{}_{ij}$. Consider the case $U_j^{old} + \alpha_{jX} \leq U_i^{old} + \alpha_{iX}$. Since $U_i^{new} + \alpha_{iX} = U_j^{old} + \alpha_{ji} + \alpha_{iX} \geq U_j^{old} + \alpha_{jX}$ (Inequality occurs when J is in the input cones of both I and X but X is not in I's input cone,) expression (4) will not decrease in value. Now considering the case $U_j^{old} + \alpha_{jX} > U_i^{old} + \alpha_{iX}$. Since U_i decreases in value, $U_i^{old} > U_i^{new} = U_j^{old} + \alpha_{ji}$. By adding α_{iX} at both sides, we get $U_i^{old} + \alpha_{iX} > U_j^{old} + \alpha_{ji} + \alpha_{iX} > U_j^{old} + \alpha_{jX}$, which contradicts to our assumption for this case (impossible case).

However, dynamically, if U_j is allowed to change to a higher value right after finishing simulating an event, we may have the traditional "race" problem. A race problem can be explained by the following case. Assuming $U_j^{old} + \alpha_{jX}$ and $U_i^{new} + \alpha_{iX}$ are the minimum terms of some W_X expression before and after the event causing U_j and U_i to change, and also assume $U_j^{new} + \alpha_{jX} > U_i^{new} + \alpha_{iX}$ but $U_j^{new} + \alpha_{jX} < U_i^{old} + \alpha_{iX}$, which implies $U_j^{old} + \alpha_{jX} < U_i^{old} + \alpha_{iX}$. Since the information of Us are carried by Ws through W-computation and W-propagation and since Ws are computed concurrently in a distributed environment, it is possible that component X receives the effect of U_j^{new} change before that of U_i^{new} if unrestricted W-propagation is allowed. For example, if component i is in component j's input cone, W'_j computation may not use the correct new U_i value if U_j is allowed to change right away. This implies that W_X value will go through a sequence of changes of $U_j^{old} + \alpha_{jX}$, $U_j^{new} + \alpha_{jX}$, and $U_i^{new} + \alpha_{iX}$. Since $U_j^{new} + \alpha_{jX} > U_i^{new} + \alpha_{iX}$, this represents a danger of decreasing W value. However, because we restrict U update in such a way that any changes of U value, after a component finishes simulating an input event and sending out possible output event, can only occur after the corresponding W chain reaction dies down by acknowledgement mechanism, the race problem cannot happen. This insures that any U_j

in any W expression will be the old value until U_i^{new} is updated and propagated to all components needing it for correct W (W') computation. Therefore we conclude that W values will not decrease even though U values may decrease.

Lemma 2: All W values increase monotonically.

Since Ks are defined as the minimum of some Ws, K will increase monotonically if all Ws increase monotonically.

Lemma 3: All K values increase monotonically.

This completes our proof of the following theorem.

Theorem 1: In YADDES, events are executed in correct order.

3.2 Freedom from Deadlock

To prove YADDES is deadlock-free, we first restate that W-computation in the data-flow network can never deadlock as the network is acyclic. We will now show that no deadlock can happen in YADDES algorithm for circuits without zero-delay cycles (loops). Loops with zero-delays are uninteresting and unrealistic as any event may traverse such a loop infinite times with no time advances.

Lemma 4: For circuits without zero delay loops, at any instant, at least one event with the smallest assertion time in the entire simulation system can be scheduled for simulation.

We prove this lemma by contractions. Assume that deadlock occurs (no event can be scheduled) during simulation and there are events E_1, E_2, E_3 ..., E_n sitting in the queues of models X_1, X_2, X_3, ..., X_n yet to be executed. Without loss of generality, assume E_i has the smallest assertion time, which implies U_{X_i} is smaller than U_{X_j} for all $j \neq i$. By the definition of U and K and model operations in Figure 3, if E_i cannot be executed, the K value of the model X_i must be smaller than or equal to U_{X_i} i.e.,

$K_{X_i} \leq U_{X_i}$ -------- (5).

By definition, K_{X_i} value is the minimum of all W values at the inputs of X_i, and, consequently, the W values at the outputs of the immediate fanin pseudo components $\{Y_j\}$ are less than U_{X_i}. However, from expression (4), we know that each W_{Y_j} has a form of of the following,
$W_{Y_j} = \min(U_{X_1}+\alpha_{X_1,Y_j}, U_{X_2}+\alpha_{X_2,Y_j}, U_{X_3}+\alpha_{X_3,Y_j}, \ldots, U_{X_i}+\alpha_{X_i,Y_j}, \ldots U_{X_n}+\alpha_{X_n,Y_j})$, where α_{X_k,Y_j} is either d_{X_k,Y_j} or $d^{switch}{}_{X_k,Y_j}$ and is a finite positive number.
Since each U_{X_k} for all $k \neq i > U_{X_i}$, and $\alpha_{X_i,Y_j} > 0$ (non-zero loop delay), we conclude that each $W_{Y_j} > U_{X_i}$ and therefore $K_{X_i} > U_{X_i}$, which contradicts inequality (5).

For multiple events having the same smallest assertion time, the above still hold true since at least one W_{Y_k} will have a term $U_{X_{min\_time}} + \alpha_{X_{min\_time},Y_k}$, with the α term > 0 (otherwise we have zero-delay loops.) Consequently, we have shown that at any instant, at least one event with the smallest assertion time in the entire system will always be able to fire. This can be restated in the following theorem.

Theorem 2: The YADDES algorithm is deadlock-free.

3.3 Termination of Simulation

A simulation terminates for model X when W_X has a value of ∞, because $W_X = \infty$ means no more events can come out of output of X. The model X can then report end_of_simulation. When all Ws are equal to ∞, the entire simulation process terminates. This condition is good for non-oscillatory circuits, whose total number of events is finite, as all Ws equal to ∞ implies all Us are equal to ∞, which in term implies no events. For oscillatory circuits, we modify the definition and say that a simulation stops when all Ws are greater than a pre-determined time - an observation period that a user is interested in, as the circuit oscillates forever. In either case, since Ws and Us increase monotonically,. since a model simulation, a uni-directional W-computation, and a message transmission all take finite time, and since there are only a finite number of models and a finite number of events, the simulation will terminate in finite time.

Theorem 3: Simulation using YADDES algorithm terminates in finite time.

4. A Preliminary Implementation and Discussions

4.1 An Implementation

The YADDES algorithm was implemented on a message passing oriented hypercluster parallel processor. The principal purpose of the implementation was to verify the correctness of the algorithm. In an experiment, a latch constructed from two cross coupled NAND gates was represented and simulated in the implementation. The propagation delay for each of the gates was chosen to be 500ns (nano-second) with the consequence that the cumulative propagation delay through the feedback loop was 1000ns. External transitions to be asserted at the primary inputs of the latch were chosen such that the latch would experience both stability and oscillation. For the experiment, the number of external transitions at both of the primary inputs were varied from 1000 to 10000 and the CPU time was measured for each case. In addition, three sets of average time periods of the external signals were selected -- 1000, 10000, and 100000, for the experiment. The results of simulations are expressed through the graphs in Figure 6a and Figure 6b.

Figure 6a is a log-log graph where the Y-axis represents the CPU time in seconds and the X-axis the number of external transitions asserted at the primary input ports. The three overlapping graphs I, II, and III refer to the three scenarios corresponding to the values 1, 10, and 100 of the ratio -- average time period of external transitions (T) to the cumulative propagation delay around the loop ($\sum_i d_i$). Figure 6b expresses the same results as in Figure 6a except that the X-axis represents different values of the ratio $T/\sum_i d_i$. Figure 6c expresses the results of simulation of the same circuit in the algorithm proposed in [7]. It may be concluded from the graphs in Figures 6b and 6c that the performance of the YADDES algorithm is independent of the ratio $T/\sum_i d_i$ and is consequently free from the limitations of the algorithm reported in [7].

4.2 Limitations of the Algorithm

We have shown that YADDES is a provably correct, non-deadlocking conservative discrete event algorithm and does not have the drawbacks

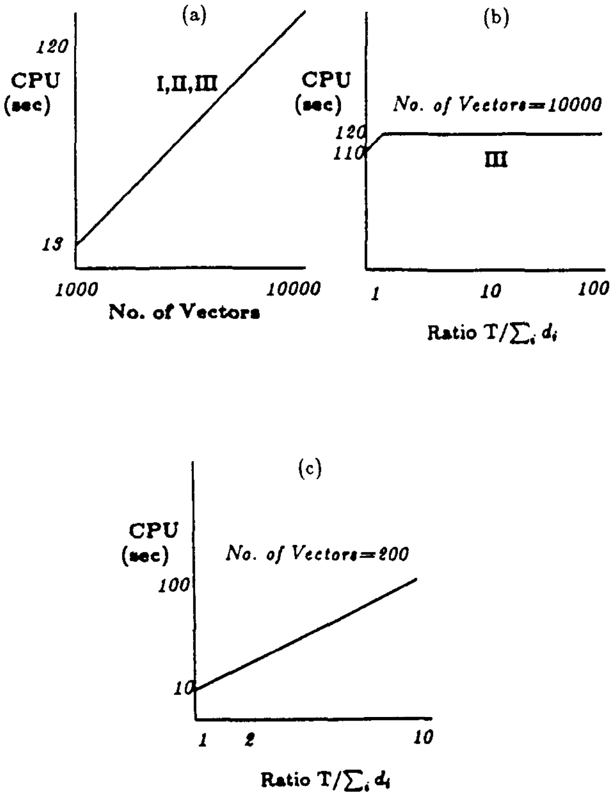

Figure 6: Performance Measurement of the Algorithm for a Cross Coupled Nand Latch.

of null message methods or deadlock recovery methods. A natural question to ask now is, "Is this algorithm a solution for all asynchronous discrete event simulation problems?". Let's first examine the complexity involved with this algorithm. The pseudo primed and unprimed components consist of nothing more than a variable W' or W along with the W-computation for each component. The data-flow network structure, other than the crossbar switch, can in practice share the network of the real simulation models. Computation overhead involved in W-computation is difficult to analyze without a good implementation and thorough experiments. However, we note that the W-computation can overlap with model executions. If a model execution is a non-trivial task, the simple W-computation overhead is comparatively quite small.

The major overhead is apparently the crossbar switch. If a circuit has few loops compared with the number of components, the switch usually degenerates to a few interconnections and is quite simple and efficient. We expect low overhead and a good speedup for this kind of applications. On the other hand, for circuits with many more loops than components, the switch approaches a full-fledged n^2 crossbar switch. In the extreme case, the data-flow network becomes a fully connected crossbar switch connecting two columns of all n components. The algorithm is then equivalent to broadcasting any event changes to all other components for computing the dynamic data dependencies. This is clearly very inefficient as a very large number of messages have to be transmitted and processed. However, in these cases, the problems themselves usually have very complicated behavior and may not lend themselves to parallel discrete event simulation. In those cases, the above scenario may indeed be the only way to obtain any lookahead information for conservative simulation approaches. The answer, therefore, really depends on the nature of the problem domain. The amount of parallelism available, the average percentage of components changing values after model execution, and the average degree of dependency are among the factors that we have to consider before choosing an appropriate asynchronous parallel algorithm. Interested readers can refer to [10] for more information. However, we do believe that the YADDES algorithm is an excellent choice for problems suitable for conservative asynchronous discrete event simulation.

5. Conclusion

This paper has presented an analytical proof of correctness of YADDES algorithm. The proof supports the claim that the algorithm is deadlock-free. It also proves that the scheduling is optimal in the sense that any model execution further in time violates the data dependency. A preliminary implementation of the algorithm shows that the algorithm does not suffer the same problem that a null message method has for cyclic circuits with slowly changing external inputs. Future researches include finding a class of applications most suitable for the algorithm, the average distance of W-computation and propagation, and a hardware implementation of W-computation network.

References

[1] M.-L. Yu, S. Ghosh, and E. DeBenedictis, "A Non-Deadlocking Conservative Asynchronous Distributed Discrete Event Simulation Algorithm," to appear in Proceedings of the 1991 Western Simulation Multiconference, Jan 23-25, 1991, Anaheim, CA.

[2] J. Misra, "Distributed Discrete-Event Simulation," Computing Surveys, Vol 18, No 1, March 1986, pp.39-65.

[3] D. Jefferson, "Virtual Time," ACM Transactions on Programming Languages, VVol 7, No 3, July 1985, pp. 404-425.

[4] Y.-B. Lin, and E. D. Lazowska, "A Study of Time Warp Rollback Mechanisms," 1990 Distributed Simulation Conference.

[5] K.M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications of the ACM, Vol 24, No 4, April 1981, pp. 198-206.

[6] K.M. Chandy, L.M. Haas, and J. Misra, "Distributed Deadlock Detection," ACM Transactions on Computer Systems, Vol 1, No. 2, May 1983, pp. 144-156.

[7] S. Ghosh and M.-L. Yu, "An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors," Proceedings of the 1988 International Conference on Parallel Processing, August 15-19, 1988, St. Charles, Illinois.

[8] D. A. Reed and A. Malony, "Parallel discrete event simulation: The Chandy-Misra Approach", Proceedings of the SCS Multiconference on Distributed Simulation, 3-5 February 1988, San Diego, California, pp.8-13.

[9] N. Deo, "Graph Theory with Applications to Engineering and Computer Science," Prentice Hall Inc. 1974.

[10] Y.-B. Lin, and E. D. Lazowska, "Optimality Considerations of "Time Warp" Parallel Simulation," 1990 Distributed Simulation

Conference.

[11] P. L. Reiher, and D. Jefferson, "Limitation of Optimism in the Time Warp Operating System," 1989 Winter Simulation Conference, pp. 765-770.

[12] L. Soule, and A. Gupta, "Parallel Distributed-Time Logic Simulation," IEEE Design & Test of Computers, December 1989, pp. 32-48.