A Novel Algorithm for Discrete-Event Simulation

Asynchronous Distributed Discrete-Event Simulation Algorithm for Cyclic Circuits Using A Dataflow Network

Erik DeBenedictis, Ncube Corporation
Sumit Ghosh, Brown University
Meng-Lin Yu, AT&T Bell Laboratories

Simulation mimics a physical process to verify correctness, identify errors, and generate performance estimates before designers fabricate a prototype. Digital hardware designs, industrial control circuits, and aircraft are usually simulated extensively. Time-based simulation techniques are efficient for processes whose activities are concentrated at regular time intervals that can be determined a priori.

Discrete-event simulation techniques apply where the activities are distributed irregularly in time, such as in digital hardware, queuing networks, and banking transactions. In discrete-event simulation, a simulation model representing an entity of the physical process remains idle except when excited by a stimulus external to it. In addition, only changes in a model's response are propagated to other models connected to its output.

Figure 1 shows an example digital hardware design. Each block A through G represents a design component and constitutes an entity of the physical process. The propagation delays of A, B, C, D, and F are 5, 2, 4, 5, and 3 nanoseconds, respectively. Assuming an external stimulus at the inputs of A and B at $t = 0$ ns, the responses from A and B may be asserted at $\{C\}$ and $\{D,F\}$ at $t = 5$ ns and $t = 2$ ns, respectively. A response from each of C, D, and F may in turn be asserted at the inputs of E and G at $t = 9$ ns, $t = 7$ ns, and $t = 5$ ns, respectively. The activities of the physical process are distributed irregularly in time, so we formulate and efficiently simulate them through discrete-event simulation.

A synthesized dataflow network and a computed quantity "time of next event" guarantee correctness and freedom from deadlock.

A traditional algorithm to perform discrete-event simulation of digital hardware on a uniprocessor proceeds as follows: An event queue stores the events in increasing order of their assertion times, where the head of the queue refers to the event with the smallest assertion time. Initially the event queue is empty, and the external stimuli are asserted at the inputs of components A and B. The stimuli generate activity, namely the execution of A and B at $t = 0$ ns, shown in the event queue in Table 1. At this stage, the algorithm examines the event queue and selects for execution events with the smallest time value, namely A and B at $t = 0$ ns. Assume that the execution of models A and B at $t = 0$ ns generates output transitions at $t = 5$ ns and $t = 2$ ns, respectively, that are asserted at the inputs of models C, D, and F, as shown in Figure 1. The new activities are expressed through events C at $t = 5$ ns, D at $t = 2$ ns, and F at $t = 2$ ns in the event queue in Table 1. Then the algorithm examines the event queue again, and selects the events with the smallest time — namely D at $t = 2$ ns and F

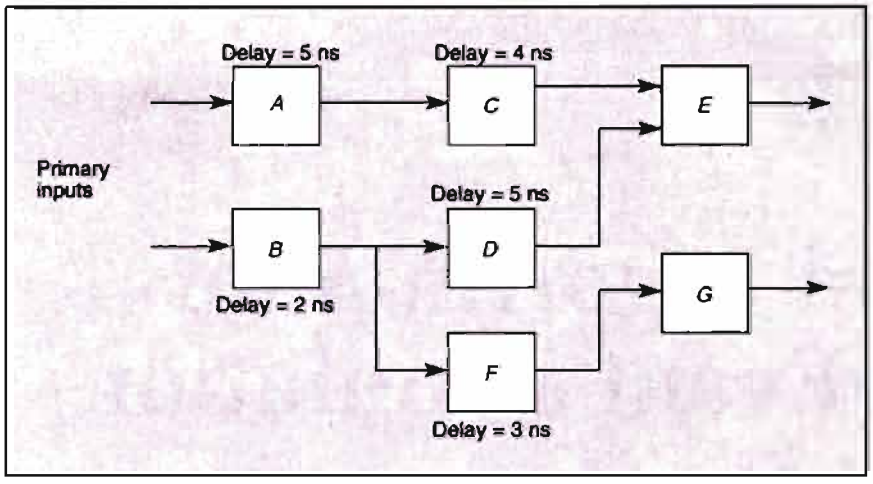


Figure 1. Discrete-event simulation of an example digital hardware design.

Table 1. Event queue for the simulation in Figure 1.

Time	Components Activated
$t = 0$	A, B
$t = 2$	D, F
$t = 5$	C, G
$t = 7$	E
$t = 9$	E

at $t = 2$ ns — for execution. The process continues until the event queue is empty and the simulation is complete. Figure 2 shows the process in a flowchart of the traditional uniprocessor-based discrete-event simulation algorithm.

For many physical processes, a directed graph corresponding to the interactions among entities may assume the form of a cyclic circuit. For example, Figure 3 shows two cross-coupled Nand gates A and B constituting an RS latch (reset-set latch).

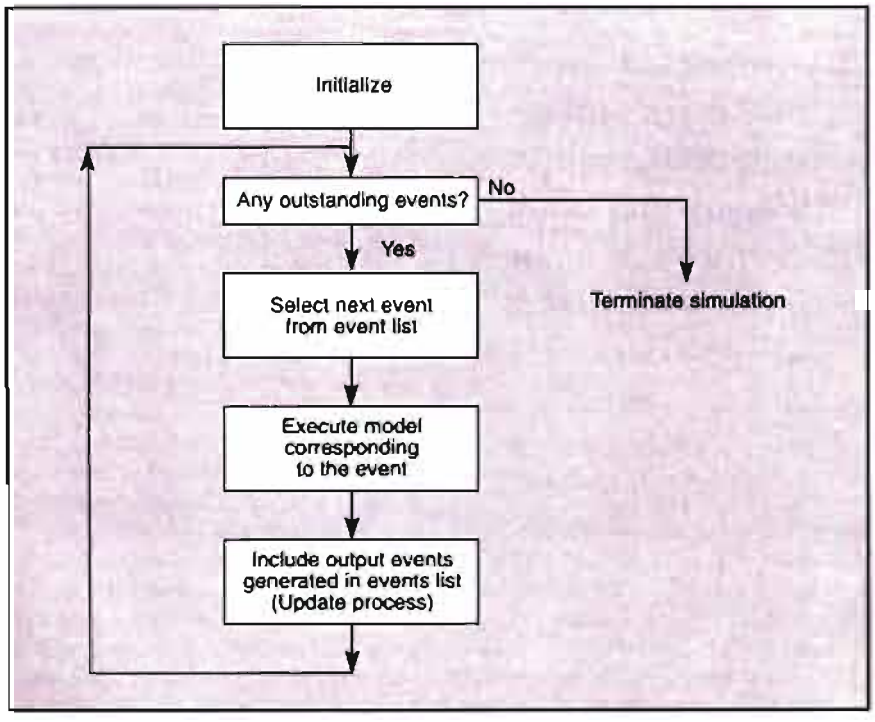


Figure 2. Flowchart of an algorithm for discrete-event simulation of digital hardware.

Assume propagation delays of 5 and 6 ns, respectively, for models A and B, and initial logical values of 0 at $t = 0$ ns at the output ports p and q. Corresponding to the signal transition at the primary input port x at $t = 0$ ns, the algorithm executes A, which generates a new signal value at its output p at $t = 0 + 5 = 5$ ns. Similarly, corresponding to the transition at the primary input port y asserted simultaneously at $t = 0$ ns, B executes and produces a new signal value at its output port q at $t = 0 + 6 = 6$ ns. Corresponding to the new signal values at p and q, models A and B execute again. If the signals at each of ports x and y are unchanged, A and B execute again at $t = 6$ ns and $t = 5$ ns to produce output signal values at ports p and q at $t = 6 + 5 = 11$ ns and $t = 5 + 6 = 11$ ns, respectively. Consequently, for each execution of A and B, the output signal produced may, in the future, cause subsequent execution of A or B.

Physical processes with cyclic circuits include all digital hardware designs with feedback, industrial control systems with negative feedback, oscillators, and sets of queuing networks interconnected in a closed loop. A well-known natural process with cyclic dependence is the food chain.

An example shows how a uniprocessor-based discrete-event simulation system for a cyclic circuit works. The digital hardware design in Figure 4 consists of three interconnected oscillators comprising the sets of gates $\{A,B,C\}$, $\{D,E,F\}$, and $\{G,H,J\}$. The propagation delay for each gate A, B, and C is 100 ns; for each gate D, E, and F, 3 ns; and for each gate G, H, and J, 5 ns. Initially, the logical value at the primary input of A is 0 and the event queue is empty. Corresponding to a signal transition 0 to 1 at the primary input of A at $t = 0$ ns, A is activated, and the algorithm includes the event in the event list in Table 2. The algorithm executes component A to produce a 1-to-0 transition at its output port at $t = 100$ ns. As a result, the event queue now contains the event at B at $t = 100$ ns. When component B executes at $t = 100$ ns, it generates an output transition at its output port at $t = 200$ ns. The event queue now contains an event at C at $t = 200$ ns. When C executes at $t = 200$ ns, it generates an output transition at its output port at $t = 300$ ns.

The event queue now contains three events: A, D, and G, each at $t = 300$ ns. Because the primary input of A is defined up to $t = 1{,}000$ ns, A executes again at $t = 300$ ns and generates an output transition at $t = 400$ ns, which causes an event at $t = 400$ ns in the event queue. Corresponding to the

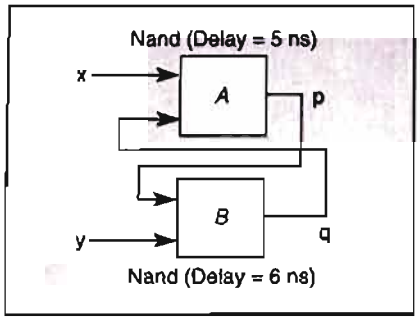


Figure 3. Cross-coupled Nand latch.

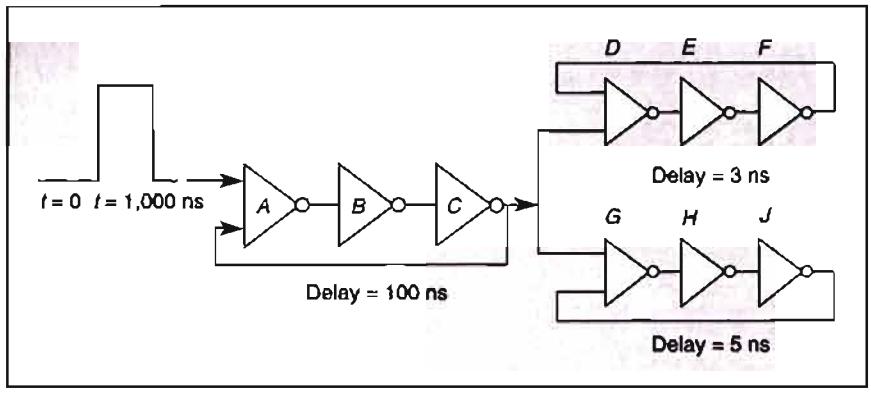


Figure 4. Discrete-event simulation of a cyclic circuit.

executions of D and G, output transitions are generated at the output ports at $t = 303$ ns and $t = 305$ ns, respectively. The event queue now contains three events — E, H, and B — to be executed at t equals 303, 305, and 400 ns, respectively. The simulation continues and terminates when either the event queue is empty or the simulation time exceeds the maximum simulation time.

The event queue in Table 2 shows that for the simulation time $t = 300$ ns, the algorithm can execute the set of components $\{A,D,G\}$ simultaneously. But in a uniprocessor system, the components must be executed sequentially. A parallel-processor system could execute the components in each of the sets concurrently, possibly speeding the simulation. A parallel-processor system might also speed simulation of acyclic circuits.

No reported parallel-processor algorithm for circuits in which the process interactions form a cyclic graph offers a solution with acceptable performance, freedom from deadlock, and provable correctness. In this article, we propose a method that uses a dataflow network synthesized on the basis of the connectivity of the circuit components. Our algorithm computes a quantity "time of next event" for each component, which permits the corresponding model to execute asynchronously as far ahead in simulation time as possible. The network ensures that a simulation process executing in a distributed processing environment will not deadlock.

Distributed discrete-event simulation

We studied distributed techniques because of their potential to simultaneously execute multiple entities of a complex discrete-event simulation and thereby speed the simulation. The synchronous, rollback, and asynchronous approaches are currently the three principal distributed techniques.

Synchronous mechanism. In the synchronous approach,[1] a processor designated as a centralized controller allocates all other entities to the processors of the parallel-processor system and initiates their executions. The controller also resynchronizes all processors at the end of every activity. An example of a system implementing the synchronous approach is the Zycad hardware accelerator machine, which uses the synchronous algorithm for gate-level logic simulation.

The synchronous approach permits the concurrent execution of entities corresponding to two or more events at the simulation time given by $t = t_1$, but it has some limitations. The processors must resynchronize at the end of each activity, even in the absence of data dependency, and message communication may not be complete at the end of an activity.

A synchronous distributed simulation of the example circuit in Figure 4 illustrates some of these problems. Assume that the components A through J are allocated to nine processors (one component per processor). A 10th processor of the parallel-processor system is the centralized controller that maintains the global event queue. Corresponding to the signal transitions at the primary input ports of A, the event queue contains a single entry: A at $t = 0$ ns. The controller initiates the execution of A. Except for the processor that contains the component A, the other eight processors are idle. Execution of A generates an output transition that causes an entry in the event queue: B at $t = 100$ ns. Then the controller initiates the execution of B, and the process continues as for a uniprocessor, with one exception. Unlike in a uniprocessor system, where the components A, D, and G must execute sequentially at $t = 300$ ns, in the synchronous approach A, D, and G can execute concurrently in three processors. However, all three must execute completely before the algorithm simulates the subsequent event E at $t = 303$ ns, followed by H at $t = 305$ ns. Components E and H are not data dependent, but the synchronous algorithm fails to achieve their simultaneous execution.

Table 2. Event queue for the simulation in Figure 4.

Time	Components Activated
$t = 0$	A
$t = 100$	B
$t = 200$	C
$t = 300$	A, D, G
$t = 303$	E
$t = 305$	H
$t = 400$	B

Rollback mechanism. The rollback mechanism[2] saves the state of the entire system periodically so the simulation system can roll back to its previous state if an error results from messages processed out of order. If a model has no information about a signal at an input port, it assumes that the signal value has remained unchanged and propagates to subsequent models the results of execution based on that assumption. If the component receives a subsequent message that contradicts its previous assumption, it propagates new results to subsequent models in the form of antimessages. Limitations of the rollback mechanism include the significant storage required to periodically save the state of the entire simulation and the uncertainty

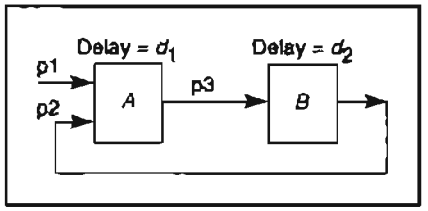


Figure 5. Asynchronous simulation of a cyclic circuit.

resulting from propagating a combination of messages and antimessages throughout the simulation system.

Asynchronous mechanism. The asynchronous discrete-event simulation mechanism[3,4] permits every simulation model to execute independently where there is no explicit data dependency, giving the potential for maximum parallelism. The example circuit shown in Figure 4 permits the following concurrent, independent executions: Component B may execute at $t = 400$ ns following execution of A at $t = 300$ ns. E at $t = 303$ ns following D at $t = 300$ ns, and H at $t = 305$ ns following G at $t = 300$ ns.

The next section reviews previous approaches to asynchronous discrete-event simulation of cyclic circuits. Then, we present our new approach and compare it with the previous approaches. We also comment on the proof of our algorithm's correctness, implementation issues, and our algorithm's performance.

Asynchronous simulation of cyclic circuits

The asynchronous approach permits every simulation model to execute independently in the absence of data dependency. Consequently, its success with a simulation system is a function of the computational requirement of the models and the degree of data dependency between the models. For a set of entities constituting a cyclic circuit, the output generated as a consequence of an entity's execution may influence its input at a later time. Consequently, compared with acyclic circuits, cyclic circuits require more synchronization, which may diminish the effectiveness of the asynchronous approach.

Figure 5 shows another serious difficulty in the asynchronous approach to cyclic circuits. The input ports of model A are p1 and p2, and the input port of B is p3. The output of model B connects to p2 of A, and the output of A connects to p3 of B. The propagation delays of A and B are d_1 and d_2, respectively. Assume distributed asynchronous discrete-event simulation of a design where A and B are associated with two distinct processors of a parallel processor. Simulation has run to termination when all externally supplied usable signal transitions have been used to generate output transitions.

Assume also that initial logical value 0 is associated with ports p3 and p2 at $t = 0$ ns. For a given signal transition 0 to 1 at port p1 of A at $t = 0$ ns, the algorithm executes A. Assume that the logical value of 1 at p1 persists up to $t = T$ ns, where T is very large. If the execution of A generates a transition at its output port at $t = 0 + d_1 = d_1$, the transition is propagated to B, causing B to be scheduled for execution at $t = d_1$. If the execution of B at $t = d_1$ generates a transition at its output port at $t = d_1 + d_2$, the transition is propagated to port p2 of A. As a result, A may execute again, and the process will continue as long as the algorithm can correctly schedule models A and B for execution.

If the execution of B at $t = d_1$ does not generate a signal transition at its output port, no message is sent from B to A. Consequently, model A is unaware that the port p2 is at logical 0 up to $t = d_1 + d_2$ and is unable to execute beyond $t = 0$ ns. The signal value of 1 at port p1 persists up to $t = T$ ns and, therefore, the simulation should execute until the transition $t = T$ ns. But A cannot execute without messages from B, and B cannot execute without messages from A. This constitutes a deadlock, caused by the absence of information at an input port of a model and the lack of global knowledge that the signal value at that port has remained unchanged.

A method called deadlock recovery[3] addresses the difficulty by letting the entire simulation system execute until it deadlocks, that is, until none of the entities is scheduled for execution and the overall simulation has progressed only partially. A distributed deadlock-detection algorithm[3,5] detects the deadlock state. The algorithm synchronously computes the minimum (say X) of all outstanding event times and the assertion times of external signals for every entity. Then, it lets every entity execute up to $t = X$ ns.

For the example circuit in Figure 5, deadlock recovery lets the entire simulation system constituted by entities A and B execute until it results in a deadlock. Assume that models A and B execute at $t = 0$ ns and $t = d_1$ ns, respectively, and that the execution of B does not generate a transition at its output. At the instant that the deadlock-detection algorithm detects the deadlock, component B has no outstanding events to execute, nor does it connect to an external input signal. Also, component A has no outstanding event to be executed, but the external signal at port p1 is defined at $t = T$ ns. Consequently, the algorithm computes the minimum X to be T ns and lets models A and B execute on the assumption that the signals at p2 and p3 have remained unchanged up to $t = T$ ns.

The deadlock-recovery scheme has limitations. The scheme fails for systems with both cyclic and acyclic circuits, that is, systems where not all entities may result in a deadlock. Moreover, issues of performance and correctness are difficult to resolve because entities execute into deadlock. Implementation[6] of the deadlock recovery scheme has shown that the simulation runs from one deadlock to the subsequent deadlock state and that the algorithm performance is nonlinear with respect to increasing problem size.

A second way[3] to handle deadlock is to identify and mark all entities of a simulation system that constitute cyclic subcircuits and set their execution modes to exception mode. In exception mode, an output signal generated when an entity executes is propagated to subsequent entities even when the signal is unchanged from its previous value. When an entity receives a message at an input port that corresponds to an unchanged signal value, the algorithm schedules the entity for execution exactly as for a message corresponding to a signal transition.

In simulation of the example design in Figure 5, first model A executes and then B executes. The execution of B generates no transition at its output. But the system is operating in exception mode, so a message corresponding to the unchanged signal value at p2 at $t = d_1 + d_2$ is propagated to A. Then, entity A executes again, generating an unchanged signal at its output at $t = 2d_1 + d_2$, which is subsequently propagated to B. The process continues until the external signal at p1 for $t = T$ ns is used and the output signals at ports p2 and p3 are appropriately determined.

This approach[3] guarantees absence of deadlock in a system with cyclic circles of any degree of complexity. Its principal limitation is its inefficiency, particularly when an external signal of a cyclic design that is nonoscillatory remains unchanged for a long time. For instance, assume that the example circuit in Figure 5 is nonoscillatory. Messages corresponding to

unchanged signal values will propagate from A to B and from B to A at intervals of $(d_1 + d_2)$ ns. The total number of iterations around the cycle until simulation terminates is approximately $(1{,}000/(d_1+d_2))$. The total CPU time required for simulation is proportional to the number of iterations, so when the ratio $(1{,}000/(d_1+d_2))$ is large, efficiency is low.

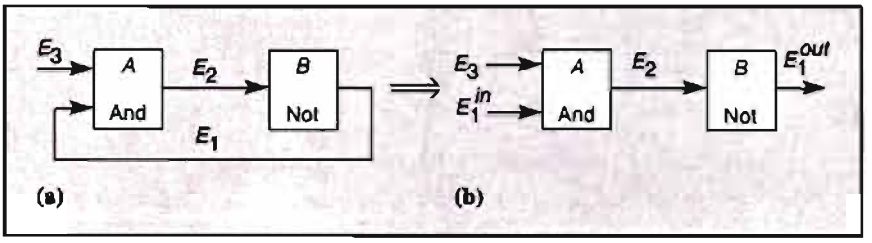


Figure 6. Reducing a cyclic directed graph (a) to an acyclic directed graph (b).

A new approach

We propose a new approach to avoid deadlock called Yaddes, which stands for "yet another asynchronous distributed discrete-event simulation algorithm." For a system such as a digital design, we identify subcircuits that constitute cyclic directed graphs and simulate only the entities of such subcircuits using the new approach. We simulate system entities that constitute acyclic graphs using the exception-mode approach[1] described in the previous section. In this article, we present the Yaddes approach for use with digital hardware, but it applies equally well to queuing networks and banking transactions.

Overview. Feedback loops are the principal cause of deadlock in traditional asynchronous distributed discrete-event simulation systems. The simulation environment represented through models connected by feedback loops cannot accurately decide the precise execution of events. To enable circuit execution in a deadlock-free environment, the Yaddes approach uses a synthesis of an acyclic circuit of pseudocomponents based on the original simulation circuit. Unlike simulation models that require substantial computational power, pseudocomponents are purely mathematical entities that evaluate functions. A pseudocomponent inherits only the input and output ports of the corresponding simulation model.

To preserve the algorithm's asynchronous and concurrent nature, each pseudocomponent represents a decision-making entity whose sole function is to determine when the corresponding simulation model can correctly execute an input event. An event refers to a signal transition at an input port. Yaddes requires that each pseudocomponent compute a quantity "time of next event" (W) at its output port. To do this, a pseudocomponent applies a minimum operator over the W values at its input ports and the simulation time of the event of the corresponding simulation model. Thus, the pseudocomponent must access the simulation time of the event from the related simulation model. This quantity is a measure of the time at which the next event is expected at that path. Furthermore, the algorithm can use this quantity to decide whether a model can safely execute an event. The minimum function shows the conservative nature of the Yaddes algorithm.

Corresponding to each of the inputs of the acyclic circuit that represent the primary inputs, the algorithm defines the W value as equal to the assertion time of the most recent transition. The remaining inputs of the acyclic circuit are unconnected because they are not influenced by any events in the circuit. Their W values are assumed to be permanently held at a very large number expressed as ∞ so they cannot influence the W computations of the pseudocomponents.

A limitation of the synthesized acyclic circuit is the lack of connectivity between the pseudocomponents of the respective feedback loops that may be required in the simulation circuit. For a given feedback loop, the W value at the output of the leftmost pseudocomponent does not reflect the simulation times of the events associated with other simulation models in the same loop or those of other models that may influence the computation. As a result, the computed W value may be inaccurate. In fact, the value will probably be "optimistic" and imply a value greater than the true value, for the following reason. Because the algorithm uses the minimum operator, W values it associates with other models imply only a lower value in the computation of the W value for a pseudocomponent.

To address this limitation, we synthesize a second identical copy of the acyclic circuit. To distinguish between them, we call the first and second acyclic circuits primed and unprimed, respectively, and express the quantity "time of next event" as W for the unprimed circuit. Each output of the primed circuit connects to each input of the unprimed circuit through a minimum operator. A crossbar switch expresses the dependency between the feedback loops in this interconnection network. If the activities of a feedback loop do not affect those of another loop, the corresponding link in the switch is considered nonexistent; otherwise, a link exists. An existent link has a weight equal to the computed propagation delay from the output of the primed pseudocomponent X' to the input of the unprimed pseudocomponent Y. Although the maximum capacity of the switch is $N \times N$, the actual size is defined by the circuit. (We discuss the role of the outputs of the unprimed circuit in a later section.) The W values computed by the pseudocomponents of the unprimed circuit correctly include the simulation times of all appropriate events in the entire circuit. The algorithm uses these values to accurately determine when an event can be executed by a model. We call the primed and unprimed circuits and the switch collectively the dataflow network for the circuit.

The optimistic nature of the evaluation process in the primed circuit acts as a window into future events. These future events are presented to the unprimed circuit, whose conservative characteristics guarantee simulation accuracy. The primary cause of deadlock — the cyclic data dependence in the feedback loops — is resolved by a dataflow network that lacks any cyclic dependence between its constituent pseudocomponents.

Yaddes algorithm. For a circuit containing feedback loops, first we identify a feedback arc set[7] S given by $S = \{E_1, E_2, \ldots, E_n\}$ of a directed graph corresponding to a digital design. This lets us render the graph as acyclic after we remove all the edges E_1 through E_n. The correctness of the approach is not contingent on identifying the minimal feedback arc set, which is difficult and time consuming to do. However, identifying the minimal feedback arc set may improve performance.[8]

For each E_i $\forall$ $i \in \{1, 2, \ldots, n\}$ in the original directed graph, we reconstruct a new acyclic directed graph by replacing E_i with two unconnected edges E_i^{in} and E_i^{out}. Figure 6a shows a cyclic circuit consisting

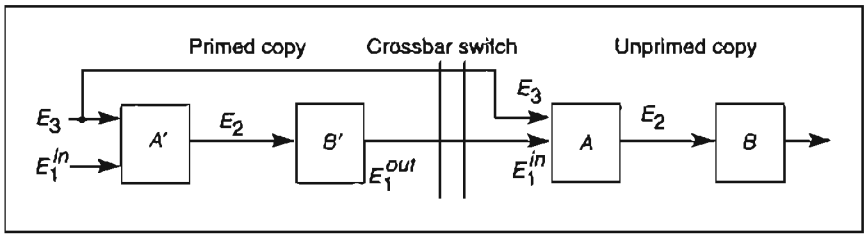


Figure 7. Dataflow network constructed for the cyclic circuit in Figure 6a.

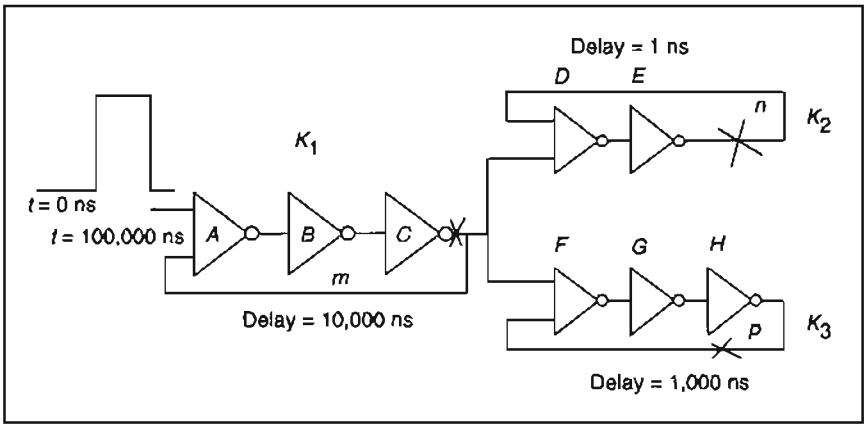


Figure 8. A digital design with cyclic subcircuits.

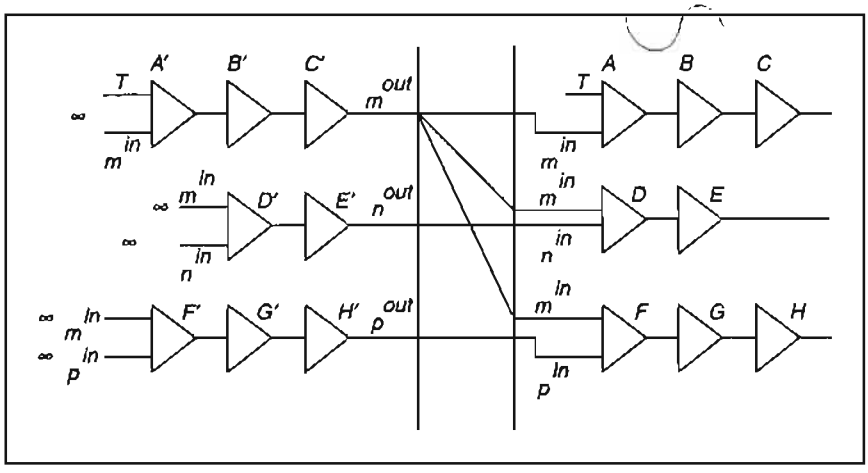


Figure 9. Dataflow network for the digital design in Figure 8.

of a two-input And gate A whose output connects through edge E_2 to the input of the inverter B. The output of the inverter B connects through edge E_1 to an input of A. The other input port of A is edge E_3.

Assume that the feedback arc set for the circuit is given by $S = \{E_1\}$. We rendered the graph in Figure 6b acyclic by removing E_1 and replacing it with E_1^{in} and E_1^{out} associated with the input of A and the output of B, respectively.

Next, we synthesized a dataflow network by connecting two identical copies of the acyclic circuit through a crossbar switch. The two acyclic circuits to the left of the crossbar switch are primed and to the right, unprimed. The entities in the dataflow network corresponding to the primed and unprimed circuits are the primed (X') and unprimed (X) pseudocomponents, where X refers to the corresponding simulation model. Every input port of a pseudocomponent X' that has a label of the form E_i^{in} is permanently held at a very large number represented by ∞. An output port of every X' that has a label of the form E_j^{out} is linked to every input port of any pseudocomponent Y in the unprimed circuit that has a label of the form E_k^{in}.

The collection of links constitutes the crossbar switch. For a feedback arc set of size N, the maximum capacity of the switch is given by N^2. Where the activities of the simulation models of a feedback loop may affect those of another loop, the corresponding link in the switch exists. Otherwise, it is nonexistent. A link connecting the output port E_i^{out} of a pseudocomponent X' in the primed circuit to an input port E_j^{in} of a component Y in the unprimed circuit merely propagates the W_x' values from X' to Y, delayed by the weight associated with the link. Figure 7 shows the dataflow network for the cyclic graph in Figure 6a.

In Figure 7, the pseudocomponents A' and B' constitute the primed circuit where the input port E_1^{in} of A' is permanently held at ∞. A' and B' correspond to the And and inverter gates in the simulation circuit. Pseudocomponents A and B constitute the unprimed acyclic circuit. The output port E_1^{out} of B' is connected via the crossbar switch to the input port of E_1^{in} of A because the activities of models A and B may affect each other. The feedback arc set has a size of 1, so the size of the crossbar switch is $1^2 = 1$. The first input ports of both A and A' connect to the external path E_3. Associated with E_3 are the externally applied signal transitions. Conceptually, we can include these transitions in the event list of model A — that is, the list of outstanding input transitions of A — and hold E_3 at ∞. Furthermore, conceptually the output port E_1^{out} of B is unconnected. However, in the current implementation of Yaddes the output of B is connected to a special entity "P_1" signifying that it is the rightmost boundary of the dataflow network. We say more about this connection later in this section.

Figure 8 shows three cyclic subcircuits. Figure 9 shows the dataflow network corresponding to their digital design. The digital design in Figure 8 consists of three cyclic subcircuits K_1, K_2, and K_3 constituted by the sets of components $\{A,B,C\}$, $\{D,E\}$, and $\{F,G,H\}$ respectively. K_1 and K_3 are oscillators with time periods of 30,000 and 3,000 ns, respectively, and K_2 is nonoscillatory. The oscillatory transitions of K_1 are

generated corresponding to a logical value of 1 at the input port of model A of K_1, and the transitions drive the models in K_2 and K_3. The feedback are set for the simulation circuit is $\{m,n,p\}$. Figure 9 shows the consequent acyclic subcircuits derived from K_1, K_2, and K_3 and the dataflow network. The input ports m^{in}, n^{in}, n^{in}, m^{in}, and p^{in} of the pseudocomponents A', D', and F' result from removing the edges m, n, and p of the original design. The input ports are all permanently held at ∞. The output port m^{out} of component C' connects to the input ports m^{in} of each of the pseudocomponents A, D, and F, reflecting the fact that the activities of the models of K_1 can influence those of both K_2 and K_3. The output port p^{out} of H' connects only to the input port p^{in} of F, reflecting the fact that the activity of K_3 influences neither K_1 nor K_2. Similarly, the output port n^{out} of E' connects only to the input port n^{in} of D. Consequently, the crossbar switch has only five links: C' to A, C' to D, C' to F, E' to D, and H' to F.

Associated with the output port of each pseudocomponent $X' \in \{A', B', ..., H'\}$ is a mathematical quantity "time of next event" represented by the symbol W_x'. (See the sidebar for mathematical definitions of W_x', U_x, and W_x.) Intuitively, W_x' is the predicted time of the next event at the output of model X. The algorithm computes it from the W' values at the input ports of X' and the simulation time of the event of model X. The computation of W_x' is triggered by any change in the values of its arguments. Any computed W_x' is propagated to other pseudocomponents connected to its fanout when it differs from the previous value. Moreover, the propagation is like a chain reaction: Subsequent components that intercept the W' values are also executed, and any changes in their output values are further propagated. The chain reaction terminates either when no new W' values are generated or when it encounters the rightmost component of the network. A process of acknowledgments detects the termination.

The W' values are optimistic. In the dataflow network in Figure 9, given that the second input port of A' is permanently held at ∞, the value of W_A' is defined only by the simulation time of the transition at the first input port of A'. We expect the value of W_A' to reflect the simulation time of the next event, yet it fails to take into account other events at B' and C' possibly with lower simulation times than that at A'. Similarly, the computation of W_D' fails to consider events at E' and even A', B', and C', because the activities of the feedback loop K_1 may affect those of K_2. Since the W' computation involves the minimum operator, failure to consider other events may yield values larger than the correct values. Moreover, these values are optimistic because they allude to events even when there may be other events with possibly lower simulation times.

Associated with each of the pseudocomponents $Y \in \{A, B, ..., H\}$ of the unprimed circuit is a similar mathematical quantity represented by W_y. The W values share the principles of computation and propagation of W' values. However, in contrast to the W' values, the W_x values are accurate. Therefore, the algorithm uses them to determine whether a simulation model event can be executed. Conceptually, the W values are accessed by the corresponding simulation models in the simulation circuit.

As with any distributed simulator, the algorithm stores the signal transitions received at the inputs of a simulation model in an event list for that model. The head of the list refers to the transition with the smallest value of simulation time. We also call this the event of the model, and represent the value of its simulation as U_x. In the Yaddes approach, every event of a model can be accessed by the corresponding primed and unprimed pseudocomponents.

The two major elements of the Yaddes

Formal definitions

Here are formal mathematical definitions of the quantities U_x, W_x', and W_x that we describe intuitively in our presentation of the Yaddes algorithm.

Definition of U_x. We associate with every simulation model X a collection of events — that is, transitions received at its input ports propagated from other models as messages. The algorithm orders the events in increasing order of their simulation times in an event list, and ultimately they may be executed by the model X. At any instant, U_x is equal to the simulation time of the event at the head of the list — the event with the smallest value of simulation time. When the list is empty, the value of U_x is considered equal to ∞. Initially, every $U_x \forall X$ in the simulation circuit is set to ∞. For the component A in the example circuit in Figure 8, assume signal transitions 0 to 1 at $t = 0$ ns and 1 to 0 at $t = 100{,}000$ ns at the external input of A. Also assume that other input ports of A receive no messages until then. Then U_A refers to the transition 0 to 1 at $t = 0$ ns, and $U_A = 0$.

Definition of W_x'. A mathematical quantity W_x' is associated with the output port of every primed pseudocomponent X' in the dataflow network. We compute it through the function $W_x' = \text{minimum}\ (U_x + d, W_1' + d, ..., W_n' + d)$ where W_1', ..., W_n' refer to the W' values at the input ports 1, ..., n of X, and d refers to the propagation delay of model X. W_x' usually is an optimistic and inaccurate measure of the simulation time when the next event will arrive at the output of model X. Initially, the algorithm sets every $W_x' \forall X$ to 0, indicating that they are not yet influenced by any event.

Definition of W_x. A mathematical quantity W_x is associated with the output port of every unprimed pseudocomponent X in the dataflow network. Formally, W_x is computed through the function $W_x = \text{minimum}\ (U_x + d, W_1 + d, ..., W_n + d)$, where W_1, ..., W_n refer to the W (or W') values at the input ports 1, ..., n of X, and d refers to the propagation delay of model X. In some cases, as with the unprimed component A in Figure 9, a W' value (in this case W_C') may be involved in the computation of a W value (in this case W_A). W_x represents an accurate measure of the simulation time when the next event will arrive at the output of model X. To preserve the correctness of the simulation — the proper order of event execution — no message with a simulation time given by $t < W_x$ can be sent by model X at its output port following the possible propagation of a message with simulation time $t = W_x$. Initially, the algorithm sets every $W_x \forall X$ to 0, indicating that they are not yet influenced by any event. We consider the simulation to be complete when W_x and $U_x \forall X$ are identical to ∞.

```
read in events at input ports — from external ports or other components
update event queue and order events according to time
if (new event alters U value) {
   initiate pseudocomponents X and X'
   wait until done signal received from X and X'
   send acknowledgment to the sender of the event
}
else if (new event does not alter the U value) {
   send acknowledgment to the sender of the event
}
read W values at every input port of the simulation model X and compute the
      minimum K
if (K value exceeds U value) {
   execute simulation model and generate output signal
   if (output signal does not differ from previous value) {
      remove U value and update event queue to reflect new U value
      if (event queue is empty) set U to infinity
      initiate X and X' to update W and W' values
      wait until done signals from X and X' are received
   }
   else if (output signal differs from previous value) {
      send output event to all models in the fanout
      wait for acknowledgment from each one of them
      remove U value and update event queue to reflect new U value
      if (event queue is empty) set U to infinity
      initiate X and X' to update W and W' values
      wait until done signals from X and X' are received
   }
}
```

Figure 10. Operations of a simulation model X.

```
read in command from simulation model X, new W' value from left, and
   acknowledgments from right

if (command from model X is read) {
   compute W'
   if (W' value remains unaltered) {
      send done signal back to model X
   }
   else if (W' computes to a new value) {
      propagate new W' value and expect acknowledgment from the receivers
      upon receiving acknowledgment, send done signal to simulation model X
   }
}
else if (new W' value is read) {
   compute the output W' value
   if (W' value is unchanged) send acknowledgment back to sender
   else if (W' computes to a new value) {
      propagate new W' value and expect acknowledgment from the receivers
      upon receiving acknowledgment, send acknowledgment to the sender on the
         left
   }
}
else if (acknowledgment received from the right) {
   if (acknowledgment is for itself) send done signal to simulation model X
   else if (acknowledgment is not for itself) relay it toward the original requestor
}
```

Figure 11. Operations of a pseudocomponent X' or X.

simulation environment are the simulation circuit and the dataflow network. The simulation circuit consists of executable models corresponding to each component of the circuit, and the flow of signals between the models is represented by messages over communication protocols. The models execute signal transitions received at the input ports, and any output transitions generated as a consequence of execution are propagated to other models connected to the output port. However, the constituents of the dataflow network generate decisions about the precise execution of an event. The primed and unprimed pseudocomponents execute concurrently and asynchronously with respect to one another and the simulation models. (In the current implementation of Yaddes, they are executed round-robin by a processor.) The execution of a pseudocomponent is initiated either by the corresponding model or by the propagation of a new W (or W') value at an input port by other pseudocomponents.

Because Yaddes is a distributed approach, the subalgorithms describing the operations of a simulation model, a primed pseudocomponent, and an unprimed component apply equally to all other respective entities in the system. Assume that a signal transition is asserted at an input port of simulation model X either by another simulation model or from the external world. When the algorithm incorporates this event in the event queue of X, the event either alters U_X or leaves it unchanged. When U_X is altered as a consequence of the incoming signal transition, model X is initiated for execution. The simulation model X initiates the corresponding pseudocomponents X and X' of the dataflow network for execution and suspends the execution of the event until the pseudocomponent executions are completed. X' evaluates W_X', and if its value has not changed from its previous value, X' propagates a message to the model X signifying that X' has completed its execution. If the new value of W_X' is a change from its previous value, X' initiates the chain reaction described earlier: It propagates the W_X' value to other pseudocomponents connected to its output port. When a subsequent pseudocomponent executes as a consequence of a new W' or W value asserted at an input port and generates a new W' or W value at its output port, it propagates the new output value to other pseudocomponents on the right through the crossbar switch if necessary.

Computation and propagation of new W' or W values at the output of pseudocomponents continue until either no new W or W'

values are generated or the rightmost boundary of the dataflow network is encountered. Then the pseudocomponents where the chain reaction terminates initiate acknowledgments and propagate them in the reverse direction. When other pseudocomponents that participated in the chain reaction receive acknowledgments, they take turns propagating them in the reverse direction. Eventually, X' (the primary initiator of the chain reaction) intercepts the acknowledgment and realizes that the process of updating W' or W values in the dataflow network caused by a change in W_x' has completed. It then sends a message to model X signifying that its execution is complete.

A pseudocomponent can be initiated even by new W or W' values asserted at its input port by other such components. The behavior of the pseudocomponent X following its initiation by model X is identical to that of X', except that the chain reaction is confined to only the unprimed acyclic network. Conceptually, pseudocomponents X and X' can be initiated concurrently by the simulation model X. Also, multiple simulation models can be executed simultaneously as a result of signal transitions at their input ports. Consequently, the computations of the W' and W values initiated by multiple pseudocomponents may overlap. Consistency and correctness are guaranteed because the computations involve a minimum operator and because the W value can never decrease.[9]

When both components X and X' have completed execution or when the signal transition asserted at an input port of model X does not alter its U value, the simulation model X sends an acknowledgment to the model that propagated the signal transition. If the signal transition was asserted externally, the acknowledgment indicates that the transition is being processed and requires the external world to send the subsequent signal transition to that primary input port.

Next, the simulation model X accesses the W or W' values associated with each of the input ports of the corresponding unprimed pseudocomponent and computes their minimum K_x. When $U_x \neq \infty$ — that is, an event exists at X and K_x exceeds U_x — the model can execute the event corresponding to U_x. When no new transitions are generated at the output port of model X following its execution, the algorithm deletes the event corresponding to U_x from the event queue and a new U_x reflects the time of the new event at the head of the event queue. When the event list of model X is empty, U_x is set to ∞. If a transition is generated at an output port as a consequence of execution of model X, it is propagated by X to other models that are connected to the output of X. Model X suspends further execution until it receives acknowledgments from each of the recipients. Then the algorithm removes U_x from the event queue and a new U_x is associated with the event at the head of the queue. The value of U is set to ∞ when the number of outstanding transitions at X is zero. The simulation model X then again initiates the pseudocomponents X and X' for execution and suspends further activity until the pseudocomponents have completed execution. The process continues until all usable external signal transitions at the primary input ports are used to generate output transitions.

Figures 10 and 11 present the precise functionalities of a representative simulation model and a corresponding pseudocomponent. The description in Figure 11 applies to both primed and unprimed components.

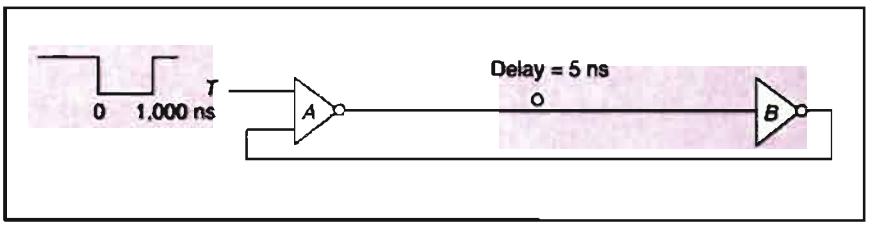


Figure 12. Example design for asynchronous distributed discrete-event simulation.

An example. Figure 12 shows how our asynchronous distributed discrete-event simulation algorithm works for an example design. In Figure 12, a Nand gate connects to an inverter through a feedback loop. The output of the Nand gate A connects to the input of the inverter B and the output of B connects to the second input port of A. The other input port of A is primary. A transition from high to low is asserted at $t = 0$ ns, followed by a low-to-high transition at $t = 1{,}000$ ns. The propagation delays of both A and B are 5 ns.

We assume that the initial values at the outputs of A and B are 0 and 1, respectively. For the given signal transition at the primary input of gate A, the outputs of both A and B change and remain stable thereafter. When simulated by a conventional asynchronous distributed discrete-event simulation algorithm,[1] gates A and B will deadlock as the signal values at the output ports do not change.

The feedback arc set for the circuit in Figure 12 is the arc from the output of B to the second input of A, and the dataflow network is constructed appropriately. Figures 13a through 13h are snapshots of the network as simulation progresses. The first input ports of both pseudocomponents A and A' are connected to T, and the second input port of A' is held at ∞. The output of B' connects to the second input of A through the crossbar switch.

The rectangular box above each pseudocomponent represents the event queue and contains the assertion time and transitions for every event. The U value (the right-hand entry in the box) for a pseudocomponent is the assertion time of the event at the head of the queue. For each of the snapshots, the figure also shows the computations of the W' and W values at the output of components. Associated with each unprimed pseudocomponent in Figures 13a through 13h is the computation of K, which is equal to the minimum of the W or W' values at the input ports of the component. The simulation models compute the K values, which are given in the figure to demarcate the states of the simulation models as simulation progresses.

Figure 13a describes the initial state where $U_A = U_B = \infty$ and $W_A' = W_B' = W_A = W_B = 0$. Assuming that the signal transitions at the primary input of simulation model A are not yet asserted, the value of K_x is the minimum of W_B' and the W value ($= \infty$) at the primary input of A. Thus, K_A computes to 0. The value of K_B is identical to W_x and computes to 0. In Figure 13b, the signal transitions are asserted at the primary input of the simulation model A and are represented through two events — $1{,}000\uparrow$ and $0\downarrow$ — of the event list of model A and pseudocomponents A and A'. The event $0\downarrow$ is at the head of the event queue and $U_x = 0$. Since the U_x value has changed from ∞ to 0, model A initiates pseudocomponents A and A' in Figure 13c. Pseudocomponents A and A' compute W_A and W_A', respectively. Because they differ

from their previous values, a chain reaction is initiated with the consequence that every W' and W is updated, as shown in Figure 13c. When the executions of the components A and A' are complete, model A computes K_A = minimum (∞,10) = 10, which exceeds the U_A value of 0. Consequently, the event 0↓ of A is simulated.

The model A executes the transition and generates a low-to-high transition at t = 5 ns at its output. In Figure 13d, an event 5↑ of model B represents the output transition of A. Model B initiates pseudocomponents B and B'. Because neither W_B nor W_B' changes, the executions of B and B' are immediately complete. Model B sends an acknowledgment to model A in Figure 13e. Then model A removes the already simulated event 0↓ from its event queue and updates U_A. The new value of U_A is 1,000, and model A again initiates pseudocomponents A and A'. The values of W_A' and W_A are updated, but those of W_B' and W_B remain unchanged. Model B computes K_B = minimum (W_A) = 15, which is larger than the U_B' value of 5. As a result, the event 5↑ at B can be simulated.

The execution of the transition by model B generates a high-to-low transition at t = 10 ns at its output. This is represented as an event 10↓ at the input of A, as shown in Figure 13f. Model A initiates pseudocomponents A and A', which compute a new value for W_A'. All other W and W' values remain unchanged. Thus, the executions of A and A' are complete, and model A sends an acknowledgment to B. The incoming event 10↓ at the second input of A displaces the event 1,000↑ at the head of the event queue and forces the new value of U_A to be 10. Model B removes the already executed event 5↑ and sets U_B = ∞ in Figure 13g. It also initiates pseudocomponents B and B', with the result that none of the W or W' values changes. In addition, model A computes K_A = minimum (W_B',∞) = 20, which exceeds the U_A value of 10. As a result, the event 10↓ can be simulated.

The simulation model A executes the transition but does not generate a new signal value at its output. Consequently, model A removes the executed event 10↓ from its event list, updates U_A, and then initiates pseudocomponents A and A'. The new value of U_A is 1,000. The quantities W_A' and W_A are computed to yield new values, and a chain reaction is initiated. The result is that W_B and W_B' values are also altered. Model A recomputes K_A = minimum (∞, W_B') = 1,010, which exceeds the U_A value of 1,000 with the consequence that the event 1,000↑ can be simulated. Although the entire design has stabilized and generated no new output values, the dataflow network computes updated values of W and W' that force the outstanding event — the external signal transition at t = 1,000 ns — to be simulated.

Comparison with previous approaches

In this section, we compare our algorithm with the two other principal algorithms proposed to avoid deadlocks.[13]

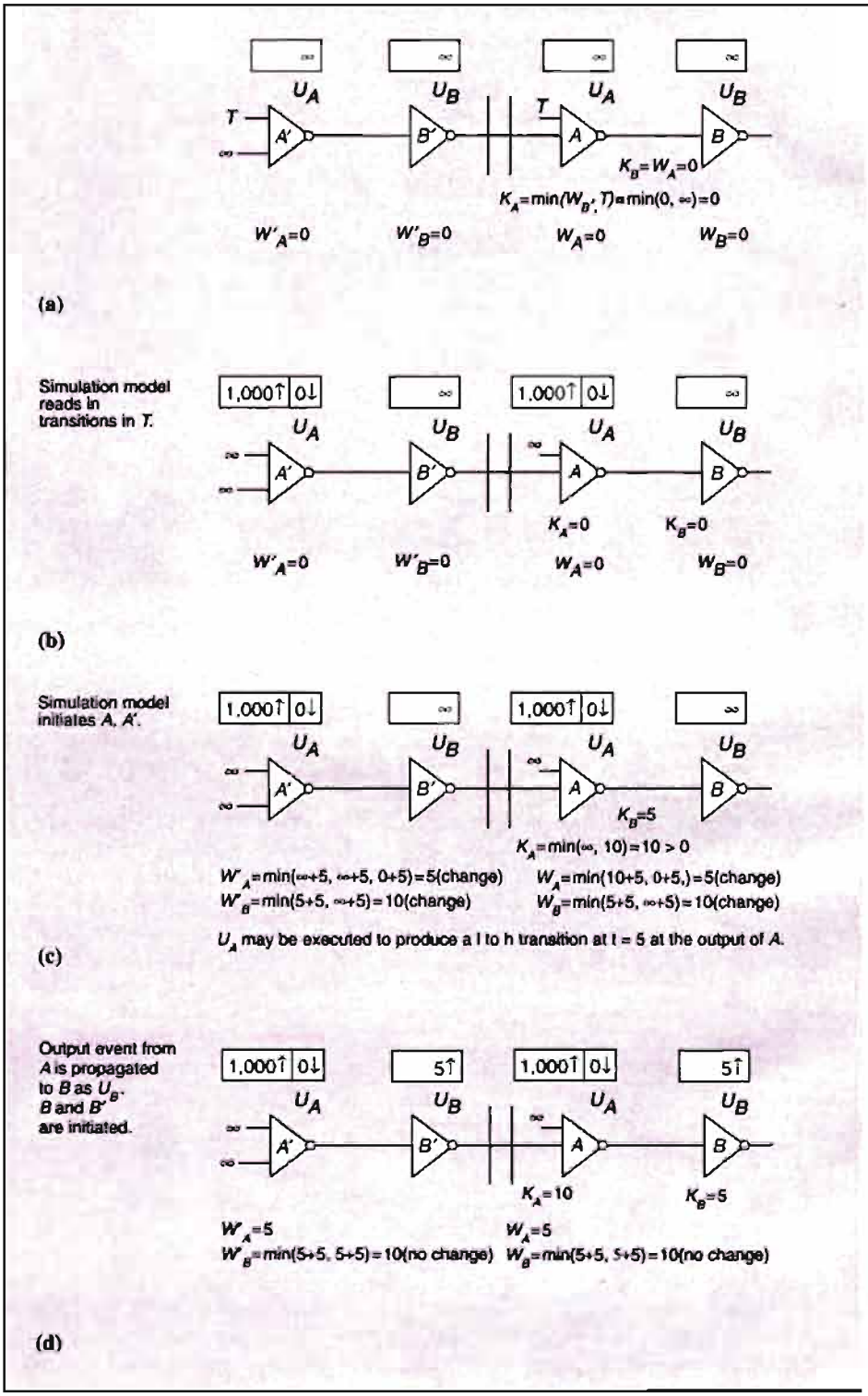


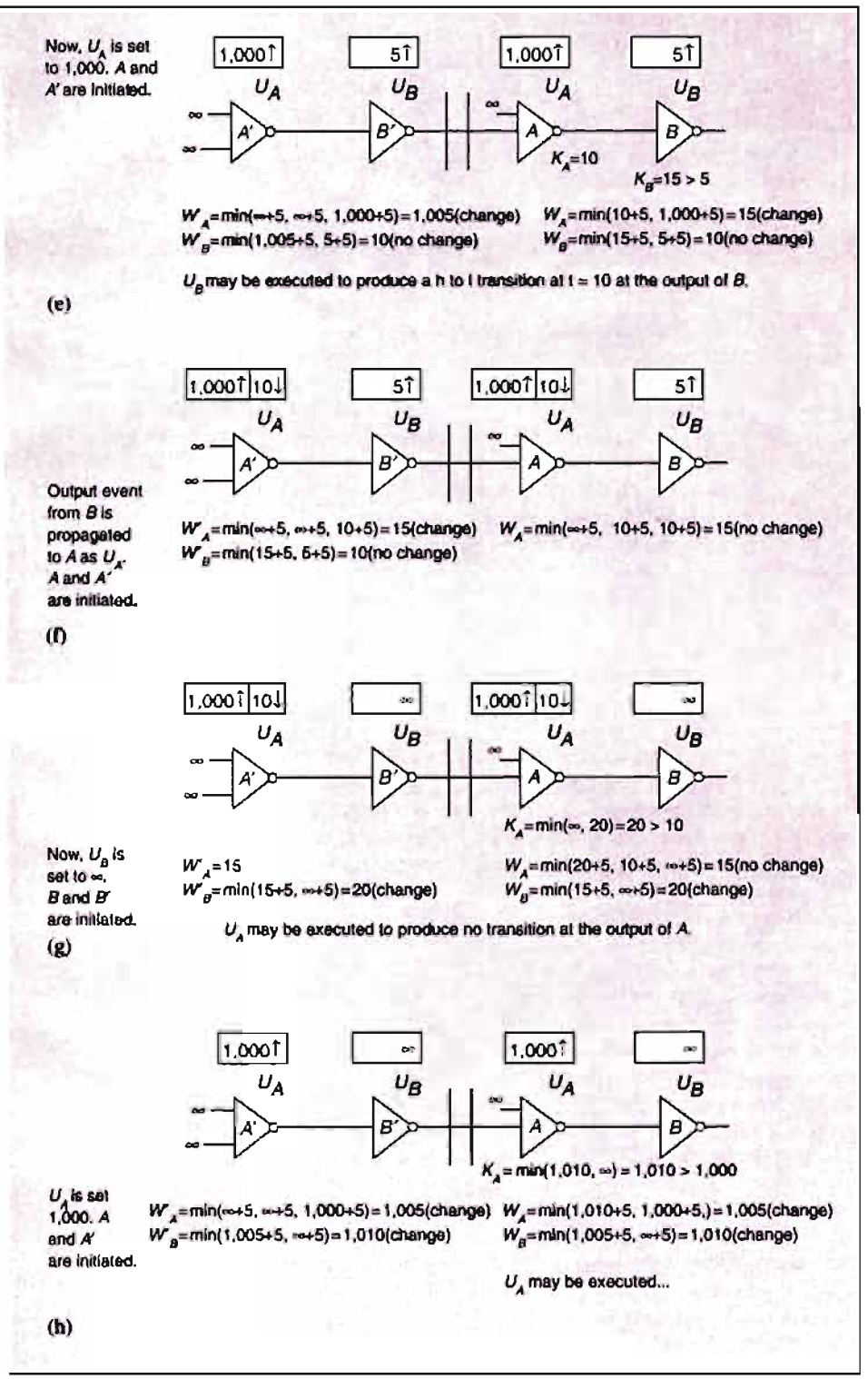
Figure 13. Snapshots of the simulation of the example design in Figure 12.

With the deadlock recovery algorithm,[5] a simulation model does not propagate any output signal information to other models connected to its output port when its value as a consequence of execution remains unchanged. Therefore, other models whose execution depends on the output value of this model may not execute, and a deadlock results. When such a deadlock occurs across the entire system, a distributed deadlock-detection mechanism detects the situation and a central entity synchronously accesses the U and W values of every model, computes their minimum, and permits execution of all models up to the minimum value.

In the Yaddes approach, the effect of any change in W or W' must ripple through the dataflow network as far as the effect can propagate. The crossbar switch implies transitive closure over all models in a feedback loop and guarantees that the U and W values of every simulation model that may possibly be affected by a change in the W or W' value of a model will be updated. In addition, the minimum operator used in the computation of every W', W, and K value ensures their correctness in the presence of multiple changes. The crossbar switch is equivalent to the traversal of all relevant loops in a design and computing the minimum over all relevant U and W values.

The exception-mode algorithm[3] sends messages with incrementally increased time values, even when the logical values at the outputs are unchanged. Consequently, the simulation time up to which every model is simulated is advanced continuously. The mechanism is necessary because a model cannot view the global picture and see, for example, an unchanged external input signal. A consequent limitation is the potentially large number of messages in the system when the external input signal remains unchanged for long periods of time relative to the cumulative propagation delays of the models in the feedback loop.

Yaddes, on the other hand, substitutes the global picture with the primed copy of the dataflow network. It permits optimistic jumps in the values of W', assuming that future events will be unable to influence and modify them. Normally such optimism leads to inconsistency and error, but the crossbar switch and minimum operator ensure the correct advancement of the W value.



Proof of correctness

The proof of correctness of the Yaddes algorithm requires the correct execution of the simulation models, execution of events in the correct order, absence of deadlock, and the termination of simulation in finite time. The execution of a simulation model implies the execution of the model description, so the accuracy of the simulation model's description also affects correctness. Complete details on the proof of correctness are presented elsewhere.[4]

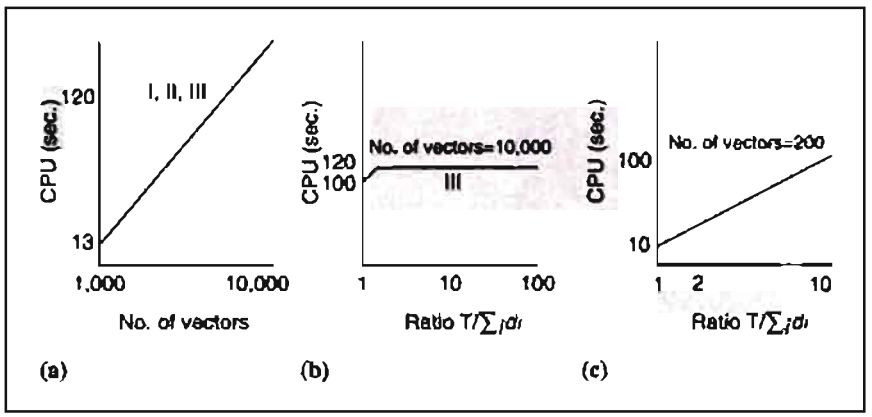


Figure 14. Yaddes performance measurements for a cross-coupled Nand latch (a, b). Performance measurements of the exception-mode algorithm for the same circuit (c).

Implementation

The implementation of the Yaddes algorithm is complex. Given any complex circuit and a user-specified partition, the total number of processors required for simulation equals $N + 2$, where N is the number of partitions. While the components of every partition execute on a processor, the algorithm models the primary inputs of the simulation circuit and the outputs of the dataflow network as entities P_0 and P_t, respectively, and executes them on unique processors. The entity P_t signifies the rightmost boundary of the dataflow network and participates in the propagation of acknowledgments. If a circuit contains feedback loops, a preprocessor that generates the dataflow network accepts the user-specified feedback arc set.

Corresponding to every simulation circuit component, the final implementation consists of three entities: a simulation model that represents the functionality of the component, and the primed and unprimed pseudocomponents. These are expressed through the C functions sim-component, ppe-component, and puc-component, respectively. Although they are conceptually concurrent entities, in the current implementation they are executed round-robin on a processor. When a partition includes multiple models, we express an interconnection between two or more models on the same processor through a data structure. When the models are located on separate processors, an interprocessor protocol represents the connection.

A significant part of the implementation consists of a kernel C description (approximately 2,500 lines) that executes on every processor except those that execute the entities P_0 and P_t. Each processor accepts a unique input file that represents information on the models and pseudocomponents and their interconnection for the corresponding partition. The input files for the partitions are generated by a preprocessor that accepts a description of the circuit in the hardware-description language ESL[19] and the user-specified partitions and feedback arc set.

During execution of the algorithm, the thread of control shifts from one entity to another. The algorithm first executes the simulation models — the sim-component functions — corresponding to the components that receive signal transitions from the external world at their primary input ports. A sim-component, in turn, initiates the executions of the puc-component and ppc-component functions, and suspends itself. When the executions of puc-component and ppc-component are complete, the sim-component is reactivated. The execution of a puc-component (or ppc-component) is complete when either the W (or W') value at the output is unchanged or an acknowledgment is received, signifying that the change in the output W (or W') value has been propagated throughout the dataflow network. Additionally, the algorithm may initiate the puc-component and ppc-component functions for execution when they receive from the left a new W (or W') value at any of their input ports. Eventually, the simulation process terminates when all events have been executed — that is, when the algorithm has used all externally supplied (usable) transitions at the primary input ports to generate output transitions.

Performance

In principle, the Yaddes algorithm can be implemented on any generic parallel processor — Ncube, Armstrong, Sequent, BBN-Butterfly, or transputers. However, the loosely coupled parallel-processor architecture is the most realistic model of an actual complex asynchronous system such as a self-timed digital design, banking system, or packet network. Our principal aim in this article is to present the Yaddes algorithm as the first successful approach to asynchronous distributed discrete-event simulation of cyclic circuits on parallel processors. Compared with other approaches, an asynchronous approach has the highest theoretical potential of using the most parallelism in a simulation. As with any asynchronous approach, the efficiency of Yaddes is realized for the simulation of systems whose models impose significant computational requirements and minimal need for data dependency or synchronization. Thus, Yaddes is not appropriate for systems whose models require minimal computation and frequent synchronization. The pseudocomponents of the dataflow network that are synthesized in the Yaddes approach are purely mathematical entities which merely evaluate functions. The significant computational load is still confined to the simulation models.

We implemented the Yaddes algorithm on the Armstrong[11] parallel-processor system at Brown University. Armstrong is a loosely coupled user-configurable parallel processor consisting of 100 processors. We reconfigured Armstrong as a six-dimensional hypercube.

The principal purpose of our implementation was to verify the correctness of the algorithm. In an experiment, we represented and simulated a latch constructed from two cross-coupled Nand gates. We set the propagation delay for each gate at 500 ns, with the consequence that the cumulative propagation delay through the feedback loop was 1,000 ns. We chose the external transitions (termed input vectors in the digital design discipline) asserted at the primary inputs of the latch so the latch experienced both stability and oscillation. For the experiment, we varied the number of external transitions at both primary inputs from 1,000 to 10,000 and measured the CPU time for each case. In addition, we selected three sets of average time periods for the external signals: 1,000, 10,000, and 100,000 ns.

The graphs in Figures 14a through 14c show the results. Figure 14a 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 expressing the average time period of external transitions (T) to the cumulative propagation delay around the loop ($\sum_i d_i$). Figure 14b shows the same results as in Figure 14a, except that the x axis represents different values of the ratio $T/\sum_i d_i$. Figure 14c shows the results of simulation of the same circuit using the exception-mode algorithm,[3] also implemented on the Armstrong parallel-processor system.

The graphs show 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 earlier algorithm.[3] We plan to publish full details of performance issues for a number of complex sequential circuits simulated with Yaddes. These will show that Yaddes is a mathematically proved algorithm applicable to any complex sequential system.

Asynchronous distributed discrete-event simulation of cyclic circuits has the potential to address problems in digital hardware design, queuing networks, and banking transactions. Until now, no reported algorithm offered freedom from deadlock and acceptable performance. The Yaddes algorithm, on the other hand, is mathematically correct and free from deadlock.

The Yaddes approach opens the possibility of modeling as discrete-event systems challenging problems from such disciplines as banking, railway and mobile phone networks, sociological interactions, human decision-making processes, aircraft simulation, oceanics, and weather forecasting. For example, we are studying the algorithm as a basis for distributed fault simulation using circuit partitioning, for distributed real-time banking systems, and for modeling large switching networks to investigate the role of overload conditions on network performance. ■

References

1. J. Misra, "Distributed Discrete-Event Simulation," *Computing Surveys*, Vol. 18, No. 1, Mar. 1986, pp. 39-65.
2. D. Jefferson, "Virtual Time," *ACM Trans. Programming Languages*, Vol. 7, No. 3, July 1985, pp. 404-425.
3. S. Ghosh and M.-L. Yu, "An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors," *Proc. Int'l Conf. Parallel Processing*, Pennsylvania State Univ. Press, University Park, Pa., 1988, pp. 74-77.
4. K.M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Comm. ACM*, Vol. 24, No. 4, Apr. 1981, pp. 198-206.
5. K.M. Chandy, L.M. Haas, and J. Misra, "Distributed Deadlock Detection," *ACM Trans. Computer Systems*, Vol. 1, No. 2, May 1983, pp. 144-156.
6. D.A. Reed and A. Malony, "Parallel Discrete Event Simulation: The Chandy-Misra Approach," *Proc. SCS Multiconference Distributed Simulation*, SCS, San Diego, Calif., 1988, pp. 8-13.
7. N. Deo, *Graph Theory with Applications to Eng. and Computer Science*, Prentice Hall, Englewood Cliffs, N.J., 1974.
8. H. Shih, P.G. Kovijanic, and R. Razdan, "A Global Feedback Detection Algorithm for VLSI Circuits," *Proc. Int'l Conf. Computer Design*, IEEE CS Press, Los Alamitos, Calif., Order No. 2079, 1990, pp. 37-40.
9. S. Ghosh, "An Asynchronous Distributed Discrete Event Simulation Algorithm for Cyclic Circuits Using a Data-Flow Network," LEMS Tech. Report No. 54, Eng. Division, Lab for Eng. Man/Machine Systems, Brown Univ., Providence, R.I., 1990.
10. S. Davidson and J. Lewandowski, "ESIM/AFS — A Concurrent Architectural Level Fault Simulator," *Proc. Int'l Test Conf.*, IEEE CS Press, Los Alamitos, Calif., Order No. 726, 1986, pp. 375-383.
11. J.T. Rayfield and H.F. Silverman, "Operating System and Applications of the Armstrong Multiprocessor," *Computer*, Vol. 21, No. 6, June 1988, pp. 38-52.

Erik DeBenedictis joined Ncube Corporation in 1990. Previously, he worked at AT&T Bell Laboratories and Ansoft Corporation, a company specializing in finite-element analysis.

DeBenedictis received a BS from Caltech, an MS from Carnegie Mellon University, and a PhD in computer science from Caltech. He is a member of the IEEE Computer Society.

Sumit Ghosh is an assistant professor in the Engineering Division and the Laboratory for Engineering Man/Machine Systems at Brown University. Previously, he was a principal investigator and member of technical staff at AT&T Bell Laboratories. His research interests include asynchronous distributed decision-making algorithms, fault simulation, test generation, distributed real-time banking, modeling and parallel simulation of broadband Integrated Services Digital Networks, distributed decision-making for military command control and communications systems, and hardware-description languages and environments for distributed execution on parallel processors.

Ghosh received his BTech in electrical engineering from the Indian Institute of Technology, Kanpur, and his MS and PhD in electrical engineering from Stanford University. A member of the IEEE Computer Society, he is vice chair of the design and test track of the IEEE International Conference on Computer Design.

Meng-Lin Yu is a member of technical staff at AT&T Bell Laboratories, Holmdel, N.J. His research interests include design automation, asynchronous circuit synthesis, parallel processing, testing and fault simulation, neural networks, and hardware accelerators for CAD.

Yu received his BS in electrical engineering from National Taiwan University and his MS and PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society and Phi Kappa Phi.

Readers may contact Sumit Ghosh, Division of Engineering, Box D, Brown University, Providence, RI 02912, e-mail sg@lems.brown.edu.