Report Number: WUCS-86-20

1986-11-01

A Unified Approach to Mixed-Mode Simulation

Roger D. Chamberlain and Mark A. Franklin

This paper presents a unified approach to mixed-mode simulation. It investigates the algorithms for both logic and circuit simulation, considering their similarities and differences, and a general framework is presented for integrating the two algorithms in uniform manner. The time advance mechanisms and component functional evaluations of the algorithms are show to be similar in nature, and mechanisms for the translation of information represented uniquely in the two algorithms are given. The resulting integrated algorithms is capable of performing mixed-mode simulation, where a circuit is partitioned into discrete and continuous regions, and each region is simulated at the appropriate... **Read complete abstract on page 2.**

A Unified Approach to Mixed-Mode Simulation

Roger D. Chamberlain and Mark A. Franklin

Complete Abstract:

This paper presents a unified approach to mixed-mode simulation. It investigates the algorithms for both logic and circuit simulation, considering their similarities and differences, and a general framework is presented for integrating the two algorithms in uniform manner. The time advance mechanisms and component functional evaluations of the algorithms are show to be similar in nature, and mechanisms for the translation of information represented uniquely in the two algorithms are given. The resulting integrated algorithms is capable of performing mixed-mode simulation, where a circuit is partitioned into discrete and continuous regions, and each region is simulated at the appropriate level. In addition, several of the issues relating to the implementation of mixed-mode simulation on multiprocessors are presented.

A UNIFIED APPROACH TO MIXED-MODE SIMULATION

Roger D. Chamberlain and Mark A. Franklin

WUCS-86-20

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This research was sponsored in part by funding from NSF Grant DCR-8417709, ONR Contract N00014-8D-C-0761, and an NSF Graduate Fellowship.

A UNIFIED APPROACH TO MIXED-MODE SIMULATION

by

Roger D. Chamberlain and Mark A. Franklin

Computer and Communications Research Center
Campus Box 1115
Washington University
St. Louis, Missouri 63130
(314) 889-6106

Abstract:

This paper presents a unified approach to mixed-mode simulation. It investigates the algorithms for both logic and circuit simulation, considering their similarities and differences, and a general framework is presented for integrating the two algorithms in a uniform manner. The time advance mechanisms and component functional evaluations of the algorithms are shown to be similar in nature, and mechanisms for the translation of information represented uniquely in the two algorithms are given. The resulting integrated algorithm is capable of performing mixed-mode simulation, where a circuit is partitioned into discrete and continuous regions, and each region is simulated at the appropriate level. In addition, several of the issues relating to the implementation of mixed-mode simulation on multiprocessors are presented.

Keywords:

logic simulation, circuit simulation, mixed-mode simulation, simulation architectures.

This research was sponsored in part by funding from NSF Grant DCR-8417709, ONR Contract N00014-8D-C-0761, and an NSF Graduate Fellowship.

A UNIFIED APPROACH TO MIXED-MODE SIMULATION

by

Roger D. Chamberlain and Mark A. Franklin

Computer and Communications Research Center
Washington University
St. Louis, Missouri

1. Introduction

The design of large digital systems and, in particular, the design of VLSI systems which require extensive simulation-based design verification prior to fabrication have increased the importance of simulation in the overall design process. As the systems being modeled have become larger, simulation tasks have become significant bottlenecks in the design cycle [1]. One approach to alleviating this problem has been to design special purpose machines tailored to the simulation process. Such machines typically have significantly higher performance than that obtainable with simulation algorithms executing on traditional sequential von Neumann architectures. Simulation in this context is taken to include both logic and circuit simulation. Logic simulation refers to discrete time-based simulation at the gate- or switch-level with node voltages being modeled by logical states. Circuit simulation refers to a continuous simulation, modeling circuit elements by their characteristic equations (or some simplification thereof) and numerically solving the differential equations that represent the circuit for the node voltages as a function of time.

There has been much work on the design of machines specialized to logic simulation [1-11] and circuit simulation [12-14] exclusively. In this paper we focus on methods for combining the two simulation tasks in a uniform manner so that they can be executed as part of a single simulation task (i.e., mixed-mode simulation). Such a unified development is needed to achieve

This research was sponsored in part by funding from NSF Grant DCR-8417709, ONR Contract N00014-8D-C-0761, and an NSF Graduate Fellowship.

high performance mixed-mode simulation on traditional von Neumann computers. In addition, such a development is a prerequisite for both effective design of mixed-mode simulation engines and implementation of mixed-mode simulators on general purpose multiprocessors (e.g., BBN Butterfly, N-Cube, Intel iPSC). Ultimately, of course, we would like to move in the direction of creating a "design automation" machine which can handle the range of computationally intensive tasks associated with VLSI design automation (e.g., logic simulation, circuit simulation, placement and routing, etc.).

Mixed-mode simulation has several potential benefits over standard techniques where circuit and logic simulation are treated separately. First, use of mixed-mode simulators eases the problems associated with setting up proper initial conditions on circuit simulations which are embedded within larger digital system models. In this situation, the logic simulation can be executed until required conditions are achieved, and then the circuit simulation can proceed with the proper input values and system state. In general, use of mixed-mode simulators greatly reduces the overall problems which can arise when two different but interacting simulation models are used by ensuring that node values at the interfaces of the models are correct.

Second, mixed-mode simulation can significantly reduce the costs associated with simulation for design verification. For example, large systems can be subdivided so that only those portions of the system requiring detailed level analysis are subject to circuit simulation while the remainder of the system is simulated at the logic level. The entire system need not be simulated at the circuit level (often this is not feasible due to large execution times) and, as each subsection is verified, it can be replaced with its logic level equivalent. Since circuit level simulation takes roughly two orders of magnitude longer then logic level simulation, overall simulation times can be reduced by successive replacement of circuit level models by logic level models.

Finally, in those situations where separate special purpose circuit or logic simulation engines are employed, the ability to perform mixed-mode simulation in a uniform manner holds

the promise of allowing the development of new engines which can do mixed-mode simulation. This could significantly reduce costs for the reasons given above, and because of potential savings when two simulation engines are replaced by a one.

This paper investigates the algorithms used to perform logic and circuit simulation. Their similarities and differences are considered, and a general framework is presented for developing an integrated algorithm which can handle both simulation types in a uniform manner. Given this framework, some issues relating to exploiting a multiprocessor for mixed-mode simulation are presented.

2. General Continuous and Discrete Simulation Algorithms

Simulation can be viewed as a series of state transitions that model the behavior of the system of interest starting from some initial state. This process can be represented as in Figure 1, where $S(t_n)$ represents the state of the system at time t_n, and $(\forall n)\,(n \geq 0 \mid t_n \leq t_{n+1})$. The simulator takes the system state at time t_n and derives the system state at time t_{n+1}. Within this general framework, the continuous and discrete simulation algorithms may be differentiated by the types of values that are allowed in the state variables and the methods employed to advance time. In the ideal continuous algorithm, state variables are treated as elements of the set of reals. This set is typically implemented using floating point numbers. In both the ideal and implemented discrete algorithm, state variables are elements of a countable set that can be mapped onto the integers.

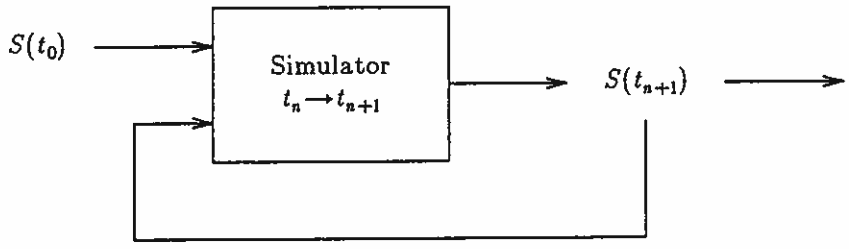


Figure 1. General simulation

There are two types of time advance mechanisms, both of which can be used with either the discrete or the continuous algorithms. The simplest approach is to advance time by a fixed amount at each step (i.e., $t_{n+1} - t_n$ is a constant). This is referred to as using a "fixed step size" when continuous algorithms are considered and is called a "unit increment" simulation in the discrete case. While simple to implement, this technique does not, however, take advantage of the fact that in both circuit and logic simulation there are often time points when little or no activity takes place (i.e., state variables remain the same). The second type of time advance mechanism exploits this temporal sparsity by skipping time points where there is no activity in the system. This mechanism is referred to as a "variable step size" in the continuous case and an "event based increment" in the discrete case.

In addition to the temporal sparsity mentioned above, not every component in the circuit need be evaluated even at time points that do have activity. This spatial sparsity of activity can be exploited in both the continuous and discrete algorithms by scheduling only those functional evaluations required rather than evaluating all components at time points with activity. In the discrete case, an event queue is typically used for this purpose.

The task of moving from time t_n to time t_{n+1} involves functional evaluation of circuit components in both the continuous and discrete algorithms. For both speed and compatibility reasons it is advantageous, where possible, to perform the functional evaluations using table lookup techniques. In this way, any special purpose hardware for table lookup that accelerates functional evaluation helps the performance of both circuit and logic simulation.

3. Circuit Simulation

In circuit simulation, the circuit of interest is modeled by a set of non-linear differential equations that are numerically solved as a function of time. Numerous options are available for solution of such sets of differential equations. Four possibilities for use as the circuit simulation algorithm are examined below. The criteria for comparing these methods are:

1. whether the algorithm takes advantage of temporal and/or spatial sparsity of activity inherent in the circuit,

2. whether the algorithm can be easily adapted for use in a distributed processing environment, and

3. whether the algorithm has proven stability and accuracy properties, or if these characteristics have been established using heuristics or empirical evidence.

3.1. Direct Methods

Direct methods take advantage of the improved speed possible using simple integration algorithms, simple circuit component models, and correspondingly simple hardware along with a small time step to insure convergence [12]. In this approach, the circuit is limited to interconnected transistors with non-zero capacitances allowed between any node and ground. The drain-source current, I_{DS}, is first calculated for each transistor connected to a node, using a simplified version of the Schichman and Hodges model [15]. The currents for all the transistors incident to the node are summed, divided by the capacitance at the node, C_S, and multiplied by the time step, Δt, to determine the change in voltage, ΔV. Therefore, $\Delta V = \frac{\sum I_{DS}}{C_S} \Delta t$. This operation is performed for each node in the circuit at each time step in the simulation.

The direct methods can employ table lookup techniques to quickly calculate I_{DS} at each time step and a special purpose machine can be built for this operation. This speed will (to some extent) compensate for the increased number of steps required by the method. A major disadvantage of this approach is the fact that the current and voltage calculations must be performed at every node at each time point. The algorithm thus does not take advantage of either temporal or spatial sparsity of activity in the circuit being simulated.

Lewis [12] presents a proposed hardware engine that uses such direct integration methods. If a process is considered to be some partition of the circuit to be simulated, interprocess communication is only required between neighboring processes (as defined by the circuit connectivity). This is advantageous when working in a distributed environment as

communications costs between processes running on distinct processors are often high. The accuracy of the method, though, is difficult to guarantee. Lewis reports timing errors of 20 % or more compared to an analytical solution for simple circuits.

3.2. Classical Methods

Another approach is to use implicit solution techniques to advance time from one step to the next. These techniques are similar to the algorithms prevalent on current von Neumann architectures [16]. The strict limits on circuit connectivity and component types associated with direct methods can be relaxed and more general circuits (even analog) can be simulated. The differential circuit equations are formulated by the equation $C\dot{v} - f = 0$ (Kirchhoff's current law), where C is the nodal capacitance matrix, $\dot{v}$ is the vector of time derivatives of node voltages, and f is the vector whose elements are the sum of the currents flowing into a node. An implicit solution method such as Backward Euler may be used to yield a set of non-linear, algebraic difference equations of the form $g(v) = 0$, where v is the vector of node voltages at the current simulated time. These equations in turn may be solved using the Newton-Raphson algorithm. At each iteration, Newton-Raphson yields a set of linear equations of the form $Av = b$, which can be solved using Gaussian elimination or more typically LU decomposition.

This technique has the advantage of allowing a larger step size than the direct methods, but requires a series of iterations at each time step to determine the node voltages at that step. This implies more time will be required to implement the iterations at each step, even if dedicated hardware is used to speed up the process. The actual step size can be made variable (limited by truncation error and stability considerations) to take advantage of temporal sparsity of activity. It has the same disadvantage as the direct methods in that every node must be solved at each time point, thereby decreasing the solution efficiency. That is, spatial sparsity of activity is not exploited.

The question of effectively implementing the Newton-Raphson algorithm and resulting LU decomposition on distributed architectures is still unresolved although certain recent

results [17,18] are encouraging. This technique does have excellent stability properties and proven accuracy [16].

3.3. Relaxation Techniques

The third possibility uses relaxation techniques to solve the set of differential equations describing the circuit [19]. A relatively new solution technique, called iterated timing analysis, has potential benefits in that it is more easily implemented on distributed architectures than the standard implicit methods. Given the same equation formulation as above, yielding the set of non-linear, algebraic difference equations $g(v) = 0$, the equations are solved using a modified Gauss-Seidel-Newton method (i.e., at each Gauss-Seidel (GS) iteration the individual difference equations, $g_i(v_1^k, \cdots, v_i^k, v_{i+1}^{k-1}, \cdots, v_n^{k-1}) = 0$, have their solution approximated by a single Newton-Raphson iteration). This alleviates the need to linearize the equations and perform an LU decomposition on the resulting matrix. The elimination of the LU decomposition step ensures that only nearest neighbor (as defined by the connectivity of the circuit) communications are required, thereby simplifying parallel implementations of the algorithm. In addition, the algorithm is event driven (i.e., an event corresponds to a single Gauss-Seidel iteration at a node for a particular time point), thus providing a direct similarity to discrete-event logic simulation algorithms. Local truncation error estimates can be used to vary the time step of the solution, which can differ for distinct nodes, thereby exploiting both the temporal and spatial sparsity of activity inherent in the circuit.

A top level view of this algorithm is given in Figure 2. Newton [19] gives a proof of the stability and accuracy of the method, along with a discussion of other relaxation techniques, such as timing analysis and waveform relaxation.

3.4. The ADEPT Algorithm

The final possibility considered here is called the ADEPT algorithm. The algorithm alters the knowns and the unknowns in the circuit equations by using voltage to solve for time, rather

```
while (termination criteria not met)
    retrieve event with smallest time value from event queue
    update simulated time
    obtain v_{i,t}^k from g_i(v_1^k, ..., v_i^k, v_{i+1}^{k-1}, ..., v_n^{k-1}) = 0 using a
        single Newton-Raphson iteration
    if convergence is achieved (v_{i,t}^k - v_{i,t}^{k-1} < ε) then
        if node is not stable (v_{i,t} - v_{i,t-Δ} > δ) then
            schedule node evaluation event (GS iteration 0) in
                event queue for a future time
        endif
    else
        schedule node evaluation event (GS iteration k+1) in
            event queue at the current time
        for all nodes that can be affected by node just evaluated
            if not scheduled for evaluation then
                schedule node evaluation event (GS iteration 0) in
                    event queue at the current time
            endif
        endfor
    endif
endwhile
```

Figure 2. Iterated timing analysis algorithm

than the other way around as in the prior methods. In this technique, described by and Odryna [20], and Vidigal [21] under the name of CINNAMON, the non-linear equations above are linearized to yield a system of linear differential equations $C\dot{v} + Gv = i$, where C and G are n by n matrices and i is an n–vector. The analytical solution is a function of the matrix $e^{C^{-1}Gt}$. The circuit is then partitioned into smaller subcircuits that have a small amount of coupling between them, C^{-1} is calculated for the subcircuits, and the amount of time for a node to change by a fixed voltage interval ΔV is determined. This time is used to schedule an event updating the node voltage.

The solution is explicit, eliminating the cost of iterations, and the solution algorithm is event driven (i.e., an event corresponds to an update of a node voltage at a particular time). This allows for the exploitation of both temporal and spatial sparsity of activity. Nearest neighbor communication is all that is required, improving the ease with which it can be implemented on a distributed architecture. Note that the voltage interval chosen at the

beginning of the simulation can fundamentally determine the accuracy of the simulation. This can be alleviated to some extent by a variable voltage step (analogous to the variable time step used in the implicit methods), but to date there have been no published reports that bound the error in the results, only empirical evidence that the error can be kept at reasonable levels.

3.5. Algorithm Comparison

Table 1 summarizes the essential features of each of the continuous simulation algorithms considered. The first two approaches do not take advantage of temporal and spatial sparsity of activity. The ADEPT algorithm does not have proven accuracy. The iterated timing analysis (ITA) algorithm both takes advantage of temporal and spatial sparsity and has guaranteed and provable accuracy. This algorithm is further explored in this paper as the approach with the greatest potential benefits.

4. Logic Simulation

The logic simulation algorithm simplifies the model of the electrical system being simulated by replacing continuous voltage and conductance information with a set of logical states. Changes in voltage or conductance are then represented by discrete jumps from one state to another, and components are evaluated only when one or more of their inputs changes state.

Table 1. Continuous simulation algorithms

Algorithm	takes advantage of: temporal sparsity	takes advantage of: spatial sparsity	proven accuracy
Direct	n	n	n
Implicit	y	n	y
ITA	y	y	y
ADEPT	y	y	n

4.1. Logic Simulation Algorithm

A top level view of the discrete-event simulation algorithm applied to logic simulation is given in Figure 3. In this algorithm, an event corresponds to a change in the output of a component. A node evaluation is required to handle wired-logic constructions. Given proper component models this algorithm can handle general MOS circuits, including bidirectional components and shared charge [22]. An alternative logic simulation algorithm due to Bryant [23] is the MOSSIM algorithm. It is primarily intended for switch-level simulation and does not gracefully handle circuit components at the gate level (i.e., NAND, NOR, NOT gates).

Both temporal and spatial sparsity of activity are exploited since only those components whose inputs change are evaluated (i.e., areas of the circuit with no activity at a time point do not have any computational cost associated with them). The distributed algorithm only requires nearest neighbor communication, and several schemes for its implementation are given in [8]. The general purpose discrete-event simulation algorithm has been proven to be correct by Misra [24], with the logic simulation algorithm presented in Figure 3 being a specific instance of that algorithm.

```
while (termination criteria not met)
    retrieve event with smallest time value from event queue
    update simulated time
    update component output associated with event
    evaluate node connected to component output
    for each component input connected to node
        evaluate component
        if component output changes from current value then
            schedule output change event in event queue
        endif
    endfor
endwhile
```

Figure 3. Logic simulation algorithm

4.2. States and Strengths

The state of a node consists of a pair of values, a level (representing the voltage) and a strength (modeling the conductance from the node to one of the power rails). The choice of the number of levels and strengths to be used depends upon the amount of detail the discrete model is intended to support. Starting with a minimum [25] of three levels,

1	high
0	low
x	unknown {0,1}

additional levels can be included in order to be able to detect design faults that are undetectable with the minimum set (e.g., multiple threshold drops before driving the gate of a transistor). The basic algorithm does not change regardless of the number of levels used, although there is an effect on the table sizes required to implement the functional evaluation of components. The minimum set above will be used for illustration purposes in this paper.

In order to handle the wired-logic constructions and charge storage inherent in MOS circuits, multiple node strengths must also be supported. Starting with a minimum of three strengths,

s	strong
w	weak
f	float

additional strengths can be supported without changing the basic algorithm (e.g., to support charge sharing). Once again the minimum set is used here for illustration purposes.

A state can be considered a point in the level, strength space where the full set of states is the cross product of the level set and the strength set. Although not all MOS logic simulators include all nine possibilities of the minimum set of states explicitly, they are typically supported implicitly in their implementation of the simulation algorithm (e.g., the set of states used in the author's *lsim* [26] does not include the $(0,f)$ and the $(1,f)$ state explicitly, but the implementation still supports charge storage).

In the illustrative set of states, a weak high is represented by the state $(1,w)$ and a strong low is represented by $(0,s)$. Wired logic connections are handled by defining a lattice on the set of states shown by the Hasse diagram of Figure 4. The resulting state of a node is then determined by taking the least upper bound of all the states driving the node. For example, if two components outputs are driving a node, one $(1,w)$ and the other $(0,s)$, the resulting state is $(0,s)$.

To perform functional evaluation of components, it is often not necessary to know the strength of its input nodes, just the level. For this reason the table sizes used to implement the functional evaluation are dependent upon the number of levels, not the total number of states.

5. Integrating the Continuous and Discrete Algorithms

The task of combining the continuous and discrete algorithms can take advantage of the following commonality:

1. Both algorithms rely on a common time advance mechanism.
2. Both algorithms use table lookup techniques for function evaluation.

The remaining tasks are to provide a method for the user to partition the circuit into continuous and discrete regions and to define mechanisms for translating the voltage and conductance information present in the continuous region into state information for the discrete

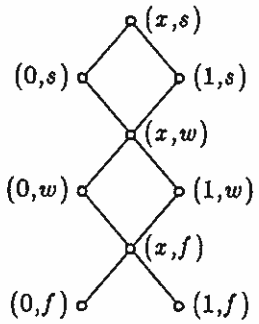


Figure 4. Logical states

region and visa versa. The sections below point out the common features of the two algorithms and provide a mechanism for the required translations referred to above.

5.1. Time Advance Mechanisms

A unifying feature of both the discrete and continuous algorithms is the presence of an event queue (see Figures 2 and 3). For the discrete case, an event consists of the change in state of a component output at a particular time point (e.g., (t_1, inverter1, (0,s))). In the continuous case, an event indicates the need to perform an iteration of the Gauss-Seidel solution at a node at a time point (e.g., (t_1, node 1, iteration 4)). The ordering of events in the event queue can then be based on time, with multiple events at a given time point being ordered on the basis of Gauss-Seidel iteration. Time is advanced in the simulation by removing the lowest ordered event in the queue and updating the simulated time clock to correspond to the time of event just removed from the queue.

5.2. Functional Evaluation

The functional evaluation of components in the both continuous and discrete domains can be performed using table lookup techniques. In the continuous domain, node voltages are used as entries into a table to determine branch currents for circuit components. The additional step of performing a Newton-Raphson iteration on the node equations is then required. In the discrete domain, the logic levels are used as entries into the table to determine component outputs.

5.3. Circuit Partitioning

The partitioning of the circuit into continuous and discrete regions is handled by having the user specify the set of components that are to be in the continuous region and the set of components that are to be in the discrete region. Nodes incident to continuous components are represented by voltages, nodes incident to discrete components are represented by logical states, and nodes that are incident to both continuous and discrete components are points where the

translations between voltage and states must occur.

There are some components that typically do not have a continuous model that completely describes the functionality of the component (e.g., gate-level components such as NAND and NOR gates and MSI-level components such as adders and multiplexers). When these components are included in a continuous region, a continuous interface is used at their inputs and outputs (e.g., a load capacitance at the inputs and Thevenin equivalent at the outputs) and the discrete model is used to represent the functionality of the component. The voltage-state translations (described below) are then utilized to map the available continuous information into the discrete information required for component functional evaluation. The reverse translations are used to map the discrete output of the component into continuous information required for the evaluation of the node connected to the component output.

5.4. Continuous and Discrete Model Interactions

There are two fundamental interactions which can occur between variables in a discrete and continuous model:

1. A discrete change may be made to a continuous variable due to actions within the discrete model.

2. A change in a discrete variable may be made due to a continuous variable achieving a threshold.

In logic and circuit simulation, the variables of interest typically are node voltages and branch currents. The values associated with each node can be either continuous or discrete variables depending upon how the user has partitioned the circuit.

For examples of these interactions consider the circuit of Figure 5. Assume both inverters, the primary input at node 1, and the primary output at node 5 have been identified as being in the discrete region and the pass transistor and the primary input at node 3 are in the continuous region (within the dashed lines). This implies that a discrete to continuous translation will have to be performed at node 2 and a continuous to discrete translation will

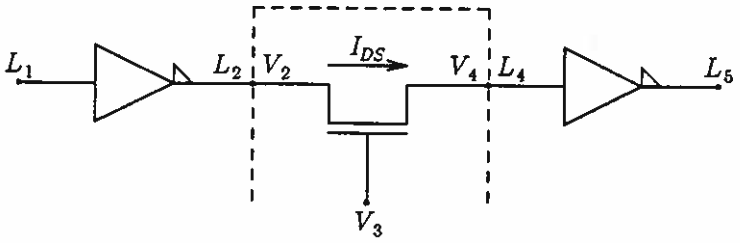


Figure 5. Interaction example

have to be performed at node 4. Nodes 1 and 5 are discrete nodes and node 3 is a continuous node. The domain of variables is denoted by an L, to represent logic level, for variables in the discrete domain and a V, to represent voltage, for variables in the continuous domain. If the logic level of node 2, L_2, is changed from "1" to "0" as a result of a change in the inverter output, there is a discrete change in the voltage associated with node 2, V_2. If the voltage at node 4, V_4, subsequently falls below a certain threshold value, say V_l, the logic level of node 4, L_4, will be changed as well. The details of implementing these interactions are described below.

5.5. Mapping States to Voltages and Back Again

We now consider the task of mapping logic levels into voltages and visa versa. To implement the translations, a pseudo-component is placed at each node where the translation takes place. The node is therefore separated into a discrete node and a continuous node with a pseudo-component connecting the two. The continuous to discrete pseudo-component is relatively straightforward. Given an input node voltage and conductance, its output is the logical state corresponding to that voltage and conductance. The pseudo-component can have a capacitor to ground on its input to model the load capacitance of the discrete output node. The diagram of such a component is given in Figure 6.

To support this, there are two user definable values that are used in the voltage to logic level mapping, V_h and V_l. V_h is defined as the lowest voltage that, when supplied at a component input, is interpreted as a logical high. V_l is defined as the highest voltage that,

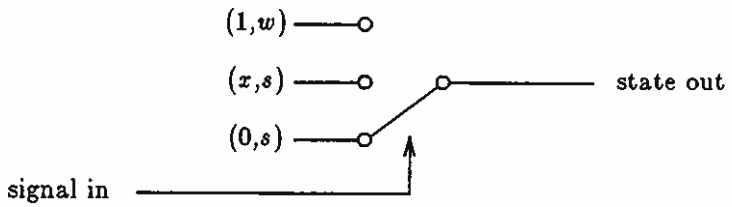


Figure 6. Continuous to discrete pseudo-component.

when supplied at a component input, is interpreted as a logical low (e.g, for TTL, $V_h = 2.4V$ and $V_l = 0.4V$). Given a continuous voltage V, the mapping from V to a level L is as follows:

$$L = \begin{cases} 1 & \text{if } V \geq V_h \\ x & \text{if } V_l < V < V_h \\ 0 & \text{if } V \leq V_l \end{cases} \quad (1)$$

Additional logic levels may be handled by adding additional threshold values.

The conductance to logic strength mapping is similar. Given a conductance, G, to power or ground, the function below translates it into a logic strength S:

$$S = \begin{cases} s & \text{if } G \geq G_h \\ w & \text{if } G_l < G < G_h \\ f & \text{if } G \leq G_l \end{cases} \quad (2)$$

where G_h is a user supplied threshold conductance to distinguish between the conductance of a pullup and pulldown transistor. For NMOS technology, a reasonable value for G_h would be $\frac{1}{R_m/2}$, where R_m is the resistance of a minimum dimension transistor. G_l is intended to distinguish between nodes that have an active transistor from the node to power or ground and nodes that are floating (i.e., the voltage is maintained by trapped charge on a non-zero capacitance from the node to ground). It is typically zero or near zero. Note that G_h and G_l are not global values as are V_h and V_l above, but are functions of circuit location (i.e., they must be determined individually for each pseudo-component forming the continuous to discrete interface).

Discrete logical state to continuous voltage and conductance require that assumptions be made when translating ambiguous logic levels into a necessarily unambiguous voltage. A mapping from level L to voltage V is given below:

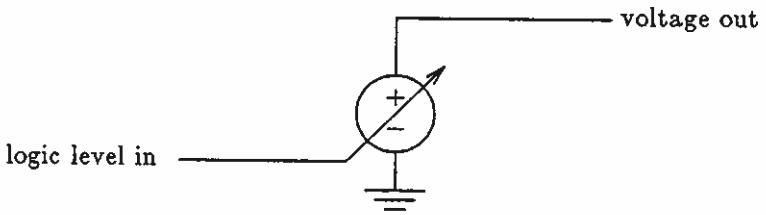
$$V = \begin{cases} 5.0 & \text{if } L{=}1 \\ \dfrac{V_h + V_l}{2} & \text{if } L{=}x \\ 0.0 & \text{if } L{=}0 \end{cases} \tag{3}$$

The first and last values are straightforward. The assumption made in translating the ambiguous level into a voltage assigns the value for "x" to be the midpoint between V_h and V_l. The diagram of a generalized pseudo-component that implements this mapping is presented in Figure 7 (a) [27].

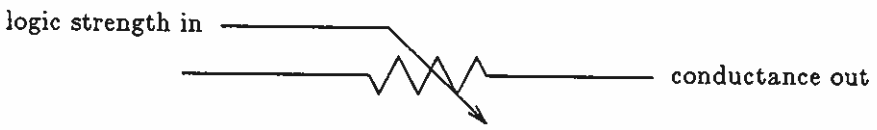
The mapping from strength, S, to conductance, G, is given below:

$$G = \begin{cases} G_s & \text{if } S{=}s \\ G_w & \text{if } S{=}w \\ G_f & \text{if } S{=}f \end{cases} \tag{4}$$

The values of G_s, G_w, and G_f are also circuit location dependent. The diagram of a pseudo-



(a) Logic level to voltage pseudo-component.



(b) Strength to conductance pseudo-component.

Figure 7. Discrete to continuous pseudo-components.

component that implements this mapping is presented in Figure 7 (b).

An alternative approach is to use a logic level-controlled switch [28], illustrated in Figure 8. The logic level of the input node is used to control a switch that connects one of a set of voltage sources to the output node through a given conductance. This approach implies that component output logic strength, and therefore output conductance, is a function of the output logic level.

6. A Simple Example

To better illustrate the relationship between the algorithms, the circuit of Figure 5 will be used as an example to step through the operations performed when executing a mixed-mode simulation. In this example, the pass transistor will be considered a continuous component and the two inverters will be discrete components. As stated previously, this implies that a discrete to continuous pseudo-component will be placed at node 2 and a continuous to discrete pseudo-component will be placed at node 4. Figure 9 gives the revised circuit explicitly indicating the existence of these pseudo-components.

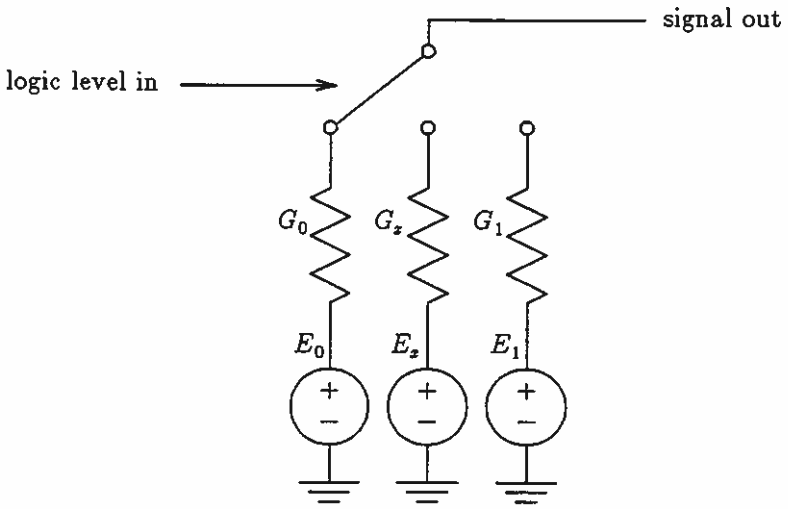


Figure 8. Logic level-controlled switch.

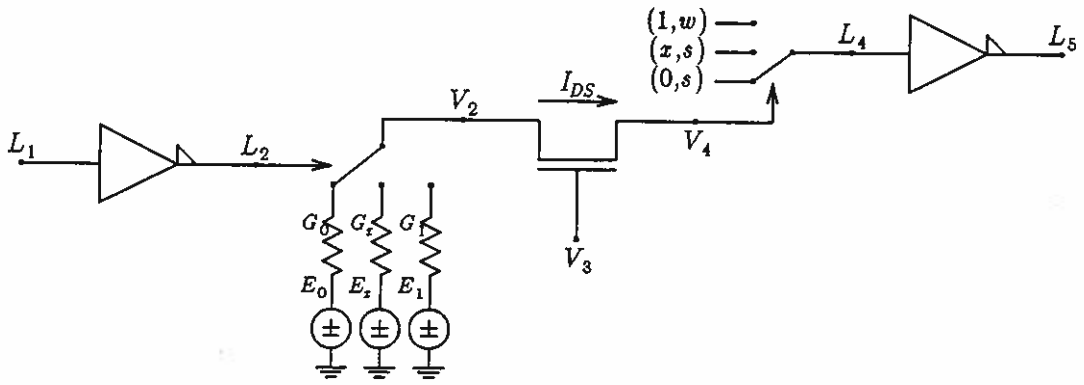


Figure 9. Simple example

The non-linear, algebraic difference equation $g_2(v) = 0$ for node 2 is

$$C_2 \cdot \left(\frac{V_{2,t_m} - V_{2,t_{m-1}}}{t_m - t_{m-1}} \right) + I_{DS}(V_{2,t_m}, V_{3,t_m}, V_{4,t_m}) - EG = 0 \tag{5}$$

where E and G are the voltage source and conductance within the discrete to continuous pseudo-component and C_2 is the capacitance to ground associated with node 2. The equation $g_4(v) = 0$ for node 4 is

$$C_4 \cdot \left(\frac{V_{4,t_m} - V_{4,t_{m-1}}}{t_m - t_{m-1}} \right) - I_{DS}(V_{2,t_m}, V_{3,t_m}, V_{4,t_m}) = 0 \tag{6}$$

where C_4 is the capacitance to ground associated with node 4.

Given the following initial conditions,

L_1	$(1,w)$	V_4	0.0	V_h	2.5
L_2	$(0,s)$	L_4	$(0,s)$	V_l	2.5
V_2	0.0	L_5	$(1,w)$	*queue*	[ ]
V_3	5.0	I_{DS}	0.0		

the activity of the simulator will be followed responding to a change in the value of L_1 from $(1,w)$ to $(0,s)$ at time t_0. Note for this example node 4 never has logic level "x", since $V_h = V_l$.

As a result of the change in the input level of the first inverter, it is evaluated and determined to change output to state $(1,w)$ at time t_1. An event to implement the output

change is therefore scheduled in the event queue:

$$queue = [\,(t_1, \text{inv } 1, (1,w))\,]$$

Since this event is the only event in the queue, it is then removed from the queue and simulated time is updated to t_1. The output of the component is changed and the discrete side of node 2 is then evaluated and determined to be $(1,w)$.

Searching the connectivity of the circuit for those components that are in the fanout list of node 2, the discrete to continuous pseudo-component is the only component found. The pseudo-component is evaluated, causing a new voltage source and conductance value to be connected to the output, and the continuous side of node 2 is scheduled for evaluation:

$$queue = [\,(t_1, \text{node } 2, k{=}0)\,]$$

This event is removed from the queue, one iteration (V^0_{2,t_1}) of the solution for the algebraic difference equation $g_2(v) = 0$ is approximated by a single Newton-Raphson iteration, and the second iteration, $(t_1$, node 2, k=1), is scheduled for evaluation. Since node 4 is in the fanout list of node 2, it is also scheduled for evaluation:

$$queue = [\,(t_1, \text{node } 4, k{=}0), (t_1, \text{node } 2, k{=}1)\,]$$

The first event is removed (ordered now by k), V^0_{4,t_1} is approximated for $g_4(v) = 0$, and the node is rescheduled:

$$queue = [\,(t_1, \text{node } 2, k{=}1), (t_1, \text{node } 4, k{=}1)\,]$$

The first event is removed, V^1_{2,t_1} is approximated for $g_2(v) = 0$, and the node is rescheduled:

$$queue = [\,(t_1, \text{node } 4, k{=}1), (t_1, \text{node } 2, k{=}2)\,]$$

This continues until the two nodes converge to some pair of values, V_{2,t_1} and V_{4,t_1}. Upon convergence, they are compared with V_{2,t_0} and V_{4,t_0}, respectively. If there is a sufficient difference between the two values (i.e., the voltage is not stable over that time range), they are scheduled for evaluation at some time $t_2 > t_1$, determined by estimates of the stability and local truncation error of the integration method:

$$queue = [\ (t_2, \text{node } 2, k{=}0), (t_2, \text{node } 4, k{=}0)\ ]$$

The operations in the paragraph above are repeated until V_4 reaches V_h, say at time t_a. Since the logic level associated with V_4 is now a "1" rather than a "0", the output of the continuous to discrete pseudo-component is updated to reflect the new state, $(1,w)$, and the discrete side of node 4 is then evaluated and determined to be $(1,w)$. The fanout list of node 4 is examined and the second inverter is evaluated for potential output changes. As a result of the evaluation, an event is scheduled at time t_b to change the output of the second inverter to $(0,s)$. Assuming V_2 and V_4 are still not stable, nodes 2 and 4 are scheduled for evaluation at time t_{a+1}:

$$queue = [\ (t_{a+1}, \text{node } 2, k{=}0), (t_{a+1}, \text{node } 4, k{=}0), (t_b, \text{inv } 2, (0,s))\ ]$$

It is now useful to examine the event queue in more detail. There are currently 3 events in the queue, node evaluation events for nodes 2 and 4, and an output modification event for the output of the second inverter. Here, t_b has been assumed to be greater than t_{a+1}, but this is not necessarily true in general. Assuming the node evaluation events do have the smallest time value, they are removed from the event queue, V_2 and V_4 are reevaluated, and nodes 2 and 4 are potentially scheduled again for evaluation.

This procedure repeats until V_2 and V_4 have reached a stable value (i.e., they are no longer changing with time). At that point, node 2 and node 4 evaluations are no longer scheduled in the queue:

$$queue = [\ (t_b, \text{inv } 2, (0,s))\ ]$$

The remaining event is removed from the queue, simulated time is updated to t_b, the component output is updated, node 5 is evaluated and determined to be $(0,s)$, the fanout of node 5 is searched and determined to be empty, and no additional events are scheduled.

The event queue is now empty and the simulation terminates. The final value of the variables of interest are

L_1	$(0,s)$	V_4	$5.0-V_{th}$
L_2	$(1,w)$	L_4	$(1,w)$
V_2	5.0	L_5	$(0,s)$
V_3	5.0	I_{DS}	0.0

where V_{th} is the gate to source threshold voltage for the pass transistor. The simulation time is t_b, indicating that the delay through the circuit is $t_b - t_0$.

7. Multiprocessor Implementation Issues

Up to this point the mixed-mode simulation algorithm has been described as if it were running on a traditional von Neumann architecture. This section addresses selected issues involved in implementing the algorithm on multiprocessor-based architectures.

There are two classes of target architectures that can be distinguished here. The first class consists of special purpose simulation engines that have been designed specifically to perform mixed-mode simulation. A nomenclature for distinguishing between the specific architecture styles in this class has been previously presented [8]. This nomenclature is intended specifically for event-based simulation, and we have shown that mixed-mode simulation fits this paradigm. Recent work [29] presents some analytical results giving performance estimates for one type of special purpose architecture running logic simulation, and these results can be extended to analyze the performance of mixed-mode simulation as well.

The second class of target architecture consists of general purpose MIMD multiprocessors of the sort commercially available. Since a long term goal is the development of a "design automation" machine, the flexibility and generality of these machines is considered important. The remainder of this section will focus on the issues of algorithm implementation on these general purpose machines.

Partitioning the simulation workload among a set of interconnected homogeneous processors can be done in two ways.

1. Across the simulation algorithm: Different subtasks of the algorithm (e.g., event queue management, functional evaluation, net-list searches) can be allocated to different processors and the data can be pipelined between the processors.

2. Across the simulated circuit: The data itself (i.e., circuit components) can be partitioned among the processors and each processor handles each of the subtasks of the algorithm for the data in its partition.

It should be clear that the number of subtasks in the algorithm (and therefore the maximum parallelism that can be exploited in the first method) is limited. The data, on the other hand, is typically quite large (e.g., 500,000 transistor chips are now possible), indicating that the potential parallelism that can be exploited with the second method is substantial.

If the second partitioning method is chosen, it is then necessary to devise an algorithm for performing the partitioning. The properties inherent in a good partitioning include a balanced workload amongst the processors as well as a balance between the computational workload performed by the processors and the communications workload for data moving between processors. Both static and dynamic partitioning schemes are possible and various heuristic-based algorithms to perform this partitioning are currently under investigation.

Another unresolved issue concerns the best method for time synchronization between processors. One option is to use a single "global clock" that is common among all the processors. Once all the processing has been completed for the current time, a single master controller sends a message to each processor indicating the next point in time to be simulated. When each individual processor has completed its tasks at this new time, it sends a message to the master saying it is done and when it has more work to do (i.e., the time of the first event in its local event queue). The master can then repeat the above cycle. This option potentially has long periods where the overall utilization is low due to idle processors waiting for work. This is especially true if a non-ideal partitioning algorithm has been used or if the communications network does not efficiently handle broadcast messages.

A second time synchronization option is to allow each processor to maintain its own local time. Different mechanisms for implementing this "distributed clock" have been presented by

Misra [24] and Jefferson [30]. Misra's mechanism is based on a simple algorithm, one susceptible to deadlock, that is enhanced with time-advance messages to perform deadlock detection and correction. Open questions include the minimum number of messages required to guarantee the algorithm to be deadlock free and whether the processor/communications workload can be balanced. Jefferson's virtual time mechanism allows a local processor to arbitrarily work ahead in simulated time without waiting for input from other processors. If a subsequent message from another processor invalidates some of the work already performed, simulated time is rolled back to the time stamp on the incoming message and the simulation proceeds. Open questions here include the overhead involved with a time rollback (i.e., processor time and memory requirements, communication requirements for transmitting annihilator messages to invalidate messages previously transmitted).

8. Conclusions

This paper has presented an investigation of algorithms for both logic and circuit simulation. Both their similarities and differences have been considered, with the goal of presenting a general framework for integrating the two algorithms in a uniform manner. The resulting integrated algorithm is capable of performing mixed-mode simulation, where the circuit has been partitioned into discrete and continuous regions, and each region is simulated at the appropriate level. In addition, several of the issues relating to implementation of mixed-mode simulation on multiprocessors have been presented.

Further work is needed to address the issues outlined in this paper. However, it is clear that multiprocessor-based mixed-mode simulation can be an effective tool in the overall design verification task.

References

[1] Gregory F. Pfister, "The Yorktown Simulation Engine: Introduction," *Proceedings of the 19th Design Automation Conference*, June 1982, pp. 51-54.

[2] Nobuhiko Koike and Kenji Ohmori, "MAN-YO: A Special Purpose Parallel Machine for Logic Design Automation," *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, Illinois, August 1985, pp. 583-590.

[3] R. L. Barto, "Architecture for a Hardware Simulator," *Proceedings of the IEEE Conference on Circuits and Computers*, October 1980, pp. 891-893.

[4] M. Abramovici, Y. H. Levendel, and P. R. Menon, "A Logic Simulation Machine," *IEEE Transactions on Computer Aided Design*, Vol. CAD-2, No. 2, April 1983, pp. 82-94.

[5] J. K. Howard, R. L. Malm, and L. M. Warren, "Introduction to the IBM Los Gatos Simulation Machine Hardware," *Proceedings of the IEEE International Conference on Computer Design*, October 1983, pp. 580-583.

[6] M. E. Glazier and A. P. Ambler, "ULTIMATE: A Hardware Logic Simulation Engine," *Proceedings of the 21st Design Automation Conference*, June 1984, pp. 336-342.

[7] Winfried Hahn and Kristian Fischer, "MuSiC: An Event-Flow Computer For Fast Simulation of Digital Systems," *Proceedings of the 22nd Design Automation Conference*, Las Vegas, Nevada, June 1985, pp. 338-344.

[8] Mark A. Franklin, Donald F. Wann, and Kenneth F. Wong, "Parallel Machines and Algorithms for Discrete-Event Simulation," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp. 449-458.

[9] *The ZYCAD Logic Evaluator: Product Description*, 1983, Zycad Corporation, North Roseville, Minnesota.

[10] William J. Dally and Randal E. Bryant, "A Hardware Architecture for Switch-Level Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-4, No. 3, July 1985, pp. 239-250.

[11] E. H. Frank, "Exploiting Parallelism in a Switch-Level Simulation Machine," *Proceedings of the 23rd Design Automation Conference*, Las Vegas, Nevada, July 1986, pp. 20-26.

[12] David M. Lewis, "A Hardware Engine for Analogue Mode Simulation of MOS Digital Circuits," *Proceedings of the 22nd Design Automation Conference*, Las Vegas, Nevada, June 1985, pp. 345-351.

[13] Ran Ginasor and Neil G. Jacobson, "The Simulation Machine: A VLSI Architecture for Circuit Simulation," EE Pub. No. 540, Dept. of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel, September 1985.

[14] J. T. Deutsch and A. R. Newton, "Msplice: A Multiprocessor-Based Circuit Simulator," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp. 207-214.

[15] H. Shichman and D. A. Hodges, " Modeling and Simulation of Insulated-Gate Field-Effect Transistor Switching Circuits ," *IEEE Journal of Solid-State Circuits*, Vol. SC-3, No. 3, 1968.

[16] Laurence W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL Memo. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, California, May 1975.

[17] Richard M. Chamberlain, "An Algorithm for LU Factorization with Partial Pivoting on the Hypercube," *Proceedings of the 2nd Conference on Hypercube Multiprocessors*,

Knoxville, Tennessee, September 1986, pp. 24.

[18] Thomas F. Coleman and Guangye Li, "Solving Ax = b on a Hypercube," *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, Knoxville, Tennessee, September 1986, pp. 29.

[19] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation Based Electrical Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-3, No. 4, October 1984, pp. 308-331.

[20] P. Odryna and S. R. Nassif, "The ADEPT Timing Simulation Algorithm," *VLSI System Design*, March 1986.

[21] L. M. Vidigal, S. R. Nassif, and S. W. Director, "CINNAMON: Coupled Integration and Nodal Analysis of Mos Networks," *Proceedings of the 23rd Design Automation Conference*, Las Vegas, Nevada, July 1986, pp. 179-185.

[22] Masato Kawai and John P. Hayes, "An Experimental MOS Fault Simulation Program CSASIM," *Proceedings of the 21st Design Automation Conference*, June 1984, pp. 2-9.

[23] Randal E. Bryant, "An Algorithm for MOS Logic Simulation," *Lambda*, Vol. 1, No. 3, Fourth Quarter 1980, pp. 46-53.

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

[25] John P. Hayes, "Digital Simulation with Multiple Logic Values," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-5, No. 2, April 1986, pp. 274-283.

[26] Roger D. Chamberlain, "Lsim: A Gate-Switch Level Logic Simulator," Master of Science Thesis, Department of Computer Science, Washington University, St. Louis, Missouri, May 1985.

[27] J. E. Kleckner, "Advanced Mixed-Mode Simulation Techniques," Memorandum No. UCB/ERL M84/48, Electronics Research Laboratory, University of California, Berkeley, California, June 1984.

[28] G. Arnout and H. DeMan, "The Use of Threshold Functions and Boolean-Controlled Network Elements for Macromodelling of LSI Circuits," *IEEE Journal of Solid-State Circuits*, Vol. SC-13, June 1978, pp. 326-332.

[29] Kenneth F. Wong and Mark A. Franklin, "Performance Analysis and Design of a Logic Simulation Machine," Technical Report WUCS-86-19, Department of Computer Science, Washington University, St. Louis, Missouri, November 1986.

[30] David Jefferson, "Virtual Time," *Proceedings of the 1983 International Conference on Parallel Processing*, 1983, pp. 384-393.