A NOMINAL/INERTIAL DELAY METAMORPHIC DIFFERENTIAL SIMULATOR

Peter M. Maurer

Dept. of Computer Science
Baylor University
Waco, TX, 76798
Peter_Maurer@Baylor.edu

ABSTRACT

Metamorphic differential simulation is an effective method of simulating digital circuits that has proven to give significant performance improvements over conventional simulation. Initially it was restricted to the zero-delay timing model, but recent work has extended it to the unit-delay, multi-delay, and nominal-delay models. Although the nominal-delay simulator gave significant improvements over conventional simulation, it was confined to the transport model of gate delays. This model is less realistic than the inertial model of delays, and can lower performance due to the enormous number of events produced. It was not obvious that extension to the inertial model would be possible because it requires a type of event cancellation that is quite different from that used in the previous simulators. We show that such an extension is not only possible, but that it produces an effective simulator that gives significant performance improvement over conventional simulation.

Keywords: Logic Simulation, Nominal Delay, Inertial Delay, Metamorphic Simulation.

1 INTRODUCTION

Simulation is one of the most important tools in the design of integrated circuits, because it permits the design to be verified as it is being created without the expense of actually manufacturing the circuit. Event-driven simulation is an important type of simulation that combines simulation accuracy with high performance (Ulrich 1978). The main mechanism for improving performance is avoiding unnecessary work by simulating only those gates whose inputs change value. The primary difference between different types of event driven simulation is the timing model used to simulate the delay of a gate. The zero-delay model treats circuit elements as pure functions with no internal delay (Wang and Maurer 1990, Maurer 2000). There are some circuits, such as RS-flip-flops, implemented at the gate level, that cannot be simulated accurately with the zero-delay model, because the behavior of the circuit depends on the internal delay of the component gates. The various types of delay models remove this shortcoming. The simplest model is the unit-delay model which simulates the circuit in terms of gate delays which are all identical (Heydemann and Dure 1988; Lewis 1989; Lewis 1991; Maurer 2012). The multi-delay model is more accurate in that it assigns differing delays to each gate (Szygenda, Rouse and Thompson 1970; Maurer 2016). However the delays are specified as integers without units, thus giving a simulation with respect to the relative delays of different gates.

Nominal delay gives even better accuracy by using floating point delays that represent actual time units. These delays are normally extracted from the layout of the circuit, making nominal delay simulation a tool for the later stages of circuit development.

One problem with nominal delay simulations is determine what happens when two events for the same net occur very close to one another. In particular, it is important to consider two complementary events that occur in an interval that is less than the delay of the gate driving the net. (This problem also occurs in multi-delay simulation.) There are two methods of handling this situation. The first is the transport delay model, which processes all events regardless of their proximity. This causes gates to transmit short pulses even though in the actual circuit these pulses would be damped by the delay of the gate. The second method is called inertial delay, which eliminates short pulses by causing two complementary events to cancel one another if they occur within the delay of the driving gate.

Neither model is 100% accurate. The transport delay model will cause simulations of down-stream gates, even though these gates may never change state in the actual circuit. The inertial delay model can mask problems that may occur when the length of the suppressed pulse is close to the gate delay. Under these circumstances, certain down-stream gates may actually change state in the physical circuit while others do not. For this reason, we prefer the transport delay model for multi-delay simulations. However, in the nominal delay model another problem occurs when using transport delay. For many circuits, the transport delay model produces a prodigious number of events. Tens of millions of events can be queued simultaneously for circuits containing only a few thousand gates. This complicates queue management and slows simulation because eventually each of these events must be processed. Although we have created a highly efficient nominal delay simulator using the transport delay model (Maurer 2018), we feel that the inertial delay is more realistic for nominal delay simulation. This paper describes our inertial delay simulator.

We have created metamorphic differential simulators for a number of timing models (Maurer 1994; Maurer 2000; Maurer 2012; Maurer 2016; Maurer 2018), and the results have uniformly provided substantial increases in performance over traditional techniques. The differential model treats both gates and nets as state machines, with state changes being stimulated by other state changes in the circuit. The metamorphic approach (Maurer 2000; Maurer 2012; Maurer 2016) is an especially useful tool for improving the performance of differential simulation. In the metamorphic approach, state codes are replaced with processing routine addresses. These addresses change to reflect the current state of the machine. The addresses point to the handling routines for the current state, eliminating state decoding and permitting immediate jumps to the appropriate processing routines.

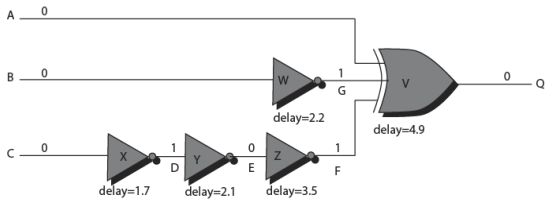
In this paper we will show that metamorphic differential simulation can also substantially improve the performance of the inertial delay model.

2 THE NOMINAL DELAY SORTING MECHANISM

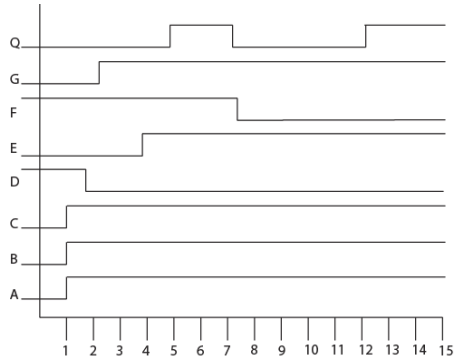
The term “nominal-delay” refers to the average behavior of a gate. In actual circuits, the delay of a gate can differ due to many different factors. In nominal-delay simulation, the average delay of the gate is calculated through circuit analysis and used for all state-changes in the gate. Delays are usually floating-point numbers with specific time units, and simulation is usually event driven with events representing a change in the value of a net. Events can be produced out of order but must be processed in ascending order by time. This implies that a sorting mechanism must be used to order the events for processing. In multi-delay simulators a timing wheel, which is a bucket-sorting mechanism, is used for this purpose. This gives an $O(n)$ sort for times specified as integers.

In nominal-delay simulation delays and times are normally specified as floating point numbers. Although these could be normalized into integers, the resultant delays and times would be very large and sparse. This would cause the simulator to spend an excessive amount of time searching empty queues for the next event. A more natural choice for nominal delay is the priority queue. The priority queue is a heap-sort mechanism that requires $O(n \lg n)$ time to sort n events. But if the average number of events in the queue is much smaller than the total number of events produced, n will be much smaller than the total number of events produced, increasing the efficiency of the sort.

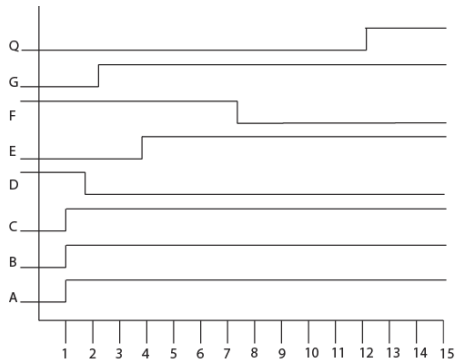
If the transport delay model is used, there can be many events for the same net in the queue simultaneously. This situation occurs when the time interval between events is shorter than the delay of the gate. In the inertial delay model, events are cancelled when the time between them is shorter than the gate-delay. This increases the efficiency of the sorting mechanism because the average number of events in the queue will be smaller than for transport delay. For some circuits such as c6288 in the ISCAS85 benchmarks (Berglez, Pownall, and Hum 1985), the transport delay model can result in millions of events in the queue for the same net. In this case the inertial delay model is clearly superior to the transport delay model. Figure 1 shows the difference between the two models.



a. Circuit



b. Transport Delay



c. Inertial Delay

Figure 1: Delay Models.

In the inertial delay model, no more than one event can be present in the queue for any net. In a conventional simulator, events can be either complementary or reinforcing, and the cancellation mechanism must take this into account to determine the correct action. In a metamorphic differential simulator, any two consecutive events for a net are complementary simplifying the process of cancellation. If an event occurs for a net, and there is already an event in the queue for the net, both events must be cancelled. In Figure 1, note the difference in the wave-form for net Q.

3 THE PRIORITY QUEUE

A priority queue is a nearly-complete binary tree, with all levels, except the lowest, complete. The leaf vertices on the lowest level are clustered as far to the left as possible. If it is necessary for a non-leaf on the second-lowest level to have a single child, this child must be a left child. The vertices of the tree contain event times and other event data. The earliest event is in the root, and for any non-leaf the events in the children are either simultaneous or later than the event in the non-leaf. This relationship is maintained when events are added or deleted.

A new event is added by creating a new vertex in the leftmost unused position on the lowest level or by creating a new level and adding the event in the leftmost position. Event data is copied into the new vertex and the required parent/child relationship is maintained by swapping the event with its parent until the relationship is correct. The new event may be pushed up to the root if it is early enough.

During normal simulation, events are removed from the priority queue at the root. When the root event is processed, the rightmost vertex on the lowest level is removed and its data is copied into the root vertex. The root vertex is pushed down by exchanging it with its earliest child until the parent/child relationship is correct or until a leaf vertex is reached. Because the height of a complete, or nearly complete binary tree with n vertices is $\lg n$, adding and deleting vertices is an $O(\lg n)$ process.

The binary tree is implemented as an ordinary array, with vertices indexed as shown in Figure 2. The left child of vertex n has index $2n$ and the right child has index $2n+1$. The parent of vertex n has index $\lfloor n/2 \rfloor$. Element zero of the array is unused. The root is contained in element 1. If the tree has n vertices, the rightmost element on the lowest level is contained in element n. Leaves have indices that are greater than $\lfloor n/2 \rfloor$. Indices 1 through $\lfloor n/2 \rfloor$ contain non-leaves.

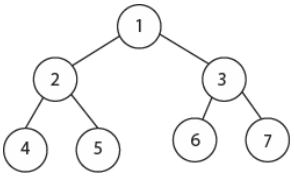


Figure 2: A Priority Queue.

In the inertial delay model, it is necessary to cancel events before they reach the apex of the priority queue. There are two degenerate cases one where the cancelled event is at the apex of the priority queue and one where the cancelled event is the event with the highest index in the queue. In the first case, the removal of the event is handled as described above, but without processing the event. In the second case the size of the queue is reduced by one without further processing. In all other cases, the event with the highest index replaces the cancelled event, and the size of the queue is reduced by one. If the replacement has a time that is earlier than its parent, it is moved up in the queue by successively exchanging it with its parent until the parent has an earlier time than the replacement, or until the replacement reaches the apex of the queue. If the replacement event has a time that is greater than or equal to its parent, it is moved down in the queue, by exchanging it with its earliest child, until both of its children have times later than the replacement, or until the replacement reaches a leaf. The new event, which would normally be added to the queue, is discarded.

When cancelling an event, it is necessary to know its location in the queue. This requires a back-pointer to the current location of the event in the queue. The back-pointer is kept with the data structures corresponding to the net so that given a net, it is possible to determine the location of its event in the queue.

Since only one event for a net can be queued at any given time, a static data structure can be used for this purpose.

4 THE SIMULATION PROCESS

Our simulator first compiles a circuit executable code that is then run to produce the simulation, allowing many simulations to be performed with a single compilation. Our simulator uses untimed vectors, meaning that the simulation of each vector starts at time zero. It could be easily modified to start the simulation of each vector at a specified time to detect timing errors between vectors.

Vectors are read one at a time and compared with the previous input vector to detect changes in the inputs. Because we use differential simulation, the circuit must be initialized to a consistent state which will be used to detect state changes when the first vector is read. During the compilation phase a zero-delay simulation is performed using an input vector of all zeros. Memory devices, if any, are initialized to zero. When input changes are detected, the corresponding events are inserted into the queue at time zero.

Simulation has two phases, the event phase during which events are processed, and the gate phase during which gates are simulated. Simulation begins with the event phase. The two phases alternate with one another until the priority queue becomes empty. Then simulation proceeds with the next vector. Output vectors, with time values, can be produced when the priority queue becomes empty or when the event phase completes, or both.

Each phase consists of a collection of data structures that is processed serially. The processing routines are embedded in the data structures themselves. Each data structure contains a pointer to its current handling routine. A data structure is processed by inserting a pointer to the structure into a global variable and calling a processing routine. The final step in each processing routine is to advance to the next data structure. Processing routines routinely replace their own addresses in the data structure to keep track of state changes. States are not explicitly recorded, because the processing routine address serves the same purpose.

When the event phase begins, the current time is set to the time of the first event in the queue. During the event phase, an event or collection of events is removed from the priority queue and processed. The event phase terminates when the event queue becomes empty or when the next event to be processed has a time later than the current time.

Each event in the priority queue contains a pointer to a linked list of fanout structures. There is one fanout structure for each fanout branch of the net and at least one other structure which is used to terminate the list. If the net is a primary output, an additional structure is added to compute the output value. (Intermediate net values are not required in differential simulation.) Each event structure contains a pointer to a processing routine and a pointer to a gate structure. The last data structure in the list terminates the processing of the current net and proceeds with processing the next event that is simultaneous with the current event. If no such event is present, the event phase ends and the gate phase begins.

A gate is placed in the gate queue when the event phase determines that the output of the gate will change value. Because gate-inputs are processed serially, a complementary change that occurs at the same time as a previously detected change will be cancelled by removing the gate from the gate queue.

During the gate phase of the simulation, all gates are removed from the gate queue and processed. The gate structure contains the delay of the gate and a linked list of fanout structures for the gate output. The gate delay is added to the current time, and an event is queued for that time.

5 EVENT AND GATE STRUCTURES

Each event structure can have two states which are associated with the one/zero values of the net, but the correspondence is not fixed. A particular state might represent either a one or a zero, depending on the gate

at the end of the corresponding fanout branch. Fanout branches that lead to AND, NAND, OR and NOR gates have two states, while fanout branches leading to NOT, BUFFER, XOR and XNOR gates have only one. When a net changes value, it either changes from one to zero or from zero to one. The two types of changes alternate with one another, so if a one-to-zero event is executed, the next event must be a zero-to-one event.

Figure 3 shows the priority queue element. There is one such element for each net. When queuing an event, a pointer to this structure is inserted into the priority queue. The time at which the event will occur is contained in the *Time* element. The *Insert* pointer points to either the *Queue* or *Dequeue* routine for the event. If there is currently no event in the queue for the net, the Insert element points to *Queue*, which will insert the event into the queue. The *Queue* routine replaces its address with a pointer to the *Dequeue* routine. The *Dequeue* routine will remove the event from the queue if there is an attempt to add a second event to the queue. The *Dequeue* routine replaces its own address with the address of the *Queue* routine. Simultaneous events for the same net are eliminated by gate queue processing, so multiple events for the same net must be consecutive and complementary with their immediate predecessors.

```
struct ChHdr
{
        double Time;
        void *Insert;
        int BP;
        Event *Data;
};
```

Figure 3: The Priority Queue Element.

The *BP* element points to the position of the event in the queue. It is used during event cancellation to find the existing event in the queue. The *Data* element points to a linked list of fanout structures for the net. The final fanout structure in this list will reset the *Insert* element so that it points to the *Queue* routine.

The fanout-structure is shown in Figure 4. The Rtn pointer points to the current handling routine. This pointer maintains the state of the structure. For structures with two states each processing routine will replace the Rtn pointer with a pointer to its opposite number. The *G* pointer points to the gate structure for the fanout branch. For structures used to compute the value of primary outputs, this pointer is overloaded to point to the variable holding the current ASCII value of the primary output.

```
struct Event
{
        Event *Next;
        void *Rtn;
        Gate *G;
}
```

Figure 4: The Fanout Data Structure.

The gate structure is shown in Figure 5. These structures are placed in the gate queue, when a change is detected in the output of a gate. If more than one simultaneous event is detected in the gate output, these events are complementary and the gate is removed from the queue. The gate structures are doubly linked, to facilitate easy removal from the gate queue. The *Begin* pointer points to the head of the fanout chain for the gate output.

The state of the gate is represented by the *Up* and *Down* pointers. These point to routines that are extensions of the event processing routine. The state of the gate is determined by how many of its inputs have the dominant value (zero for AND and NAND, one for OR and NOR). These routines perform the state changes for the gate, and queue the gate structure when its output changes value. These pointers are unused for

NOT, BUFFER, XOR and XNOR gates. The *Rtn* pointer of the gate structure points to the processing routine that is used for all gates. This pointer is not used to maintain a state, but facilitates the use of a trailer routine for the gate queue.

```
struct Gate
{
        Gate *Next;
        Gate *Prev;
        void *Rtn;
        double Delay;
        Event *Begin;
        void * Up;
        void * Down;
        void * Schedule;
}
```

Figure 5: The Gate Data Structure.

The *Schedule* pointer handles the queueing and dequeueing of the gate structure. If the gate structure is not currently on the gate queue, then the *Schedule* pointer points to a routine that places the structure onto the queue. If the gate structure is already on the gate queue, then the *Schedule* pointer points to a routine that removes it from the queue.

The *Delay* element contains the nominal delay of the gate.

6 DIFFERENTIAL SIMULATION

In differential simulation, each gate maintains a state that is used to determine whether the gate will propagate an event. Events on the inputs of the gate trigger state changes that can, in turn, trigger output events. There is no need to recompute the value of the output for each input event, and the operations used to compute new states are all binary operations, as opposed to the *n*-ary operations needed to recompute the gate output value. Intermediate net values are not required, with only the values of primary inputs and primary outputs being explicitly computed. This permits much coding to be omitted.

In our simulator, each AND, NAND, OR and NOR gate has an internal state which is established by the initial zero-delay simulation performed during compilation. (The Up and Down pointers of the gate structure given above maintain this state.) The state is determined by a count of the number of inputs that have dominant values. If a fanout branch leads to an AND, NAND, OR, or NOR gate, its state is determined by whether the next event for the net will cause the dominant count to increase or decrease. If an event causes the dominant count to increase, the next event on the same net will cause the dominant count to decrease.

When an event causes the dominant count of a gate to increase from zero, or decrease to zero a queueing action is performed. A queueing action always occurs for NOT, BUFFER, XOR or XNOR gates, since every change in an input triggers a change in the output.

Event processing continues until the next event to be processed has a time that is later than the current time. When this occurs, all gates on the gate queue are removed, their states are reset to indicate that they are no longer on the gate queue, and an events for the gate outputs are inserted into the priority queue. The time for these events is the current time plus the delay of the gate.

7 METAMORPHIC SIMULATION

Metamorphic programming is similar to polymorphic programming in which each object has a pointer to a table of virtual functions that perform specialized operations for the object. In metamorphic programming

the table of functions still exists, but the set of virtual functions is permitted to change as program execution proceeds. Metamorphic programming also provides metamorphic subroutines that execute in the stack frame of the caller instead of creating a new stack frame for each call. The main objective is to avoid state codes and state analysis wherever possible by using specialized handling routines for each state. Subroutine pointers replace state codes. No state analysis is necessary because the handler for the current state can be called directly from the pointer. When states change, subroutine pointers are changed accordingly.

A metamorphic subroutine essentially replaces the body of the existing subroutine when it is called. No new stack frame is created. Instead the metamorphic subroutine executes in the environment of its caller. In practice, this is done by using highly structured routines and computed gotos, but there is no reason that a few simple compiler enhancements could not provide these features directly.

In our simulator, fanout and gate structures are metamorphic objects. Fanout structures for fanouts that lead to AND, NAND, OR and NOR have two handler routines, EVUP, and EVDN, which increment and decrement the gate state. These handlers alternate with one another during simulation. Each replaces its own address in the fanout structure with a pointer to its opposite number. Figure 6 shows these two routines along with the NOT routine that is used to handle NOT, BUFFER, XOR and XNOR gates. This code uses the gcc extensions for computed gotos. When the statement "goto * X" is executed, X must be a void pointer containing the address of a label in the current function. Label addresses are obtained using the "&&" operator.

```
EVUP:
Ev->Rtn = &&EVDN;
Gt = Ev->G;
goto * Gt->Up;

EVDN:
Ev->Rtn = &&EVUP;
Gt = Ev->Gate;
goto * Gt->Down;

NOT:
Gt = Ev->Gate;
goto * Gt->Schedule;
```

Figure 6: Fanout State Handlers

In Figure 6, *Ev* is a global variable pointing to the Event structure for the current fanout branch, and *Gt* is a global variable pointing to the current gate structure. Note the handling of the EVUP and EVDN routines. The *Up* and *Down* pointers of the gate structure are extensions of the event handlers that maintain the dominant count of the gate. The *Schedule* pointer of the gate structure is used to queue (or dequeue) gate structures in the gate queue.

The dominant count does not actually exist. There is an Up/Down pair of routines for each value of the dominant count. These routines are generated at compile time to guarantee that there are a sufficient number of them to handle all types of gates in the circuit. Figure 7 shows some sample Up/Down pairs. Note how each replaces the pointers to the existing pair with the pair to handle the new state. Also note that the scheduling routine is called only by UP0 and DN1, because these represent the dominant count being incremented from zero or decremented to zero. The other routines continue with the next event.

The Schedule pointer of the gate structure points to one of the two routines GATEQUEUE and GATEDEQUEUE. These alternate with one another by replacing the contents of the Schedule pointer with a pointer to the opposite routine. These routines are conventional doubly-linked-list queueing routines. At the end of each, processing proceeds with the next fanout branch in the event chain.

Gate processing processes every gate in the gate queue by calling Rtn pointer of every structure in the gate queue. This routine queues the output of the gate or cancels existing events when necessary.

UP0: Gt->Up = &&UP1; Gt->Down = &&DN1; goto * Gt->Schedule;	DN0: shp = Ev->Next; goto * Ev->Rtn;
UP1: Gt->Up = &&UP2; Gt->Down = &&DN2; Ev = Ev->Next goto * Ev->Rtn;	DN1: Gt->Up = &&UP0; Gt->Down = &&DN0; goto * Gt->Schedule;
UP2: Gt->Up = &&UP3; Gt->Down = &&DN3; Ev = Ev->Next; goto * Ev->Rtn;	DN2: Gt->Up = &&UP1; Gt->Down = &&DN1; Ev = Ev->Next; goto * Ev->Rtn;

Figure 7. The Up and Down Routines.

8 TRAILER ROUTINES

Trailer data structures are used to avoid the necessity of checking for the end of event chains and empty queues. Three types of data structures are used, one for event-chains, one for the gate-queue and one for the priority queue. These structures have special processing routines as handlers. The priority queue trailer prints an output vector and proceeds with the next input vector. The event-chain trailer goes to the next event if it is simultaneous with the current event. Otherwise it proceeds with gate queue processing. The gate trailer advances the current time to match the time of the next event in the priority queue and begins the processing of the event.

9 EXPERIMENTAL DATA

Several experiments were run to assess the efficiency of our metamorphic differential inertial delay simulator. The ISCAS85 benchmarks (Berglez, Pownall, and Hum 1985) which have become a standard tool for assessing simulation performance were used for this purpose. We added random floating-point gate delays in the range (0,8). Delays were generated in such a way as to minimize the likelihood of any two gates having the same delay. We could also generate delays in a more realistic way, with the gate delay based on the number of inputs, but random delays give a much more stressful test, since each path through the circuit has a different delay. We ran each simulation both on our metamorphic differential simulator and on our conventional inertial-delay simulator. Simulation times were obtained and are reported in Table 1 as seconds of execution times. The time required to read input vectors and print output vectors is not included. Each result is the average of five repetitions of the experiment.

Table 1: Experimental Results

Circuit	New	Conventional	Improvement
c432	5.134	7.046	27%
c499	7.101	9.400	24%
c880	12.562	17.547	28%
c1355	22.446	30.585	27%
c1908	42.978	62.487	31%
c2670	55.435	85.228	35%
c3540	83.487	123.855	33%
c5315	136.813	213.558	36%
c6288	825.890	1174.704	30%
c7552	247.001	383.480	36%

Table 1 shows that metamorphic differential simulation provides a substantial performance improvement over conventional simulation. The simulator used for comparison purposes is one of a set of conventional simulators, developed by us, to compare the simulation speeds of various algorithms. These simulators

allow us to strip out the simulation algorithms from the overhead and concentrate purely on simulation issues.

10 CONCLUSION

This research shows that the metamorphic differential technique is adaptable to inertial-delay simulation, just as it was to zero-delay, unit-delay, multi-delay and nominal/transport delay simulation. Substantial performance gains over conventional simulation have been realized for the inertial delay timing model. As it stands, the metamorphic differential technique is a powerful tool that can be used effectively in virtually all types of digital simulation.

REFERENCES

Brglez, F., P. Pownall, R. Hum. 1985. "Accelerated ATPG and Fault Grading via Testability Analysis," ISCAS, pp. 695-698.

Heydemann, M., D. Dure. 1988. "The Logic Automation Approach to Accurate Gate and Functional Level Simulation," Proceedings of ICCAD, pp. 250-253.

Lewis, D. M. 1989."Hierarchical Compiled Event-Driven Logic Simulation," Proceedings of ICCAD, pp.498-501.

Lewis, D. M. 1991. "A Hierarchical Compiled Code Event-Driven Logic Simulator," IEEE Transactions on Computer Aided Design, Vol 10, No. 6, pp.726-737.

Maurer, P. M. 1994. "The Inversion Algorithm for Digital Simulation," Proceedings of ICCAD, pp. 259-61.

Maurer, P. M. 2000. "Event Driven Simulation Without Loops or Conditionals," ICCAD 2000.

Maurer, P. M. 2012. "Unit Delay Simulation With the EVCF Algorithm," Proceedings of WorldComp 12/MSV 12, http://elrond.informatik.tu-freiberg.de/papers/ WorldComp2012/MSV4422.pdf.

Maurer, P. M. 2016. Metamorphic differential simulation using the multi-delay timing model, Symposium on Theory of Modeling and Simulation, (SpringSim).

Maurer, P. M. 2018. "A Nominal Delay Metamorphic Differential Simulator for Digital Circuits", Proceedings of the 2018 International Conference on Modeling, Simulation & Visualization Methods, pp. 26-32.

Szygenda, S., D. Rouse, E. Thompson. 1970. "A Model and Implementation of a Universal Time-Delay Simulator for Large Digital Nets," Spring Joint Computer Conference, pp. 491-496.

Ulrich, E. G. 1978. "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, "JACM, V.21, N.9, pp. 777-85.

Wang, Z. and P. M. Maurer. 1990. "LECSIM: A Levelized Event Driven Compiled Logic Simulator," Proceedings of the 27th Design Automation Conference, pp. 491-496.

AUTHOR BIOGRAPHIES

PETER M. MAURER is a Professor of Computer Science at Baylor University, Waco TX, USA. He holds a PhD in Computer Science from Iowa State University. His research interests lie in simulation of digital circuits, VLSI design automation, and random generation of simulation data. His email address is Peter_Maurer@Baylor.edu.