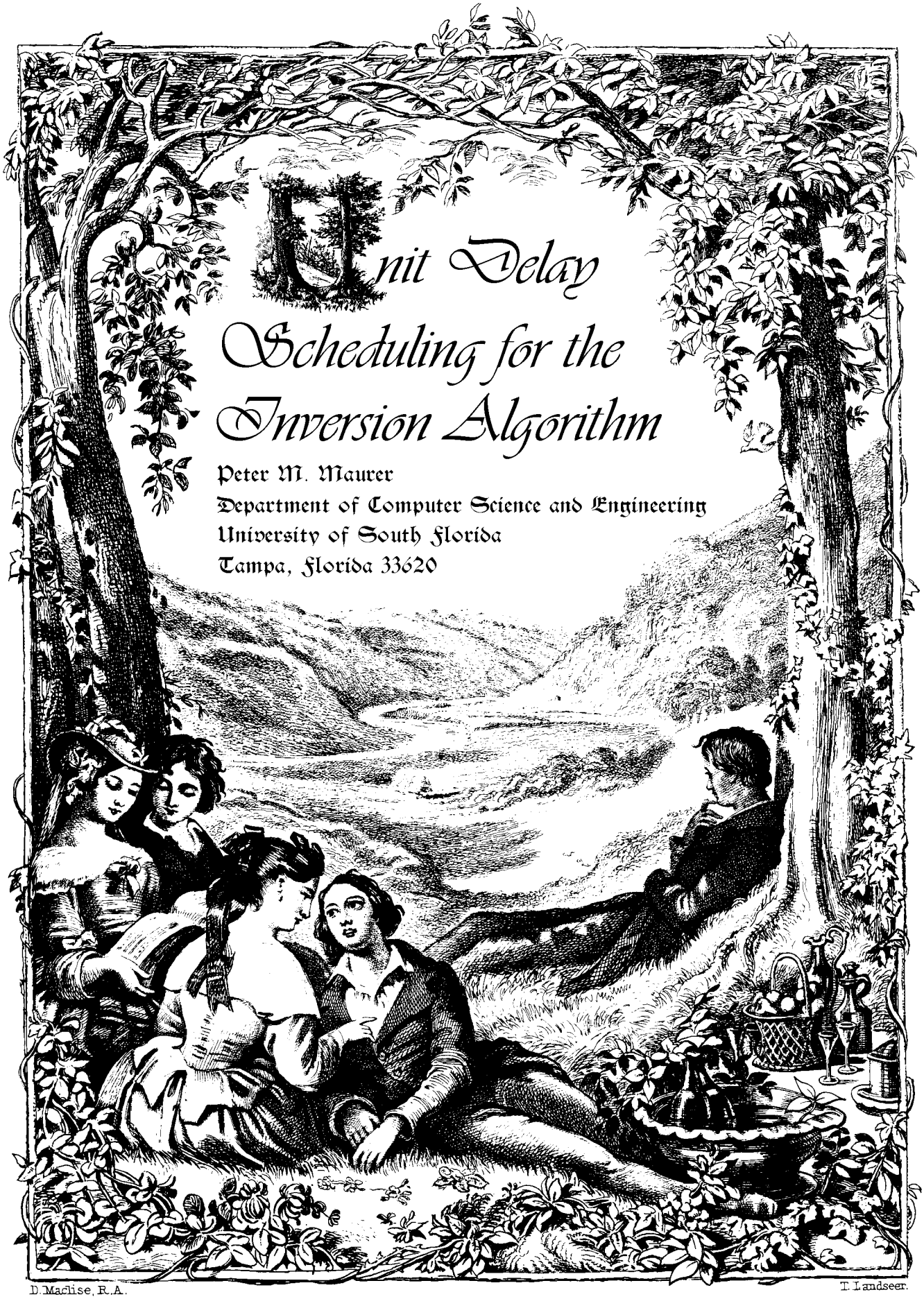


Unit Delay Scheduling for the Inversion Algorithm

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, Florida 33620

D. Maclise, R.A.

T. Landseer.

UNIT DELAY SCHEDULING FOR THE INVERSION ALGORITHM

Peter M. Maurer
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

The Inversion Algorithm is an event driven algorithm whose performance meets or exceeds that of Levelized Compiled Code simulation, even when the activity rate is unrealistically high. Existing implementations of the Inversion Algorithm are based on the Zero Delay model. This paper presents an implementation which is based on the Unit-Delay model. Although the most basic form of the Inversion Algorithm can be converted to Unit Delay with little difficulty, special considerations must be taken to avoid scheduling conflicts. The main problems discussed in this paper are avoiding scheduling conflicts, and minimizing the amount of storage space required to do so. These problems are made considerably more difficult by the deletion of NOT gates and the collapsing of various connections. These optimizations transform the simulation into a multi-delay simulation under the transport delay model. A complete solution to the scheduling problem is presented under these conditions.

Principal Author's Note

A number of fine 19th Century engravings have been added to this manuscript. These engravings, which have only a tenuous relationship to the accompanying text, are strictly for the enjoyment of you, the reviewer. They will not be included in the final draft. The task of reviewing a new manuscript is often dull and usually thankless. I hope that these engravings make your job more enjoyable. Oh, and thank you.

UNIT DELAY SCHEDULING FOR THE INVERSION ALGORITHM

Peter M. Maurer
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

1. Introduction

The Inversion Algorithm[35] is an event-driven logic simulation algorithm that provides significant advantages over existing simulation techniques[1-60]. Although it is event-driven, its performance is comparable to that of Levelized Compiled Code (LCC) simulation[21], even when the activity rate is unrealistically high. At lower activity rates, the performance of the Inversion Algorithm improves, while the performance of LCC simulation remains constant. Furthermore, the amount of run-time code required by the Inversion Algorithm is only a tiny fraction of that required by other simulation algorithms, particularly LCC simulation. The amount of code is small enough to permit assembly language routines to be used at run time without sacrificing code portability. A separate run-time module would be required for each new platform, but such a module would require no more than a few days to create.

Despite its advantages, the current implementations of the algorithm have some drawbacks. First, in all existing implementations, the simulation is two-valued. Because the Inversion Algorithm requires all nets to be initialized to consistent values, it is necessary to simulate at least one input vector at compile time. Because of this, these implementations cannot be used effectively with asynchronous sequential circuits. It may be impossible to fully initialize such circuits using a single input vector, and the correct initialization sequence may be difficult or impossible to discover. The current implementations of the algorithm may be used effectively only with combinational and synchronous circuits. A second drawback is that all of the existing implementations of the Inversion Algorithm are based on the zero-delay timing model. This does not permit one to detect static and dynamic hazards, nor does it permit one to do detailed timing analysis.

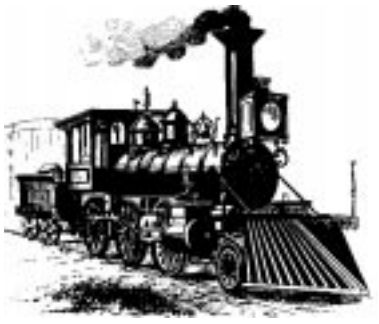
Although the two-valued zero-delay model can be enormously effective in diagnosing and fixing design problems, the Inversion Algorithm must be extended to include multiple logic values and non-zero delays if it is to achieve its full effectiveness as a design tool. The problem of adding the unknown value to the simulation has been addressed in a recent paper[36]. The purpose of the current paper is to extend Inversion Algorithm to the unit-delay timing model. Incorporation of the unit-delay timing model is an important first step in extending the Inversion Algorithm to more complex timing models. The unit-delay model allows one to detect hazards, and thus is more accurate

than the zero-delay model, without incurring the severe performance penalty usually associated with more detailed models. As it turns out, certain optimizations of the Inversion Algorithm[35] will require the simulator to handle delays greater than one, thus many of the problems that appear in more complex timing models must be handled by the unit delay simulator.

2. Fundamental Problems.

The primary obstacle in adapting the Inversion Algorithm to Unit-Delay scheduling, is that the Inversion Algorithm is essentially a *single-list* algorithm. Most event-driven logic simulation algorithms can be categorized as *single-list* or *double-list* algorithms.

In double-list scheduling two queues are used, one for events and one for gates. Any change in the value of a net generates an event, which is represented by an event structure. Event structures can be placed on the event queue during input-vector processing or during gate simulation. During input vector processing, events are generated by comparing the new input values to those of the input previous vector. Any change in an input net generates an event. No events are processed until the input vector has been completely examined. Once all events have been generated, the event handler is called to process the event queue. When an event is processed for a net N, all gates in the fanout list of N are added to the gate queue. Once the event queue has been exhausted, the gate-simulation routine is invoked to process the gate queue. After a gate is simulated, its output nets are examined for changes, and an event is added to the event queue for each changed net. Once the gate queue is exhausted, the simulator invokes the event processor to process any events that might have been queued during gate simulation. Simulation terminates when both queues become empty simultaneously. No new events are added to the event queue during event processing, and no gates are added to the gate queue during gate simulation.



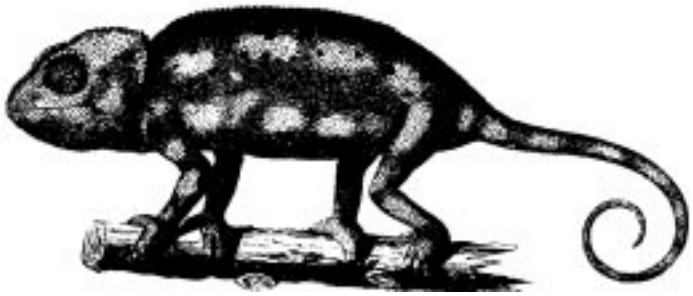
In contrast, single-list simulation uses only an event queue. During the processing of an event for net N, all gates in the fanout of N are simulated, and any new events are immediately inserted into the event queue. When an event for a net N is placed in the queue, it is possible for there to be an unevaluated event for net N already in the queue. Single-list scheduling may evaluate a particular gate several times at one instant of simulated time. In double list scheduling it is possible to guarantee that no more than one event is queued for a particular net at any given time and it is possible to prevent unnecessary simulations. (Despite these drawbacks, single-list scheduling is considered by many to be faster than double-list scheduling.)

Because the Inversion Algorithm performs no gate evaluations (except for monitored nets), there is no gate evaluation step. Thus the Inversion Algorithm *must be* implemented as a single-list algorithm. This causes no difficulties in zero-delay

simulation, because no event can be scheduled more than once. However, when it is possible to schedule two or more events for the same net simultaneously, the static storage management techniques used by the Inversion Algorithm may fail, leading to errors in simulation. In the zero delay version of the algorithm, a data structure is created for each fanout branch of each net. Events are queued by linking the data structures directly to other data structures already in the queue. It is possible to switch to a dynamic storage management technique, but this could impair the performance of the algorithm. The alternative is to retain the static storage mechanisms, but provide extra data structures to handle the simultaneous queuing of events.

3. The Red/Green Method.

The main problem with static storage management is not simultaneous queuing of events, but simultaneous queuing of event *structures*. Therefore it should be possible to avoid duplicate scheduling of event structures by creating more than one structure per fanout branch. Any mechanism to permit simultaneous queuing of several events for a single net, must also support *event collapsing*. (In the Inversion Algorithm, events are queued on a net by net basis, but are processed one fanout branch at a time. See reference [35] for details.) Any time an event is to be queued for a time slot *t*, the algorithm must examine the queue to determine whether there is an event queued for time slot *t*. If so, the new and old events must be combined into a single event. In two-valued simulation this results in both events being dropped.



The number of distinct structures that are required for each net is equal to the maximum number of events that could be simultaneously queued for any net. As Lemma 1 indicates, in unit delay scheduling, no more than two event structures will be required.

> Lemma 1. *In Unit Delay Scheduling with event collapsing and events processed in ascending order by time, there can be no more than two events queued for any net at any time.*
>
> Proof. Due to event collapsing, there can be no more than one event queued for any net in time-slot *t*. Any attempt to queue a second event for the net at time-slot *t* will result in the two events being combined. Since the delay of every gate is at most 1, when event is processed during time-slot *t*, the latest time-slot for which events can be queued is time-slot $t+1$. Since events are processed in ascending order by time, there can be no events queued for any time-slot $s<t$. Thus, when an event E is processed during time-slot *t*, there can be events queued for time slot *t* and for time slot $t+1$, but there can be no events queued for any other time slot.

Since no net can have more than two queued events at any time, we have adopted an odd/even strategy for event queuing. In this strategy, each fanout branch is assigned two event structures known as the Red and Green event structures. As in the zero delay algorithm, each of these structures contains the operating data required to process an event. However, the structures are queued in alternating fashion. When a green structure is processed, any propagated events will be queued using the red data structures, and vice versa. Input vector processing schedules only the red event structures. This strict alternation in structures, effectively, assigns colors to time-slots, as illustrated in Figure 1.

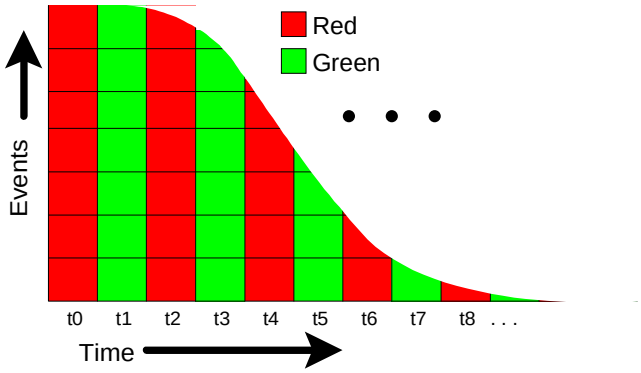


Figure 1. The Odd/Even Strategy.

4. Eliminating Useless Event Structures.

The main advantage of the Red/Green scheme is that it can be implemented using static pointers which require little run-time management. The main disadvantage is that it doubles the amount of storage required for static data structures. Fortunately, it is possible to eliminate many of these data structures, because not all nets can undergo both a red and a green event. For example, consider the circuit pictured in Figure 2.

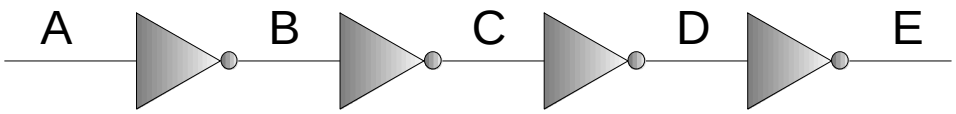


Figure 2. A Sample Circuit.

A simulation of the circuit of Figure 2 will result in at most one event being processed for each of the nets A-F. A red event will be processed for net A, a green event for net B, and so forth. Because no green event will ever be queued for net A, and no red event will ever be queued for net B, it is possible to eliminate the event structures for those events. Since the eliminated structures are never referenced by the simulation code, it may be possible to eliminate the unused structures using “dead code” techniques, however, there is a more effective method for finding and eliminating unusable structures.

The first step in eliminating useless structures is to compute the PC-Set for each gate and each net of the circuit[41]. First, each net is assigned the set {0}. Next, the circuit is processed in levelized order proceeding from the primary inputs to the primary outputs. A gate is processed when each of its inputs have been assigned PC-Sets. When a gate is

processed, the algorithm computes the union of the PC-Sets of the gate-inputs and increments each PC-Set element by 1. This set becomes the PC-Set of the gate. The PC-Set of a net is computed when all gates that drive the net have been assigned PC-Sets. The PC-Set of a net is simply the union of the PC-Sets of the driving gates. Since most nets have a single driving gate, the PC-Set of a net is usually identical to the PC-Set of the driving gate.

The PC-Set of a net can be used to determine which event structures are required. If the PC-Set contains an even number, then a red event structure is required for each fanout branch, and if it contains an odd number, a green event structure is required.

5. Elimination of Additional Structures.

The reason for using two event structures per fanout branch is the possibility of queuing two events for the same net simultaneously. Since this occurs only when events are queued in two consecutive time-slots, it is possible to eliminate one set of event structures for nets that have no consecutive PC-Set elements. However one must use caution when performing this operation, as Figure 3 illustrates. In the network of Figure 3, the net Y has a PC-Set of {1,4}. Because events can occur at both an even and odd time, it is possible for events on Y to occur during both red and green time-slots. However, since these events can never be queued simultaneously, it may appear possible to eliminate one of the event structures for the net. The difficulty is that every event structure must specify the *color* of the events that will be scheduled when an event propagates. If the following net or nets require a strict alternation in red and green events, it is possible for the single event structure to schedule the wrong event structure at certain times, introducing errors into the simulation.

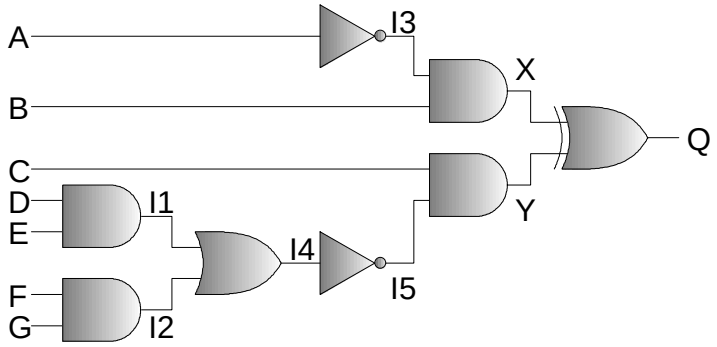


Figure 3. Non-Consecutive PC-Set Elements.

Suppose that the circuit of Figure 3 has been simulated with the input vector (A=0, B=0, C=0, D=0, E=0, F=0, G=0), and then the vector (A=1, B=1, C=1, D=0, E=0, F=0, G=0) is applied. This should cause the output Q to change from 0 to 1. The simulation is illustrated in Figure 4. The reader is encouraged to refer to both Figure 3 and Figure 4 while reading the following discussion, which may be difficult to follow otherwise.

The PC-Set of net X is {1,2}, while that of net Y is {1,4}. Suppose the green structure has been eliminated for net Y and that the red structure will be used for all events. This implies that any event on net Y will propagate a green event to net Q. Since Net X has consecutive PC-Set elements, both the red and the green structure are retained

for this net. The change on Net B will cause Net X to change from 0 to 1 at time 1. The green event structure will be queued for this net. Similarly the change in net C will cause Net Y to change from 0 to 1, which will cause an event to be queued for Y at time 1. Since only the red event structure has been retained, the Red structure will be queued for net Y. An event will also be queued for Net I3 at time 1. The event on I3 will cause Net X to change from 1 to 0 at time 2 (a static hazard). This will cause the Red event structure to be queued for Net X at time 2. Processing the Green event structure for Net X will cause the Red event structure for net Q to be queued at time 2, and processing the Red event structure for net X will cause the green event structure for Net Q to be queued at time 2. Thus, three events will be queued at time 2, a red event for net X and both a red and green event for net Q. Suppose that the event for Net X is processed first. This event will cause an event to be queued for net Q at time 3. However since the green event for net Q is already queued, it will be dequeued from the time 2 slot rather than being queued at time 3. This will leave the red event queued for Net Q at time 2. Although the final result will be the same, the change in Net Q will appear to take place at time 2 rather than time 3, which is an error.

Despite the difficulties, it is possible to eliminate some data structures for nets that have no consecutive PC-Set elements.. For example, consider a circuit identical to the cone of net Y in Figure 3. This circuit would require only a single event structure per net, even though the PC-Set for net Y is {1,4}.

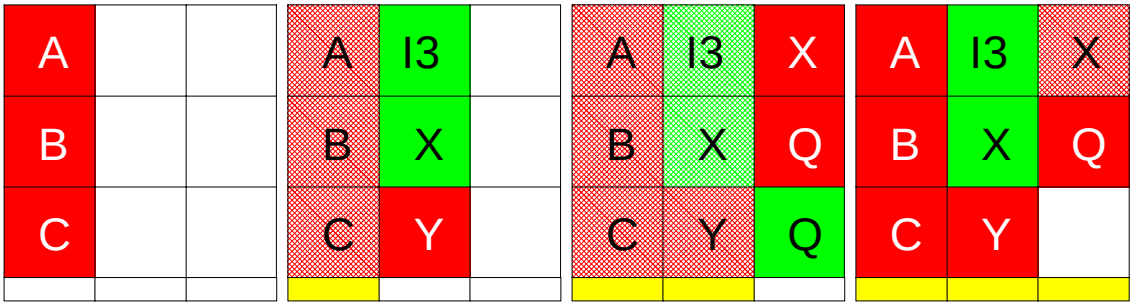


Figure 4. Timing Error due to Event Merging.

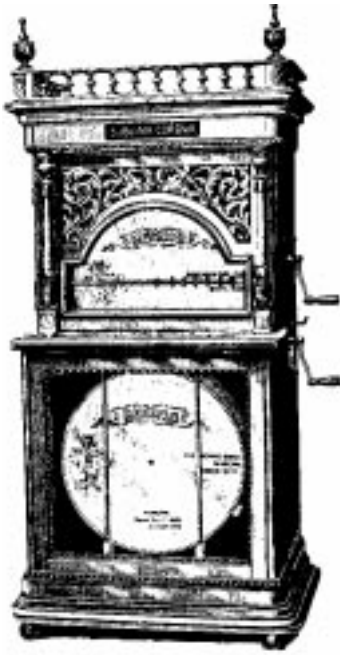
Color analysis is used to determine the number of event structures required for the fanout branches of each net. As a part of this process, the elements of all PC-Sets are set to one of three values: Red, Green, or Colorless. If a PC-Set contains consecutive elements, the even-numbered element of each consecutive pair is set to Red, while the odd-numbered element is set to Green. All other PC-Set elements are set to Colorless. The objective of color analysis is to determine the status of each net, Monochrome, BiColored, or Colorless. BiColored nets require two event structures per fanout branch, while Monochrome and Colorless nets require only one. It is necessary to distinguish between Monochrome and Colorless nets because the process for generating data structures is somewhat different.

In addition to the three final net-states, there are two intermediate states, M+ and B+, which are assigned to Monochrome and BiColored nets with some colorless PC-Set elements. Before a final status can be assigned to M+ and B+ nets, it is necessary to assign colors to the colorless PC-Set elements. It is also necessary to avoid scheduling

conflicts such as those illustrated in Figure 4. This is done by identifying and properly coloring PC-Set elements that could cause scheduling conflicts if left colorless. Five techniques are used to color PC-Set elements, Consecutive Element Coloring as described above, Demand Coloring, Sympathetic Coloring, Minimal Coloring, and Parity Coloring.

Consecutive element coloring is performed once at the beginning of color analysis. The other coloring procedures are executed in the order given above, until no more elements can be colored. The procedure advances to a new coloring method only when no more nets can be colored using the previous method. For example Sympathetic Coloring is done only if no net can be colored by Demand Coloring.

Demand Coloring is applied to all gates G with a BiColored output N. For each Red(Green) element k of N, all elements k-1 are selected from the PC-Sets of the inputs of G. These elements are colored Green(Red). Demand coloring is scheduled whenever an element of a BiColored net is assigned a color, and is performed repeatedly until no more nets can be colored. Demand coloring may change a Colorless net to a Monochrome net or a Monochrome net to a BiColored Net.



Sympathetic Coloring is applied to all gates with an M+ output, N, of color Red(Green). All colorless elements of the PC-Set of N are colored Red(Green). Sympathetic coloring must be propagated to the primary outputs of the circuit. If the element k of net N is colored Red(Green) by Sympathetic Coloring, it is necessary to examine the outputs of any gate which uses N as an input, and select all PC-Set elements of value k+1. These elements must be colored Green(Red). Sympathetic coloring preferentially starts with a PC-Set having all odd or all even elements.

Minimal coloring is applied to colorless nets N whose PC-Sets have more than one element. N must be the input of a gate with a BiColored output. The lowest PC-Set value of N is colored consistently with its parity, and the remainder of the PC-Set elements are colored using Sympathetic Coloring. Minimal coloring must propagate to the primary outputs.

Parity coloring is used only when no other type of coloring applies. Parity coloring is applied to the PC-Sets of B+ nets. The colorless elements are colored consistently with their parity, and the color is propagated to the primary outputs. Parity coloring may create nets that can be colored by one of the other methods.

When none of the above coloring techniques can be applied, the coloring process stops, and data structures are generated for the fanout branches of each net in the circuit. Both red and green data structures are generated for BiColored nets. A single Red(Green) data structure is generated for the fanout branches of monochrome nets. The Red(Green) data structure will schedule a Green(Red) structure for any propagated events. To simplify the scheduling of events for monochrome nets, a second dummy data structure of opposite color is overlaid on top of the generated structure. This allows the data structure to be scheduled either as a Red or a Green data structure.

A single data structure is also generated for each the fanout branch of a colorless net. As for monochrome nets, a dummy data structure is overlaid on top of the first, allowing

the data structure to be scheduled either as a red or a green data structure. For colorless nets, propagated events may schedule either the red or green data structure, whichever is convenient. The following lemma shows why this is possible.

Lemma 2. *Let G be a gate with input net N and output net M. Suppose that N is a colorless net. Then M must be either colorless or monochrome.*

Proof: The only other possibility is that M is BiColored. If this were the case then demand coloring would color at least one PC-Set element of N. Once this were done, N would no longer be colorless.

As Lemma 2 shows, whenever an event is propagated from a colorless net, the net to which it is propagated has a single data structure. Since each single data structure is overlaid with a dummy structure of the opposite color, the structure can be scheduled as either a red or a green data structure with identical results.

It is possible to eliminate even more data structures by performing color analysis on individual fanout branches instead of whole nets. Since data structures are generated for each fanout branch instead of each net, it is possible for a BiColored net to have only monochrome fanout branches. Under the current scheme, these nets would have two data structures generated for each fanout branch when, strictly speaking, only one is necessary.

6. Elimination of Gates and Connections

In the zero-delay Inversion Algorithm it is possible to eliminate NOT, BUFFER, XOR and XNOR gates. It is also possible to collapse heterogeneous and homogeneous connections. (See reference [35] for an explanation of this terminology.) However in the Unit-Delay model these optimizations cause a fundamental change in the timing model. Although it is possible to eliminate all NOT gates from the simulation without changing the final values computed by the simulator, it is necessary to retain the delay of all eliminated gates to avoid invalidating the hazard analysis. For example, a chain of NOT gates may have been added to the circuit to balance path-lengths and eliminate hazards. If the NOT gates are simply deleted from the simulation without retaining the delay, the NOT chain will appear to have no effect on the dynamic behavior of the circuit.

Eliminating gates and connections effectively transforms the unit-delay simulation into a multi-delay simulation under the transport-delay model. (The inertial delay model is inappropriate, because gates exhibiting delays larger than 1 are actually collections of simpler gates.) This compounds the problems that led to the adoption of the red/green model. Figure 5 illustrates the collapsing of gates and connections. Once the NOT gate and the homogeneous connection have been collapsed out of the circuit, the resulting gate has three inputs, one (C) with a delay of 1, and two (A and B) with delays of 3. These delays, which are constant and associated with gate inputs, indicate the number of time-slots that must be added to the current slot to find the appropriate queue location for

propagated events. It is possible for different fanout branches of a net to have different delays.

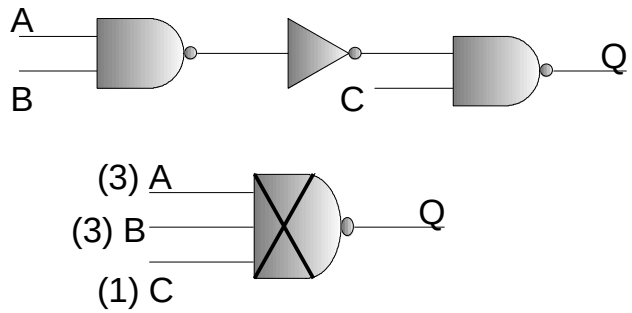


Figure 5. Collapsed Gates and Connections.

The following lemma is the multi-delay extension of Lemma 1.

Lemma 3. *Let* G *be a gate with output net* N *and let* k *be the maximum delay on any input of the gate. The maximum number of events that can be queued simultaneously for the net is* k+1.

Proof. Similar to that of Lemma 1.

More important than the lemma itself is the following corollary.

Corollary 3.1. *Let* k *be the maximum delay value over all inputs of all gates. Then the maximum number of events that can be queued for any net is* k+1.

Corollary 3.1 allows us to extend the idea of the red/green technique to multiple colors. Instead of red and green we will use the set of colors {0,1,2,...,*k*}, where *k* is the maximum delay described in Corollary 3.1. We then allocate *k*+1 event structures for each net which are colored with the colors 0 through *k*. If *i* is less than *k*, then all data structures of color *i* will schedule structures of color *i*+1. Structures of color *k* will schedule structures of color 0.

7. Eliminating Data Structures for Collapsed Networks.

Using *k* structures for each fanout-branch in the circuit is sufficient to avoid conflicts, but not necessary. As is the case for uncollapsed networks, it is possible to identify unused structures and eliminate them. It is first necessary to compute the PC-Sets of the collapsed network, using a slightly modified algorithm. Because delays are associated with the inputs of a gate rather than with the gate itself, it is necessary to add the delay to each PC-Set element before forming the union of the PC-Sets of a gate's inputs. Since delays are not necessarily one, it is the actual delay associated with the input that must be added to each element of the input's PC-Set before forming the union.

Each element *e* of a PC-Set is categorized using the function (e mod *k*+1), where *k* is the maximum color value. If the PC-Set for a net N has no elements of category *c*, then a structures of color *c* are not required. More structures may be eliminated by performing an operation similar to the search for adjacent PC-Set elements described above. However, because delays are not uniformly 1, the PC-Set does not provide enough information to determine the number of colors needed by a particular net. The maximum number of colors needed by a net is equal to the maximum number of events that may be simultaneously queued for the net. The PC-Set gives information about when events will be processed, but no information about when events are queued. To determine the maximum number of events that can be queued at any one time, it is necessary to know both when an event enters the queue and when it leaves the queue. To make this determination, it is necessary to compute the Queue Density Function $D_N(i)$. For each net N, $D_N(i)$ gives the maximum number of events that can be queued at time *i*. Technically, the domain of the queue density function is the entire set of natural numbers, but the only domain elements that are of interest are 0-*m*, where *m* is level number of the net in question.

The first step in computing $D_N(i)$ is to compute the PC-Set of each net in the circuit. Once all PC-Sets have been computed, they can be used to compute the queue density functions in the following manner. Let G be a gate with *n* inputs. If the largest delay on any of the *n* inputs is *k*, then create a queue *Q* with *k*+1 elements. The queue consists of a collection of Boolean values, which indicate whether the corresponding time slot is empty or full. The queue density function is computed by Algorithm 1.

```
For k := 0 to n do
    Push empty onto tail of Q;
EndFor;
Initialize the set S to empty;
For each input i do
    For each element x of the PC-Set of i do
        d := DelayOf(i);
        Add (x,d) to S;
    EndFor
EndFor
/* Sort by PC Set Value */
Sort the elements of S into ascending order
        by the value of the first coordinate;
CurrentQueuePosition := 0;
/* Process elements of S in sorted order */
```

For each (*x*,*d*) in *S* **do**
    While *CurrentQueuePosition* < *x* **do**
        k := The number of *full* positions in *Q*;
        Set the value of $\mathbf{D_N}$(*CurrentQueuePosition*) to *k*;
        Pop Head of *Q*;
        Push *empty* onto tail of *Q*;
        CurrentQueuePosition **:=** *CurrentQueuePosition* **+** *1*;
    EndWhile
    /* First Queue Position is 0 */
    Set the *d*th element of *Q* to *full*;
EndFor
While *Q* is not empty **do**
    k := The number of *full* positions in *Q*;
    Set the value of $\mathbf{D_N}$(*CurrentQueuePosition*) to *k*;
    Pop Head of *Q*;
    CurrentQueuePosition **:=** *CurrentQueuePosition* **+** *1*;
EndWhile

Algorithm 1. Queue Density Function Computation.

Algorithm 1 can also be used to compute the Queue Population Function $P_N(i)$, which will be useful in assigning colors to slots. The function $P_N(i)$ is similar to $D_N(i)$, but $P_N(i)$ returns the set of filled queue positions, instead of the number of filled positions. The function $D_N(i)$ can be computed from $P_N(i)$, but the reverse is not true.

Once all queue density functions have been computed, it is possible to determine the maximum number of colors required by each net. Let N be a net with queue density function *f*. Assume further that the level of net N is *m*. Let *c* be the maximum of *f*(*x*) 0<=*x*<=*m*. The events of N can be scheduled without conflict using no more than *c* colors. Unfortunately, *c* colors may not be enough to prevent scheduling conflicts in successor gates. Even if *C* is the maximum value over all queue density functions, it may be necessary to use more than *C* colors to schedule the entire circuit without conflict. Two types of conflicts arise when combining gates into networks: parent-child conflicts and sibling conflicts. The circuit pictured in Figure 6 illustrates a parent-child conflict. In this figure, the numbers in curly braces are the PC-Sets of the corresponding nets, while the numbers indicated by "d=" are the delays associated with the inputs.

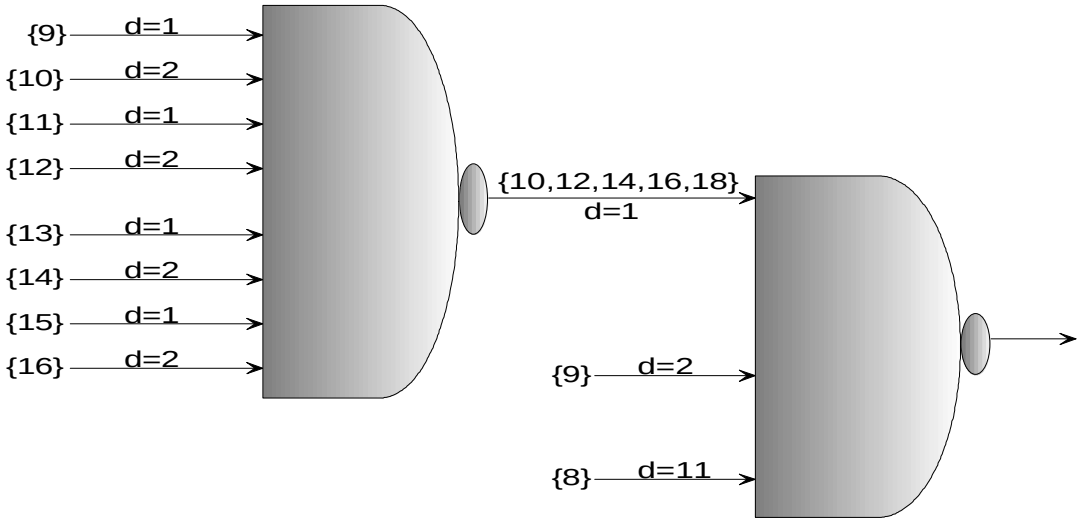


Figure 6. Scheduling Conflicts.

In Figure 6, there are only two queue density functions of interest, those for the outputs of the NAND gates. The maximum queue density for either of these nets is 2. However to avoid scheduling conflicts, it is necessary to use 3 colors to schedule the events of the first NAND gate. To illustrate how the conflict occurs, consider the computation of the queue density functions illustrated in Figure 7 and Figure 8.

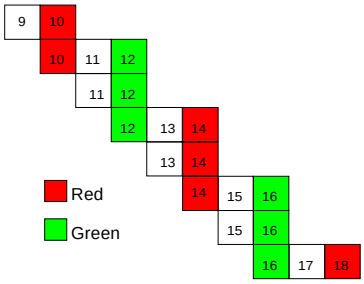


Figure 7. Computation of the First Queue Density Function.

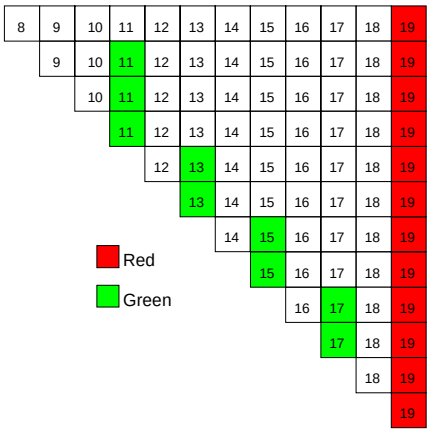


Figure 8. Computation of the Second Queue Density Function.

In Figure 8 slots 17 and 19 must be assigned different colors, since events can be queued in slot 17 and slot 19 simultaneously. Since events are propagated using static pointers, this requires two different colors to be assigned to slots 16 and 18 in Figure 7. If these two slots were assigned the same color, they could not queue events of two different colors. In Figure 8 slot 15 must have a different color from slot 19, which implies that in Figure 7, slot 14 must have a different color from slot 18. But in Figure 7 slot 14 and 16 must have different colors as well. Putting this all together, in Figure 7, slots 16 and 18 must be of different colors, slots 16 and 14 must be of different colors and slots 18 and 14 must be of different colors. This cannot be accomplished without using three colors. However, as the figures clearly show, the maximum queue density of either net is 2.

To avoid parent-child conflicts, it is necessary to propagate color information from the output of a gate to its inputs, but propagating this information may give rise to sibling conflicts. (Color propagation is the multi-color analog of Demand Coloring.) Sibling conflicts arise between two or more fanout branches of a single net. To understand how such conflicts arise, it is necessary to have a precise understanding of the scheduling mechanisms used in the Inversion Algorithm. The data structures used for scheduling are illustrated in Figure 9.

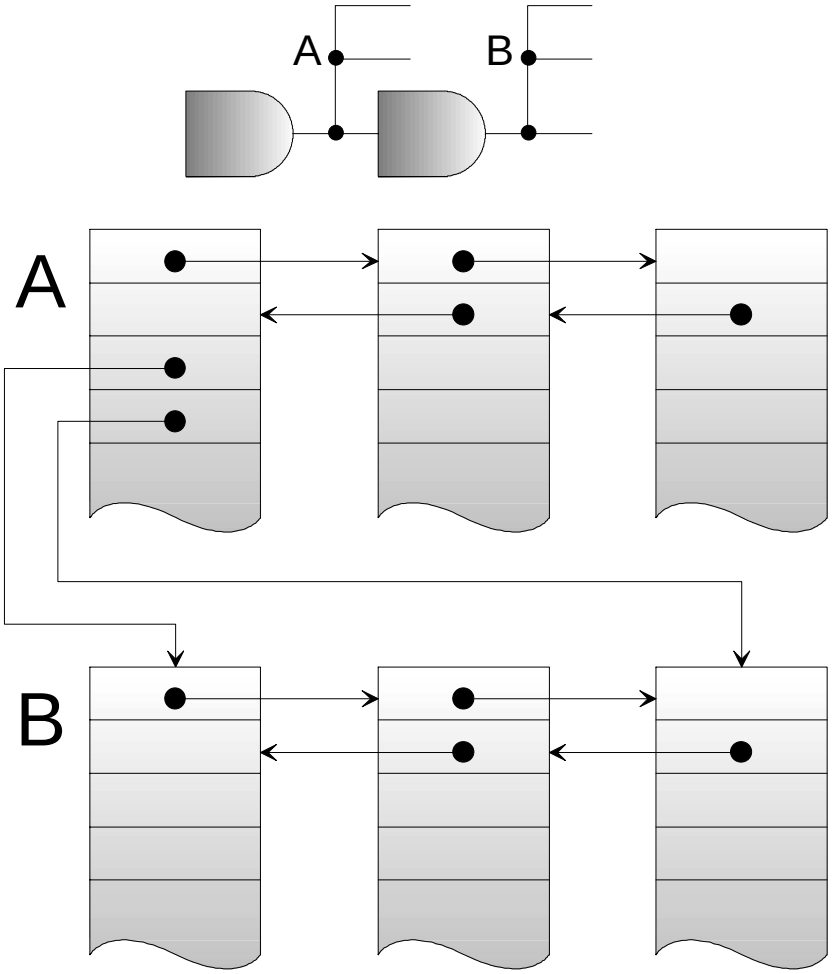


Figure 9. Scheduling Data Structures.

As Figure 9 illustrates, each fanout branch of a net is represented by a separate data structure. The set of data structures representing a net are chained together using static forward and back pointers. Each data structure contains static pointers to the chain that will be scheduled if an event propagates. Because static pointers are used, no data structure may appear in more than one chain. Figure 10 illustrates how sibling conflicts occur.

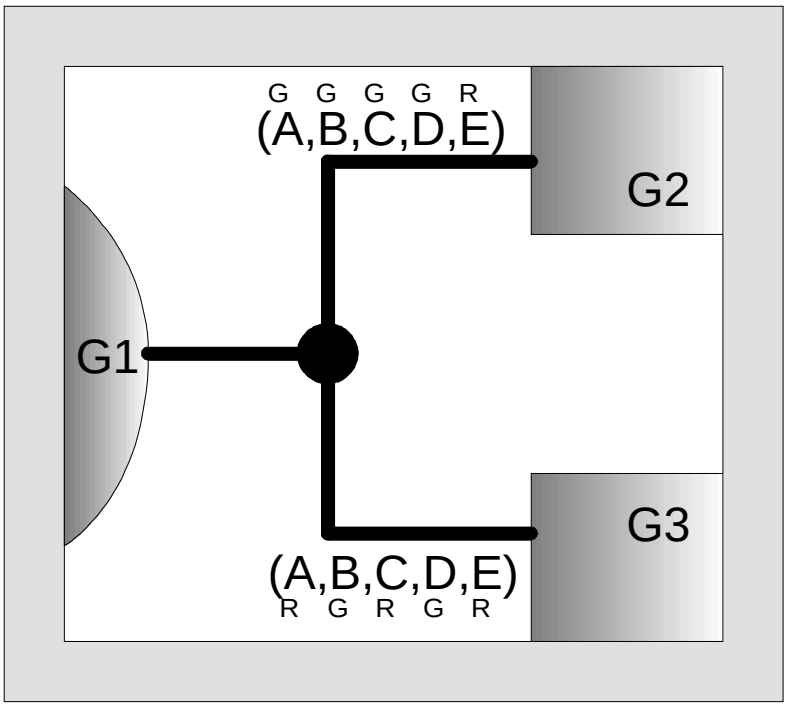


Figure 10. Sibling Conflicts.

In Figure 10, it is assumed that colors have been assigned to all time slots of the outputs of G2 and G3, and that color information has been propagated from outputs to inputs. Assuming that the net pictured in Figure 10 has the PC set {A,B,C,D,E}, it is necessary to determine the color of the structures that must be scheduled if an event propagates at any one of these times. Assuming further that the colors Red and Green are sufficient to schedule events for the outputs of G1 and G2, the indicators R and G indicate which data structures must be scheduled at each time slot. During time slot A, a Green event must be scheduled for gate G2, and a Red event must be queued for gate G3. During time slot B, a green event must be queued for both gates. Writing the combinations as a sequence of ordered pairs, the complete list is (G,R), (G,G), (G,R), (G,G), and (R,R). There are three distinct pairs, (G,G), (G,R), and (R,R). Because static pointers are used, it is necessary to create three distinct chains of data structures. In the first chain, the data structures point to the green structures for both nets, in the second, the data structures point to the red structures, while in the third one points to the red structures, and the other points to the green structures. The data structures are illustrated in Figure 11. Note that in this figure, it has been necessary to use three different colors to construct the data structures for the net. The information provided by the queue population function may require additional data structures to be created.

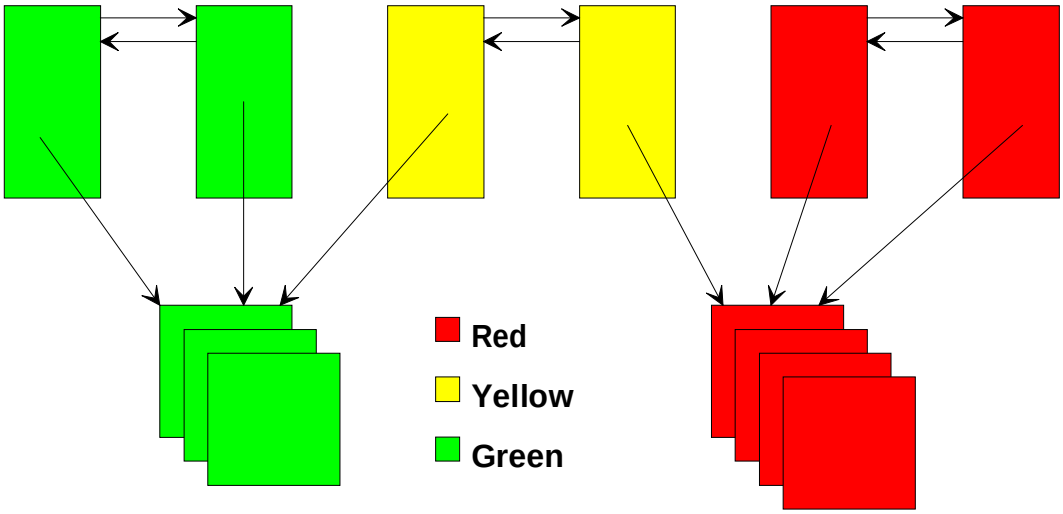


Figure 11. Sibling Conflict Resolution.

We have developed a colorizing function which can color the time-slots of all nets in such a way as to avoid both parent-child conflicts and sibling conflicts. Before applying the coloring function to a network, it is necessary to break any cycles and levelize the resultant acyclic network. For synchronous sequential circuits, this can be done in straightforward way by breaking each synchronous flip-flop[44]. For asynchronous sequential circuits, a method such as the convergence algorithm outlined in [40] can be used. Colorizing starts with all fanout-free primary outputs of the network. Algorithm 2 is used to assign colors to these nets. (This form of the algorithm is for illustrative purposes only. In practice, the obvious changes will be made to improve performance.)

For Each Fanout Free output *x* **do**
  For Each element *p* of the PC set of *x* **do**
    For Each time-slot *i* in the set $P_x(p)$ **do**
      j := 0;
      While there is a time-slot in $P_x(p)$ colored *j* **do**
        j := *j*+1;
      EndWhile
      Color time-slot *p* with color *j*;
    EndFor
  EndFor
EndFor;

Algorithm 2. Assign Colors to Primary Outputs.

Once all primary outputs have been colored, the main coloring algorithm, Algorithm 3, is used to color the remainder of the network. This algorithm is based on two queues, a queue of gates, *GQ*, whose input nets require coloring, and a queue of nets, NQ, whose fanout branches must be combined to resolve sibling conflicts. In Algorithm 3, the PC-Sets of the fanout branches of a net are considered to be distinct from the PC-Set of the net itself. The **Propagate** function propagates colors to fanout branches, the **Combine**

function propagates colors from the fanout branches to the net itself, and the **Recolor** function refines the coloring generated by the **Combine** function.

```
For Each gate G whose output is a fanout-free Primary Output do
    Add G to GQ;
EndFor
While GQ is not empty do
    For Each gate G with output N in GQ do
        Propagate the coloring of N to each input I of G;
        Delete G from GQ;
        If all fanout branches of the input I have been colored Then
            Add I to NQ;
        EndIf
    EndFor;
    For Each Net N in NQ do
        Combine the color information of the fanout branches of N;
        Recolor slots based on the Queue Population Function of N;
        Delete N from NQ;
        If N is the output of a gate G Then
            Add G to GQ;
        EndIf
    EndFor
EndWhile
```

Algorithm 3. The Main Coloring Algorithm.

Algorithm 4 is the **Propagate** function used in the main coloring algorithm. This algorithm propagates colors to the fanout branches of a net, not to the net itself. No conflicts can arise during this process.

```
Propagate(N:OutputNet,G:Gate)
begin
    For Each input I of G do
        d := DelayOf(I);
        For Each time-slot t in the PC Set of N do
            c := ColorOf(t);
            If the PC Set of I contains the element t-d Then
                Color the element t-d with color c,
                    in the PC Set of the fanout branch of I
                    leading to G;
            EndIf
        EndFor
    EndFor
End Propagate;
```

Algorithm 4. The Propagate function.

Algorithm 5 is the **Combine** function used to prevent sibling conflicts. This algorithm also illustrates how forward scheduling information is obtained for each data structure.

```
Combine(N:Net)
Var S:Set of Tuples, k:Integer;
begin
    S := The Empty Set;
    k := The number of Fanout branches of N;
    For Each element t in the PC Set of N do
        If the k-tuple of propagated colors for time slot t
            is not already contained in S Then
                Add the k-tuple of propagated colors to S;
        Endif;
    EndFor;
    Assign the colors 0 through |S|-1 to the elements of S,
            and to the corresponding time-slots;
    /* Retain information for the Data-Structure Generator */
    Retain the set S, and links between the elements of S and
        time slots;
End Combine;
```

Algorithm 5. The Combine Function.

The **Combine** function not only performs an initial color assignment, it also creates a vital piece of scheduling data, the *k*-tuple of colors associated with each time-slot. Effectively, this algorithm assigns a two-dimensional color to each time slot. The first component is the color created by the **Assign** statement, while the second component is the *k*-tuple of propagated colors. Both the color and the *k*-tuple will be used to create and link the final data structures.

Finally, Algorithm 6 is the **Recolor** function used to assign the final colors to the time-slots of a net. Although it does not appear explicitly in the algorithm, the association between time-slots and *k*-tuples is maintained throughout the recoloring process.

```
Recolor(N:Net)
Var K:k-tuple, c:color;
begin
  Repeat
    For Each element t in the PC Set of N do
      For Each element s in P_N(t) do
        If s has the same color as a previous element of P_N(t) Then
          K := the k-tuple associated with t;
          If there is a time-slot u of color c,
            and c does not appear in P_N(t),
            and c is greater than the current color of s Then
              Assign c to s;
          Else
            c := The smallest color not used to color any element of
                 the PC-Set of N;
            Assign c to s;
          EndIf
        EndIf
      EndFor
    EndFor
  Until No Slots are Recolored;
End Recolor;
```

Algorithm 6. The Recolor Function.

Note that when a time-slot is recolored by the **Recolor** function, the new color will always be numerically larger than the existing color. This eliminates any circularity problem that may occur should it be necessary to recolor a time-slot more than once. Since it is possible to recolor events more than once, it is possible that multiple recolorings of several nets will leave the net in a state where scheduling conflicts are still possible. Hence it is necessary to repeat the recoloring process until no more nets can be recolored.

Once the main coloring algorithm has completed, it is possible to create scheduling data-structures for each of the nets. First, a set of data structures is created for each net. Suppose a net *N* has fanout *k* and that *c* colors have been used to color the time-slots of the net. In this case *c* chains of data structures will be created, each one of which has *k* elements. The forward scheduling information for each data structure will be taken from the *k*-tuple associated with the color of the data structure. Although, strictly speaking, *k*-tuples are associated with slots rather than colors, the process of assigning colors to slots guarantees that if two slots have the same color, then they are associated with the same *k*-tuple.

The scheduling and simulation code is quite simple, and essentially identical to that used by the zero delay algorithm. For the sake of completeness, this code is replicated in Figure 12. In this code, the scheduling data structures are referred to as *Shadows*[38]. This code is written in C, the Inversion Algorithm implementation language. The computed goto at the end is actually implemented in assembly language.

```
INCREMENTX:
   /* Alternate the INC & DEC processors */
   *Current_Shadow->subroutine = &DECREMENTX;
   (*Current_Shadow->Lock)++;
   /* If a change in the output will occur */
   if ((*Current_Shadow->Lock) == 1)
   {
      /* If the gate is not already queued */
      if (Current_Shadow->last_fanout->next ==
         Current_Shadow->last_fanout)
      {
         /* queue the gate for simulation */
         if (Queue_Tail != NULL)
         {
         Queue_Tail->next =
            Current_Shadow->first_fanout;
         Current_Shadow->first_fanout->previous =
            Queue_Tail;
         }
         else
         {
            Queue_Head =
      `        Current_Shadow->first_fanout;
            Current_Shadow->first_fanout->previous =
               NULL;
         }
         Queue_Tail = Current_Shadow->last_fanout;
         Current_Shadow->last_fanout->next = NULL;
      }
      else
      {
         /* dequeue the gate */
         Current_Shadow->last_fanout->next->previous =
            Current_Shadow->first_fanout->previous;
         Current_Shadow->first_fanout->previous->next=
            Current_Shadow->last_fanout->next;
         Current_Shadow->last_fanout->next = NULL;
         Current_Shadow->first_fanout->previous =
            Current_Shadow->first_fanout;
      }
   }
   Temp = Current_Shadow->next;
   Current_Shadow->next = Current_Shadow;
   Current_Shadow = Temp;
   if (Current_Shadow == NULL) return;
   Goto **Current_Shadow->subroutine;
```

Figure 12. Inversion Algorithm Code.

As noted in reference [35] the run-time code for the Inversion Algorithm consists of several (less than ten) slightly different versions of the routine pictured in Figure 12.

8. Conclusion.

Although the Inversion Algorithm has been proven to be effective for zero-delay simulation, there has been some question about its application to more complex timing models. This paper shows that the algorithm can be adapted to the Unit-Delay model in a straightforward way. The two most significant problems in this adaptation, are providing the ability to eliminate gates and connections the same way this is done in the zero delay

model, and reducing the number of duplicate data structures required to prevent scheduling errors. This paper has also shown, in a preliminary way, how the more complex multi-delay model could be used with the Inversion Algorithm. However, in the Unit-Delay model, even after deletion of NOT gates and collapsing connections, the number of non-unit delays should be relatively small. In the multi-delay model, non-unit delays tend to be the norm, which may impose more stringent demands on the elimination of duplicate data structures. The multi-delay model also supports several types of delay including both transport and inertial delay. The current paper discusses only the transport delay model. Support for the inertial delay model, with event canceling, will be significantly different.

This paper serves to demonstrate the versatility and adaptability of the Inversion Algorithm. It must be noted that the run-time code required by the Unit-Delay model is virtually identical to the run-time code for the Zero-Delay model. Due to the extremely small size of the code, the Unit-Delay model should be just as adaptable as the Zero-Delay model. Finally, this paper demonstrates that the Inversion Algorithm is a widely applicable technique that will prove to be an effective design automation tool in the future.

9. Bibliography.

1. Abramovici, M., Y. Levendel, P. Menon, “A Logic Simulation Machine,” *Proceedings of DAC-19,* June 1982, pp. 65-73.
2. M Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing, and Testable Design,* Computer Science Press, New York, 1990.
3. Agrawal, P. “Concurrency and Communication in Hardware Simulators,” *IEEE Transactions on Computer-Aided Design,* Vol. CAD-5, No. 4, Oct. 1986, pp. 617-622.

4. Agrawal, P., W.J. Dally, W.C. Fisher, H.V. Jagadish, A.S. Krishnakumar, and R. Tutundjian, "MARS: A Multiprocessor-Based Programmable Accelerator," *IEEE Design and Test of Computers*, Oct. 1987, pp. 28-36.

5. V. Aho, J. E. Hopcoff, J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

6. Appel, A. W., "Simulating Digital Circuits with One Bit Per Wire," *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.

7. Ashok, R. Costello, P. Sadayappan, "Modeling Switch-Level Simulation Using Data Flow," Proceedings of the 22nd Design Automation Conference, 1985, pp. 637-644.

8. W. Y. Au, D. Weise, S. Seligman, "Automatic Generation of Compiled Simulations through Program Specialization," *Proceedings of the 28th Design Automation Conference*, 1991, pp. 205-210.

9. Barzilai, Z., J. L. Carter, B. K. Rosen J. D. Rutledge, "HSS–A High-Speed Simulator," *IEEE Transactions on Computer Aided Design*, Vol. CAD-6, No. 4, , pp. 601-617, July, 1987.

10. Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, G. M. Silberman, "SLS-A Fast Switch-Level Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 8, August, 1988, pp. 838-49.

11. Bell, J. "Threaded Code," *Communications of the ACM*, Vol. 16, No. 6, June 1973, pp. 370-372.

12. Blank, T. "A survey of hardware accelerators used in computer-aided design," *IEEE Design and Test of Computers*, vol. 1, Aug. 1984, pp. 21-39.

13. Breuer, M. A., A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Woodland Hills, CA, 1976.

14. Brglez, F., D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of ISCAS-89*, pp. 1929-1934.

15. Brglez, F., P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of the International Conference on Circuits and Systems*, 1985, pp. 695-698.

16. R. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," Proceedings of the 24th Design Automation Conference, 1987, pp. 9-16.

17. Bryant, R. E., "Data Parallel Switch-Level Simulation," *IEEE International Conference on Computer Aided Design*, 1988, pp. 354-357.

18. Catthoor, J. van Sas, L. Inze, H. de Man, "A Testability Strategy for Multiprocessor Architecture," IEEE Design & Test of Computers, Apr. 1989, pp. 18-34.

19. Chappell, S. G., H. Y. Chang, C. H. Elmendorf and L. D. Schmidt, "Comparison of Parallel and Deductive Simulation Techniques," *IEEE Transactions on Computers*, Vol C-23, pp. 1132-1139.

20. Chen, C. F., C-Y Lo, H. N. Nham, P. Subramaniam, “The Second Generation Motis Mixed-Mode Simulator,” *Proceedings of the 21st Design Automation Conference*, pp. 10-17, 1985.

21. Chiang, M., and R. Palkovic, “LCC Simulators Speed Development of Synchronous Hardware,” *Computer Design*, Mar. 1, 1986, pp. 87-91.

22. Coelho, "VHDL: A Call for Standards," Proceedings of the 25th Design Automation Conference, 1988, pp. 40-47.

23. Denneau, M., “The Yorktown Simulation Engine,” *Proceedings of DAC-19,* June 1982, pp. 55-59.

24. Doshi, R. B. Sullivan, D. M. Schuler, "Themis Logic Simulator - A Mix Mode, Multi-Level, Hierarchical Interactive Digital Circuit Simulator," Proceedings of the 21st Design Automation Conference, pp. 24-31, 1985.

25. "Electronic Design Interchange Format Version 2 0 0" Recommended ANSI Standard EIA-548, Electronic Industries Association, Washington D.C., 1988.

26. S. Gai, F. Somenzi, M. Spalla, "Fast and Coherent Simulation with Zero Delay Elements," *IEEE Trans. on Computer-Aided Design,* Vol. 6, No. 1, Jan., 1987, pp.85-91.

27. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of 25th Design Automation Conference,* 1988, pp.712-15.

28. Heydemann, M., D. Dure, “The Logic Automation Approach to Accurate and Efficient Gate and Functional Level Simulation,” *Proc. ICCAD-88,* 1988, pp. 250-253.

29. Lee, T. F. Fang, "A Mixed-Mode Analog-Digital Simulation Methodology for Full Custom Designs," Proceedings of the IEEE 1988 Custom Integrated Circuits Conference, pp. 3.5.1-3.5.4.

30. Lee, Y, and P. Maurer, “Two New Techniques for Event-Driven Compiled Multi-Delay Simulation,” *Proceedings of 29th Design Automation Conference*, June, 1992.

31. Lee, Y., P. Maurer, “Parallel Multi-Delay Simulation,” *ICCAD-93*, Nov. 1993, to appear.

32. Lee, Y., P. Maurer, “Two New Techniques for Compiled Multi-Delay Simulation,” *Proceedings of Southeastcon 92*, Apr, 1992.

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

34. Lewis, D. M., “Hierarchical Compiled Event-Driven Logic Simulation,” *Proceedings of ICCAD-89*, pp.498-501.

35. P. M. Maurer, “The Inversion Algorithm for Digital Simulation,” *Proceedings of ICCAD-94*, pp. 258-61.

36. P. M. Maurer “Three-Valued Simulation with the Inversion Algorithm,” Submitted for Publication. Available from the author (maurer@usf.edu).

37. P. M. Maurer, "FHDL Tutorial," University of South Florida, Department of Computer Science and Engineering Technical Report Number CSE-88-00011, Tampa, FL 33620, 1988.

38. Maurer, P., “The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation,” *IEEE Transactions on Computer Aided Design*, Vol. 12 No. 9, Sept. 1993, pp. 1411-1412.

39. Maurer, P., “Optimization of the Parallel Technique for Unit-Delay Compiled Simulation,” *Proceedings of ICCAD-90*, pp. 70-73.

40. Maurer, P. “Gateways: A Technique for Adding Event-Driven Behavior To Compiled Unit-Delay Simulations,” *IEEE Transactions on Computer Aided Design*, Vol. 13, No. 3, Mar. 1994, pp. 338-352.

41. Maurer, P. M., and Z. Wang, “Techniques for unit-delay compiled simulation”, *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-484.

42. Maurer, P. M, “Two New Techniques for Unit-Delay Compiled Simulation,” *IEEE Transactions on Computer Aided Design*, Vol. 11, No. 9, Sept. 1992, pp. 1120-1130.

43. Maurer, P. M., “Scheduling Blocks for Hierarchical Compiled Simulation of Combinational Circuits,” *IEEE Transactions on Computer Aided Design*, Vol. 10, No. 2, Feb. 1991, pp. 184-192.

44. Maurer, P. M., Z. Wang, C. D. Morency, “Techniques for Multi-Level Compiled Simulation,” Department of Computer Science and Engineering Technical Report CSE-89-04, University of South Florida, 1989.

45. P. M. Maurer, Z. Wang, C. Morency, A. Tokuta and N. Bhate, “The Florida Hardware Design Language,” *Proceedings Southeastcon-90*, pp. 430-434.

46. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI" IEEE Transactions on Computer Aided Design, Vol CAD-4, No. 3, July 1985, pp. 336-349.

47. Pfister, G., “The Yorktown Simulation Engine: Introduction,” *Proceedings of DAC-19*, June 1982, pp. 51-54.

48. D. Schuler “Simulation of NAND Logic,” *Proceedings of COMPCON 72*, Sept. 1972, pp. 243-245.

49. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM", *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.

50. Szygenda, S. A., E. W. Thompson, “Digital Logic Simulation in a Time Based Table Driven Environment,” *Computer*, March 1975.

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

52. Takasaki, S., F. Hirose, A. Yamada, “Logic Simulation Engines in Japan,” *IEEE Design and Test of Computers*, Oct. 1989, pp. 40-49.

53. Tham, K., R. Willoner, D. Wimp, “Functional Design Verification by Multi-Level Simulation,” *Proceedings of the 21st Design Automation Conference*, 1984, pp. 473-478.

54. "VHDL Language Reference Manual," IEEE Standard 1076, IEEE Computer Society Press, Los Amitos, CA, 1987.

55. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, " *Journal of the ACM*, Vol. 21, No. 9, Sept., 1978, pp. 777-85.

56. E. G. Ulrich, D. Herbert, "Speed and Accuracy in Digital Network Simulation based on Structural Modeling," *Proceedings of the 19th Design Automation Conference*, 1982, pp.587-93.

57. E. G. Ulrich, "Concurrent Simulation at the Switch, Gate, and Register Levels," *Proceedings of the 1985 International Test Conference*, 1985, pp.703-9.

58. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," *Proceedings of the 24th Design Automation Conference*, 1987, pp.2-8.

59. Wang, Z., P. Maurer, "Scheduling High-Level Blocks for Functional Simulation," Proceedings of the 26th Design Automation Conference, 1989.

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