Timing Verification Using Statically Sensitizable Paths

JACQUES BENKOSKI, MEMBER, IEEE, ERIK VANDEN MEERSCH, LUC J. M. CLAESEN, MEMBER, IEEE AND HUGO DE MAN, FELLOW, IEEE

Abstract—**This paper presents a new approach to the false path problem in timing verifiers. This approach is based upon the modeling of both the logic and timing behavior of a circuit. Using the logic propagation conditions associated with each delay, efficient algorithms have been developed to find statically sensitizable paths. These algorithms simultaneously perform a longest path search and a partial verification of the sensitizability of the paths. The resulting paths undergo a final and complete sensitization. The algorithms find the longest statically sensitizable path, whose length is a lower bound to the critical path length, and its associated sensitizing input vector. The algorithms can be easily modified to provide an ordered list of all the statically sensitizable paths above a given threshold. An initial analysis of the circuit by the PERT algorithm guides the critical path search and allows pruning of subgraphs that cannot lead to the solution. The results of our experiments show that the presented techniques succeed in curbing the combinatorial explosion associated with the longest statically sensitizable path search.**

I. Introduction

In THE last few years, increasing emphasis has been placed on the need for fast verification of the timing constraints of VLSI circuits. With the recent appearance of many timing optimization tools, this need has been even more strongly felt. While in the past only a rough estimate of the longest propagation path in a circuit was sufficient, the development of new applications with tighter timing constraints has pushed the timing verification tools to their limits.

Timing verifiers [6], [7], [11] have been available for many years, and their wide use by the design community is the best proof of the need for such tools. Their principal advantage over simulators is that they find the longest (critical) propagation path without requiring input excitations. In the timing verification approach, the logic behavior of the circuit is disregarded and the timing behavior is approximated by simple delay models. As a result of this abstraction, the problem of finding the longest propagation path in a circuit is equivalent to finding the longest path in a graph where the weights on the edges represent the delays of the gates. This problem can be easily solved in linear time $O(|E|)$ by the PERT or the DFS algorithms [5]. Because the logic behavior is not taken into account, the critical path found by these algorithms might not be logically meaningful, i.e, no input vector will exercise the path. In this case, the path is said to be a *false path*.

The problem of eliminating the false paths is well known. Some of the previous approaches use case analyses to explicity eliminate some of the false paths, while others try to automatically recognize the direction of the signal flow. More recently, the need to formally account for the logic behavior of the circuit has been increasingly recognized. The approach presented in this paper uses a model of the logic behavior and verifies the static sensitizability of the paths during the longest path search. By definition, statically sensitizable paths are associated with an input vector that activates them. The longest statically sensitizable path is only a lower bound on the true critical path, but it is guaranteed to exist. Several algorithms have been recently developed to compute upper bounds to the length of the true critical path that are tighter than a simple PERT estimate [2], [4], [9], [12]. To date, no algorithm has appeared in the literature which is guaranteed to find the true critical path over a circuit. Unless and until such an algorithm appears and demonstrates reasonable performance, the length of the longest statically sensitizable path length can be used to calibrate the upper bounds. Even if much tighter upper bounds were found, there is still no guarantee of accuracy without the lower bound. Moreover, if the longest statically sensitizable path is equal to the upper bound, then the designer is ensured of the accuracy of this result. Finally, the input vector exercising the path is generated as a by-product of the statically sensitizable path algorithms. This input vector can be directly used in a simulator to verify the accuracy of the timing verifier which only uses estimates of the delays.

In the next section, we will describe the different aspects of the false path problem and we will show that the false path problem is inherent to timing verifiers. In Sec-

Manuscript received January 19, 1988; revised July 18, 1988 and April 23, 1989. This work was supported in part by the EC under the ESPRIT-1058 Project, in part by Philips and Silvar-Lisco, and in part by the Belgian IWONL and MIETEC. This paper was recommended by Associate Editor R. K. Brayton.

This is an expanded version of the paper presented at ICCAD-87.

J. Benkoski was with IMEC, Heverlee, Belgium, on leave from the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA. He is now with Central Research and Development, SGS-Thomson, 38019 Grenoble, France.

E. Vanden Meersch was with IMEC, Heverlee, Belgium. He is now with MIETEC, B-1140 Brussels, Belgium.

L. J. M. Claesen and H. De Man are with IMEC, B-3030 Heverlee, Belgium.

IEEE Log Number 9036680.

tion III, previous attempts to solve the problem are reviewed. In Section IV, we will elaborate on our approach and on the relationship between statically and dynamically sensitizable paths. In Section V, we will describe the SLOCOP environment, which has served as a basis for the development of our algorithms. Section VI deals with the method used to verify the static sensitizability of the paths. The issues related to the complexity of the search for the longest path are discussed in Section VII, while the algorithms are presented in Section VIII. In Sections IX and X, the results of extensive experiments are shown and the performance of the algorithms is analyzed.

II. The False Path Problem

One of the principal reasons for the success of timing verifiers is that they do not require the specification of input stimuli in order to find the longest propagation path. Before the development of this approach, designers had to rely on their experience and intuition in order to determine the critical portions of their designs, and the verification of the timing constraints was based upon extensive simulation. This technique was not only expensive from the computation viewpoint, but it also delegated to the designer the difficult task of determining the combination of input values that would result in the longest propagation time. As timing verifiers compute the latest arrival time at each point in the circuit on the basis of the delays of the gates (or the subcircuits) only, they indeed appear to have removed the problem of finding input stimuli. However, the transition from the simulation technique to the value-independent approach has resulted in the appearance of the *false path problem*.

It is interesting to note that the false path problem is only a reincarnation of the problem of finding the right input stimuli. In the original problem, we look for the input stimuli that exercise the path with the longest propagation delay, whereas in the false path problem we have a path that exhibits the longest propagation delay, but we do not know if there exists a combination of input stimuli that exercises this path. If the path that was found by the timing verifier does not correspond to any input stimulus combination, it cannot be exercised. Such a path is called a *false path*.

Fortunately, most circuits do not cause the appearance of false paths in timing verifiers, and it is possible to sort the sources of false paths into five categories.

- *Incompatible transitions:* In this very simple case, a false path results from the combination of incompatible transitions. A typical example is the addition of the delays associated with the $1 \rightarrow 0$ transitions in subsequent inverters instead of alternating the $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions. This type of false path can easily be removed by computing separately the latest arrival time for the $1 \rightarrow 0$ and the $0 \rightarrow 1$ transitions for each node. In the upcoming discussions, we will always assume that such a separate analysis is performed.
- *Incorrect signal flow:* This type of false path appears when using pass transistor logic or barrel shifters. In these cases, as a result of the propagation conditions of the transmission gates, there is only a single conducting path from the input to the output for each combination of input values. However, many timing verifiers cannot take these propagation conditions into account and generate false paths.
- *Synchronization:* Whether the synchronization is simply performed within the clocking scheme or in a more complex way, it implies that signals are latched and have to wait for the next synchronization point. When transparent latches are used, depending on their arrival time, signals might be allowed to continue to propagate. Therefore, the propagation conditions of these latches may or may not be compatible with those of the path that lead to them. In the latter case, the ignorance of these conditions will lead some timing verifiers to incorrectly view these latches and the subsequent combinational logic as part of the path.
- *Explicit incompatibility:* Although all false paths result from some incompatibility in the propagation conditions, only the false paths in which this incompatibility is explicit appear in this category. Especially when dealing with multiplexers, explicit propagation conditions are set up. Often, if more than one multiplexer is included in a path, these propagation conditions are contradictory and false paths are generated (see Fig. 1) [10].
- *Implicit incompatibility:* In the three previous categories, we have described different types of false paths which can often be identified by the designers. In this last category, we include the false paths that designers are often unaware of. The reason for the existence of such false paths resides in the introduction of parallelism in the logic, usually in order to obtain a speedup (e.g., a CLA adder).

The presence of false paths has many undesirable effects. The first and most obvious effect is the loss of accuracy. There is no theoretical limit to the difference between the longest true path and the longest false path. Therefore, the loss of accuracy can be large and results in a loss of confidence in the timing verifier on the part of the designer. A secondary effect appears from the perspective of optimization. Since the longest propagation path determines the length of the clock period, a large difference between the longest false path and the longest true path will result in unnecessarily conservative designs, or alternatively, in wasted power and silicon area. As the real longest paths are hidden behind the longer false paths, they are not identified in the optimization. In Fig. 2, we see side by side a 12-b ALU using a carry propagation chain and a 16-b ALU using the carry by-pass technique [13]. The longest false path lies in the carry by-pass adder on the right, while the real longest path is in the propagation ALU on the left. In this example, an optimizing program will tend to increase the size of the transistors in the carry by-pass ALU and decrease the size of the transistors in the carry propagation ALU. Thus the "optimization" will decrease the circuit speed and increase the power consumption and silicon area.

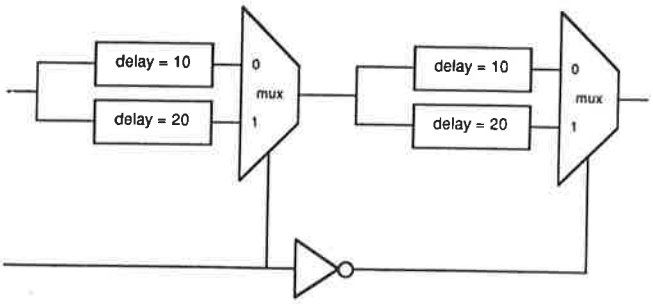

Fig. 1. Contradicting propagation conditions in two multiplexers.

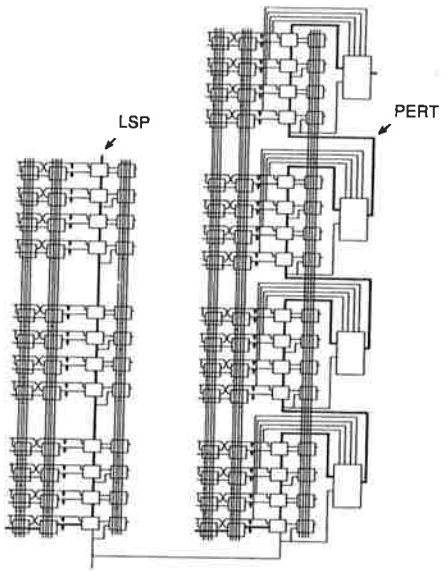

Fig. 2. The LSP and the longest PERT path in a circuit composed of a 12-b ALU and a 16-b carry by-pass ALU.

At first glance, one might think that the problem of the elimination of false paths is not so acute and that the few false longest paths can be removed easily by the designer, either manually or through *case analyses* (see Section III). Unfortunately, since false paths are made of an incorrect composition of separately true paths, they tend to be among the longest in the circuit. In addition, if there is at least one long false path in the circuit, it will be combined with all of the possible true paths and will lead to the creation of many other false paths. To understand the complexity of the problem, it is important to stress the following characteristics of false paths according to our experiences (see Section IX).

1) When false paths exist, the longest paths are usually false.

2) When false paths exist, they are numerous.

3) The number of false paths in a circuit cannot be predicted.

III. Previous Approaches

The false path problem has been known since the beginning of the development of timing verifiers. At first, false paths were considered harmless because they can only lead to conservative results; but rapidly, mostly because of the resulting lack of confidence displayed by designers, remedies were sought.

In order to deal with incorrect signal flow control, two main streams of techniques were developed: recognition techniques and tagging techniques. Recognition techniques [8] try to determine the direction of the signal flow through all of the transistors. These techniques are based upon rules that attempt to recognize common structures such as buses, pass transistor logic, storage nodes, etc. Many assumptions about the design style (e.g., ratioed logic) must be made and some of the structures still require user input to be identified. In contrast to the recognition techniques, tagging techniques rely directly on user input [11]. In the environment in which these timing verifiers work, the designer is supposed to indicate with a tag the signal flow through problematic transistors. This technique requires many runs in order to identify all of the problematic transistors.

The problems of synchronization and incompatibility of propagation conditions were dealt with in the general framework of case analyses. A case analysis is performed whenever the user determines that the logic state of a given node has to be set to a predetermined value in order to eliminate some of the false paths. For example, when dealing with a two-phase clocking scheme, a separate case analysis has to be performed for $\phi_1 = 0$, $\phi_2 = 1$ and for $\phi_1 = 1$, $\phi_2 = 0$. The same method can be used to avoid incompatible propagation conditions in pass transistor logic, multiplexers, etc. Of course, this technique implies that the incompatibilities are located and removed, and, as we have seen above, the implicit incompatibilities are often not identified. It is also important to note that those conditions are usually not propagated to the nodes that are simply validated by clocks or whose value depends upon the value of a set node. In addition, the number of case analyses grows exponentially with the number of incompatible propagation conditions.

A completely different approach was developed by Brand and Iyengar [2]. In their tool, the functional behavior of the circuit is used to automatically eliminate some of the false paths. The paths are traced from the outputs, and the logic implications associated with the paths are propagated through functional models. If an incompatibility between implications appears, the path is guaranteed to be false. Because the reverse statement is not always true, not all of the false paths are removed and the result is an upper bound on the length of the critical path. McGeer and Brayton [9] developed the concept of *viable paths*. The longest viable path is also guaranteed to be longer than the real critical path and has been proven to be a tighter bound than the one found by Brand and Iyengar. In the results reported by McGeer and Brayton, the longest viable path is almost always equal to the longest sensitizable path. If the results of both tools are equal, the path is guaranteed to be the true critical path.

Finally, as part of the first version of the SLOCOP timing verifier [15], a more primitive algorithm had been de-

veloped to eliminate false paths in a post-processing step. The PERT algorithm was modified and allowed the tool to compute the n longest paths in $O(|E|n \log n)$. Those paths, sorted by decreasing length, were then tested for static sensitizability using the D-algorithm [14] on the logic model of the circuit. Since the number of false paths to be eliminated before the first statically sensitizable path was fairly large and could not be predicted, the choice of n was difficult and required an impractical number of runs. The results obtained at that time clearly showed the problems associated with the post-processing approach.

IV. The New Approach

The approach proposed in this paper uses the logic information present in the circuit to find statically sensitizable paths, which are guaranteed to be exercisable. In order to reach this goal, it relies on both the logic behavior and timing behavior of the circuit. In our implementation the SLOCOP timing verifier provides such models, but other representations can be easily accommodated. As in earlier tools, the timing behavior is modeled by a directed weighted graph. The edges of the graph represent the propagation of a signal from one node in the circuit to another. The weight of the edge is equal to the propagation delay through the circuitry between the two nodes. Associated with each edge are a number of conditions for the propagation of the signal through the circuit. During the search for the longest statically sensitizable path, an edge can only be added to a path if it does not affect the sensitizability of the path. (i.e., its propagation conditions are compatible with the propagation conditions of all of the edges already in the path). Since this verification is computationally expensive, our approach attempts to divide it into two parts. During the search, the conditions associated with a new edge are only propagated in their neighborhood using simple rules. This propagation eliminates almost all of the nonstatically sensitizable paths. When the search finally reaches the output and produces a path, a final and complete verification of the sensitizability is performed. If the local propagation detects an incompatibility, the edge cannot be added to the path.

The problem can thus be viewed as a search for the longest path in a graph where some of the edges are incompatible, hence, the term *conditional propagation graph*. Such a graph is shown in Fig. 3, where the dotted lines represent an incompatibility between the propagation conditions of two edges.

Searching for the longest path in such a conditional propagation graph is also a difficult problem. In the worst case, the number of longest path searches is exponential in the number of incompatibilities in the graph, yielding an equivalent graph without incompatibilities counting $2^n|E|$, where $|E|$ is the number of edges in the original graph and n is the number of incompatible edges. It is thus imperative to reduce the size of the search space as much as possible. The algorithms that will be described in the following attempt to curb the complexity of the search in the average case by using simple heuristics to guide the search. When possible, bounds on the length of the longest statically sensitizable path in the graph are continuously updated, which allow us to prune subgraphs that cannot lead to the solution.

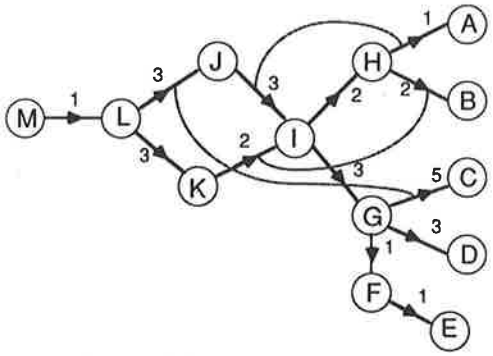

Fig. 3. A conditional signal propagation graph.

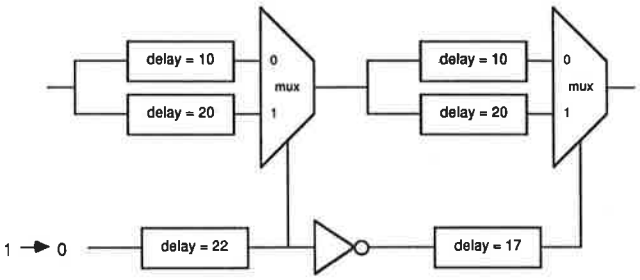

Fig. 4. Contradicting propagation conditions with delays.

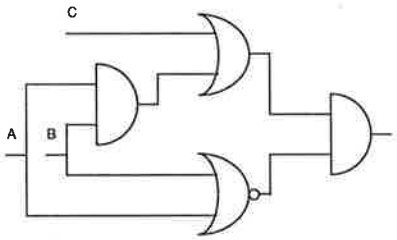

Fig. 5. Critical path resulting from multiple input changes.

The underlying assumption that was made is that the logic propagation conditions must be present at all the nodes for all the time it takes for the signal to propagate through the path. This implies that the path can be activated in isolation of all the other paths, with all its side-inputs held at a constant value. Such a path is said to be *statically sensitizable*, and differs from a *dynamically sensitizable* path where at least one of the side-inputs must be switched in order to allow the signal to propagate along the path. A path can be dynamically sensitizable without being statically sensitizable. In the case shown in Fig. 4, the signal through the 20-ns delay can traverse the first multiplexer because the logic value on the control line is still 1. Then the signal can go through the second 20-ns delay and traverse the second multiplexer since the logic value on its control line has changed to 1 before the signal arrives. Fig. 5 shows a circuit taken from [2] in which the length of the longest statically sensitizable path is 2, even though the simultaneous switching of A and B can result in a path of length 3.

Our approach searches for the longest statically sensitizable path and thus yields a lower bound on the true critical path. It can be used in conjunction with the PERT longest path and other upper bounds to estimate the length of the true critical path. Dynamically sensitizable paths can disappear if the delays of their side-inputs are changed. In Fig. 4, if the delay before the inverter is changed to 19 ns, the path will disappear. Although this change will affect the longest PERT path, it implies that a path which required optimization has been removed. Because statically sensitizable paths do not rely on the interaction between signals, they will vary in length only and will not disappear.

V. The SLOCOP Environment

The SLOCOP timing verifier is targeted towards digital MOS circuits and is characterized by the modeling of both logic and timing behaviors. In the first processing step, the transistor network is partitioned into unidirectional subcircuits. The partitioning is based upon pattern-matching techniques which identify classes of subnetworks. These classes of subnetworks are easily described through the use of the LEXTOC language [3]. Using this language, one can describe the possible transistor connectivity patterns of a design style (e.g., domino CMOS). The resulting subcircuits are always unidirectional and are associated with a logic expression in terms of four operators: NOT, AND, OR, and SEL. Using these operators, the logic behavior of all of the subcircuits can be modeled. Their formal definitions are the following.

- $y = \text{NOT}(x)$: the output is the Boolean negation of the input.
- $y = \text{AND}(x_1\ x_2\ \cdots\ x_N)$: the output is the Boolean AND of the inputs.
- $y = \text{OR}(x_1\ x_2\ \cdots\ x_N)$: the output is the Boolean OR of the inputs.
- $y = \text{SEL}(d_1\ s_1\ d_2\ s_2\ \cdots\ d_N\ s_N)$: this operator models an ideal level sensitive latch with N select and data inputs. Its behavior is given by:

1) IF $s_i = 0$ for all i THEN the output is unrelated to the data inputs;
2) IF $s_i = 1$ for any i THEN $y = d_i$;
3) not more than one select signal can be high.

For a subcircuit with N inputs, $N \cdot 2^N$ excitations exist in which one input makes a transition between the two logic levels (denoted as $0 \rightarrow 1$ or $1 \rightarrow 0$) and the rest of the inputs are either 0 or 1. However, only a few of these combinations cause the input transition to propagate to the output. The logic expression of the subcircuit allows the determination of these transitions and the corresponding conditions on the values of the other inputs. Such a transition corresponds to an edge in the conditional propagation graph of the circuit, as depicted in Fig. 6. As a result of this partitioning technique, all of the false paths resulting from incorrect signal flow are eliminated.

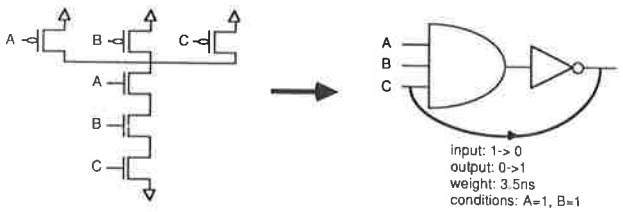

Fig. 6. SLOCOP logic and timing model (partial) for a NAND gate.

After the logic model is obtained, the timing model (i.e., the weight of the edge in Fig. 6) is extracted from an analog simulation. The propagation conditions that were just described also allow us to automatically generate the input excitations for this analog simulation. The delay is defined as the interval between the time points where the input and output cross the 2.5-V level. It has been shown [1], [15], that such a model of the timing behavior of entire subcircuits is much more accurate than a stage-based model [11]. Comparison with a traditional simulation of the critical path, in a number of circuits made up of conventional complementary CMOS gates, yielded errors on the order of 10% of the total propagation delay.

Although other methods of obtaining the logic and timing models can be accommodated, the SLOCOP environment provides excellent accuracy in the timing modeling, eliminates the local false paths resulting from incorrect signal flow, and builds an equivalent model of the logic behavior.

VI. Verification of Static Sensitizability

As the longest path search is conducted, edges are added to the partial path that is currently tracked. In order to eliminate all of the false paths, it is necessary to check that the addition of these new edges does not endanger the static sensitizability of the path. The detection of the possible incompatibilities between the logic propagation conditions associated with the delays is performed by an algorithm that we have developed on the basis of the D-algorithm [14] by removing its justification step. The goal of this algorithm is to find all of the local logic implications of the propagation conditions of the edges included in the partial path. The required logic values are recursively propagated throughout the network, on the basis of the logic model provided by the SLOCOP environment and using the following set of rules.

- $y = \text{NOT}(x)$:
  1) IF x is known, set y to $\bar{x}$.
  2) IF y is known, set x to $\bar{y}$.
- $y = \text{AND}(x_1, x_2, \cdots, x_N)$:
  1) IF $x_i = 1$ for all i, set y to 1.
  2) IF $y = 1$, set x_i to 1 for all i.
  3) IF $x_i = 0$ for any i, set y to 0.
  4) IF x_j is unknown and, $x_i = 1$ for all $i \neq j$ and, $y = 0$, set x_j to 0.

- $y = \text{OR}(x_1, x_2, \cdots, x_N)$:
  1) IF $x_i = 0$ for all i, set y to 0.
  2) IF $y = 0$, set x_i to 0 for all i.
  3) IF $x_i = 1$ for any i, set y to 1.
  4) IF x_j is unknown and, $x_i = 0$ for all $i \neq j$ and, $y = 1$, set x_j to 1.
- $y = \text{SEL}(s_1, d_1, s_2, d_2, \cdots, s_N, d_N)$:

IF $s_i = 1$ for any i:
  1) set s_j to 0 for all $j \neq i$.
  2) IF d_i is known, set y to d_i.
  3) IF y is known, set d_i to y.

Using this method, logic values that are necessary for the static sensitizability of the partial path can be determined. Logic incompatibilities between the propagation conditions are detected when opposite logic values are required at the same node. In addition, this technique can be implemented to allow incremental verification: all of the assignments of values associated with the addition of a new edge are recorded and can easily be removed in case of incompatibility or in future backtracking.

When the search algorithms find a path that does not contain any local incompatibility, the justification step of the D-algorithm must be performed to ensure the static sensitizability of the path. Fig. 7 shows an example of a path that appears to be sensitizable after the local propagation of the implications, but fails the final D-algorithm. Our implementation is based upon a queue of gates requiring justification, and a stack of all of the decisions made so far. The following algorithm takes the network, including the local implications of the logic propagation conditions, and completes the D-algorithm.

Final D-algorithm

Notation:

Q: a queue of gates requiring justification.
S: a decision stack.

Let g be the "active gate."

The algorithm is initialized with: Q = all the gates in the network which require justification; $S = 0$.

1. **IF** $Q = 0$ **THEN** path is statically sensitizable
2. **ELSE**
   (a) let g be the first gate in Q
   (b) let i be the first unexplored input of g
   (c) set i to the required controlling value v
   (d) push decision (g, i, v) onto S
   (e) **IF** the implications of (g, i, v) do not result in any incompatibility
   THEN add new gates requiring justification to Q and goto 1
   (f) **ELSE**
      i. pop top decision (g, i, v) and associated implications from S
      ii. **IF** there are still unexplored inputs in g **THEN** goto 2.b

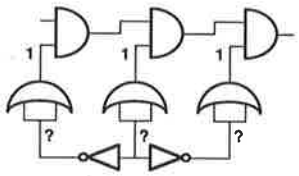

Fig. 7. The need for the *final D-algorithm*.

   iii. **ELSE**
      A. **IF** $S = 0$ **THEN** path fails
      B. **ELSE** goto 2.f.i.

VII. Guiding the Search

When searching for the longest path in a graph without incompatibilities, it is possible to rely on greedy algorithms to find the solution. The reason is simply that the result is dependent only upon the weight of the edges. If the longest paths to all of the nodes that are connected to a certain node are known, one can immediately compute the longest path to that node. Therefore, the weights on the edges can be considered as local constraints. The compatibility constraints, on the other hand, can be viewed as global constraints. Indeed, as shown in Fig. 8, the longest path to a given node can no longer be computed on the sole basis of the longest paths to the preceding nodes. In this figure, we see that the length of the longest path from A to B is larger than the distance from A to B along the longest statically sensitizable path in the graph $(ACBD)$. These global constraints thus forbid the use of greedy algorithms, and one would have to resort to exhaustive search or path enumeration techniques in order to guarantee a solution. Since exhaustive search is an unrealistic solution, we have developed algorithms that guide the search in the most promising directions computed on the basis of local constraints, and backtrack whenever global constraints cannot be met.

Since it is impossible to find the direction that will lead directly to the optimal solution in a reasonable time, one would like to gather some information regarding the "good" search directions. At this point, a compromise has to be made between the time spent gathering this information and the quality of the information. As the number of incompatible edges is relatively small, we have assumed that the direction of the longest path, even if it is false, it still constitutes a good indication as to where long statically sensitizable paths lie. In addition, this heuristic has several important advantages.

- *Low computation cost:* The length of the longest path from each node to any output can be computed by the PERT algorithm in $O(|E|)$.
- *Path invariance:* The longest path from each node to the output is constant, regardless of the partial path that lead to the node. Therefore, it must be computed only once.
- *Conservative estimate:* Since the PERT algorithm computes the longest path regardless of the logic incompatibilities, it is always longer than the longest statically sensitizable path.

The low computation cost and the invariance of the result make the PERT algorithm an excellent candidate for

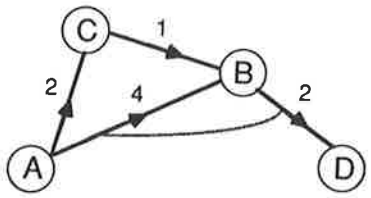

Fig. 8. Global versus local constraints in a conditional propagation graph.

inexpensive heuristics. As we will see in the next sections, the third property is even more important in reducing the complexity of the search. In the remainder of this paper, we will use the term *PERT path* to refer to a path that has been computed with the PERT algorithm and which is, therefore, not necessarily statically sensitizable.

Since the length of the longest PERT path is always an overestimate of the length of the longest statically sensitizable path, it can serve as an upper bound. As the search proceeds, partial paths are built. We will refer to the length of the longest statically sensitizable path that can still be found with a given partial path as its *esperance*. *The esperance of a partial path will be computed as the sum of its length and the length of the longest PERT path from its last node to the output.* In addition, the algorithms keep track of the length of the longest statically sensitizable path that was found so far. Obviously, this value constitutes a lower bound to the length of the longest statically sensitizable path in the graph.

Using information already gathered about the graph, we try to eliminate the need to search large parts of the graph. Before the search enters any new subgraph, the length of the longest path found so far is compared with the esperance of the subgraph. This esperance constitutes an upper bound on the length of all of the possible statically sensitizable paths that we could find in the subgraph. Therefore, if the esperance is found to be lower than the length of a known statically sensitizable path, it is sure that the solution does not lie in that subgraph and this search direction may be pruned. This method allows drastic reduction of the complexity of the search in the average case.

VIII. The Algorithms

According to the principles described in the previous section, we have developed two new algorithms. The first algorithm is a modified version of the *depth-first search* (DFS) algorithm [5]. This algorithm performs a depth-first search using the esperance of the subgraphs to choose the order in which they will be visited. It backtracks to the last explored node whenever an incompatibility is encountered and modifies the esperance of the partial path accordingly. In addition, only subgraphs whose esperance is larger than the length of the longest known statically sensitizable path are searched. The algorithm also continuously updates an upper bound and a lower bound on the length of the longest statically sensitizable path in the graph. The lower bound is the length of the longest known statically sensitizable path and the upper bound is the esperance of the root.[1]

[1]The root is a special node which is connected to all of the inputs of the circuit.

The Modified DFS with Pruning

Notation:

- $o(i)$: The length of the longest PERT path from node i to the output.
- $l(X)$: The length of the path X.
- $E(X)$: The *esperance* of the path X.
- L: The length of the longest known statically sensitizable path.

Let n be the "active node," and P the "active path" from the root to n.

The algorithm is initialized with $n = root$, $P = \{root\}$, $L = l(\{root\}) = 0$.

1. Among all of the unexplored nodes connected to n, find the node i whose $o(i)$ is maximum and whose addition to P does not cause any local incompatibility. Mark all the nodes that caused a violation, and mark i as having been explored. Add node i to P to form a new path Pi and compute $E(Pi) = l(Pi) + o(i)$.
2. **IF** $E(Pi) > L$
   THEN (forward)
   (a) mark n as father of i
   (b) if $l(Pi) > L$ and i is an output, then if Pi successfully passes the *final D-algorithm then* $L = l(Pi)$
   (c) $P = Pi$
   (d) $n = i$
   ELSE (backward)
   (a) if $n = root$ then exit
   (b) let j be the father of n
   (c) erase exploration marks to all the sons of n
   (d) remove n from P
   (e) $n = j$.

Tables I and II show the result of the PERT analysis and the application of the modified DFS algorithm to the graph appearing in Fig. 3. Notice that neither the $\{A, B, H\}$ nor the $\{F, E\}$ subgraphs are searched. This algorithm can be modified to search for all the paths above a given threshold L' by replacing L with L' in 2(b) and printing the path instead of assigning $l(Pi)$ to L.

The principal advantage of the modified DFS algorithm lies in its low memory requirements which are proportional to $|E|$ as well as the availability of a lower bound. It is important to understand the local nature of this algorithm which, when an incompatibility is encountered, backtracks only to the previously explored node. Our second algorithm, on the contrary, has a more global view of the graph, but involves exponential memory requirements. In the same way as the A* algorithm [16] on which it is based, it keeps track of all of the esperances of the alternative partial paths. Therefore, as soon as the esperance of the current partial path decreases because of an incompatibility, it can switch to a more promising partial path. The A* algorithm was primarily intended to search for the shortest path in the presence of underestimates.

TABLE I
RESULTS OF THE PERT ANALYSIS

i	A	B	C	D	E	F	G	H	I	J	K	L	M
$o(i)$	0	0	0	0	0	1	5	2	8	11	10	14	15

TABLE II
APPLICATION OF THE *MODIFIED DFS WITH PRUNING* ALGORITHM

n	$l(P)$	L	max $E(Pi)$	next move
M	0	0	15	forward
L	1	1	15	forward
J	4	4	15	forward
I	7	7	15	forward
G	10	10	13	forward
D	13	13	13	backward
G	10	13	12	backward
I	7	13	11	backward
J	4	13	4	backward
L	1	13	14	forward
K	4	13	14	forward
I	6	13	14	forward
G	9	13	14	forward
C	14	14	14	backward
G	9	14	12	backward
I	6	14	10	backward
K	4	14	4	backward
L	1	14	1	backward
M	0	14	0	exit

We have modified it to search for the longest path using the length of the longest PERT path as an overestimate. No lower bound is available since the algorithm completes when an output is reached. An upper bound on the length of the longest statically sensitizable path in the graph is continuously updated. Since this algorithm knows about all of the alternative partial paths, it uses the largest of their esperances as an upper bound. While the esperance of the root in the modified DFS requires that the algorithm backtracks to the root to be updated, this upperbound is tighter as it is modified each time an incompatibility is encountered.

*The Modified A**

Notation:

$o(i)$: The length of the longest PERT path from node i to the output.
$l(X)$: The length of the path X.
$E(X)$: The *esperance* of the path X.

Let Q be an ordered queue of the current paths and their *esperances* $(S, E(S))$. The queue is sorted by decreasing *esperance*. The *esperance* $E(S)$ of a path S is computed as $E(S) = l(S) + o(i)$, i being the last node in S.

The algorithm is initialized with $Q = \{(root, E(root)\}$.

1. Take the head of the queue $(P, E(P))$ and delete it from the queue.
2. If P reaches a primary output, then if P successfully passes the *final D-algorithm*, exit.
3. For each node j connected to i (the last node of P) which does not cause any local incompatibility with any of the edges already in P:

TABLE III
APPLICATION OF THE *MODIFIED A** ALGORITHM

Q
(M,15)
(ML,15)
(MLJ,15),(MLK,14)
(MLJI,15),(MLK,14)
(MLJIG,15),(MLK,14),(MLJIH,11)
(MLK,14),(MLJIGD,13),(MLJIGF,12),(MLJIH,11)
(MLKI,14),(MLJIGD,13),(MLJIGF,12),(MLJIH,11)
(MLKIG,14),(MLJIGD,13),(MLJIGF,12),(MLJIH,11),(MLKIH,10)
(MLKIGC,14),(MLJIGD,13),(MLJIGF,12),(MLKIGD,12),(MLJIH,11), (MLKIGF,11),(MLKIH,10)
EXIT

(a) form the new path P_j and compute its *esperance* $E(P_j) = l(P_j) + o(j)$.
(b) add the new path in the ordered queue.

Table III shows the application of this algorithm to the graph appearing in Fig. 3. Notice that here again neither the $\{A, B, H\}$ nor the $\{F, E\}$ subgraphs are searched.

The above algorithm searches only for the longest statically sensitizable path. It can be easily modified in two ways to produce a different result.

- If 2 is modified to print the path instead of exiting, this algorithm yields an ordered list of all of the statically sensitizable paths in the network. The algorithm completes when the queue is emptied.
- The same modification can also allow one to search all the paths above a given threshold value T. In this case, the algorithm completes when the esperance of the head of the queue is smaller than T.

IX. IMPLEMENTATION AND RESULTS

The algorithms presented above have been implemented in C under UNIX and have been linked to the existing SLOCOP environment. The CPU times in the following tables are for a VAXstation 3200. The results of experiments conducted on several practical test cases appear in Table IV. In Fig. 9, we have plotted the longest statically sensitizable path (LSP) and the longest PERT path in ALU's with different wordlengths. It should be noted that the PERT analysis is performed on a graph setup by SLOCOP which already eliminates the false paths due to incorrect signal flow and opposite transitions. Fig. 10 shows the LSP in a circuit with explicit incompatibility similiar to the one in Fig. 1.

The following observations can be made.

- For carefully optimized circuits, such as the ALU's which contain carry by-pass circuitry, the difference between the longest path found by the PERT algorithm and the longest statically sensitizable path can be fairly large. In contrast, we included a multiplier implemented as a simple array of full adders. In this case, the lack of optimization results in the presence of many paths of similar length.
- With the exception of the third example in Table IV, the final D-algorithm did not cause the elimination of the path found to be "locally" sensitizable, and the difference in computation cost for the algorithms with and without this step is small ($< 10\%$). In the third example, the LSP without the final D-algorithm was

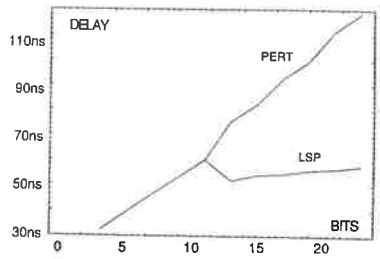

Fig. 9. LSP and longest PERT path versus wordlength in ALU's.

TABLE IV
COMPARISON BETWEEN THE PERT CRITICAL PATH AND THE LSP

circuit	transistors	PERT	LSP	Mod-DFS	Mod-A*
combinatorial logic	46	24.1ns	19.6ns	0.4s	1.0s
8 bit comparator	173	19.3ns	18.7ns	0.7s	1.9s
10 bit counter	260	29.4ns	11.4ns	5.6s	22.2s
7 bit carry-sel add	210	29.8ns	29.8ns	0.1s	0.2s
8x8 multiplier	1840	226.1ns	223.6ns	54.3s	217.2s
12 bit by-pass ALU	720	34.1ns	28.6ns	8.1s	29.2s
12+16 bit ALU (Fig. 3)	1760	108.3ns	77.4ns	30.2s	57.5s
selectors (Fig. 10)	2430	129.8ns	104.9ns	1379.3s	NA
4 bit by-pass	280	32.4ns	32.4ns	0.1s	0.8s
14 bit by-pass	910	75.4ns	51.9ns	723.5s	2123.7s
18 bit by-pass	1170	92.8ns	54.5ns	1620.0s	NA
24 bit by-pass	1680	118.5ns	57.8ns	4012.4s	NA

28.7 ns, and the CPU times for the modified DFS and modified A* were 0.9 and 1.3 s, respectively. To obtain the correct result, a total of 269 paths were discarded, 43 in the final D-algorithm.

- The fact that the difference between the LSP and the PERT critical path can be large shows that a lower bound can be useful. Whenever the difference between the LSP and the PERT critical path is large, the user knows that both results should be taken carefully, but at least he is able to bound the length of the critical path. On the other hand, when the difference is small or nonexistent, the user knows that he can trust the result.
- The search time is strongly related to the gain in accuracy that is obtained. If the difference between the PERT critical path and the longest statically sensitizable path is small, the search terminates quickly.
- The modified DFS algorithm consistently outperforms the modified A* algorithm. This unexpected result is analyzed in detail in the next section.

To carefully analyze these results and to gain some insight into the distribution of the paths as well as into the respective contributions of the local propagation of implications and of the final D-algorithm, we have implemented a special version of the modified A* algorithm. In this version, the local propagation is not performed during the search. Only when a path reaches the primary output is the local propagation performed, followed immediately by the final D-algorithm. As Table V shows and as was also found by [4], the ratio of regular false paths (RFP's), i.e., paths that are found to be nonsensitizable by the local propagation, to statically sensitizable paths (SSP's) is large. We also found that the number of paths that fail the final D-algorithm (DFP's) is small. More importantly, the number of RFP's to be eliminated before the first SSP is found is significant. These data explain the small difference in CPU requirements between the algorithms with and without the final D-algorithm.

TABLE V
PATH DISTRIBUTIONS

circuit	total # of paths	# of RFPs before LSP	# of SSPs	# of DFPs
combinatorial logic	158	84	24	2
selectors (Fig. 10)	112	56	32	0
example in Fig. 5	10	4	6	0

X. PERFORMANCE ANALYSIS

In our presentation of the two algorithms in Section VIII, we stressed the advantage of the more global view that characterizes the modified A* algorithm. When we developed those algorithms, our expectations were that the modified A* would easily outperform the modified DFS, except for circuits where the large memory requirements of the modified A* would slow it down or even prevent completion. The modified DFS is actually faster in all of the examples that were tried. The example appearing in Fig. 2 was definitely the most surprising as we expected the modified DFS to wander in the by-pass ALU for a long time while the modified A* would switch to the carry propagation ALU earlier. Although part of this result is due to a longer initialization phase in the modified A* algorithm, the larger examples point to more basic advantages of the modified DFS algorithm.

The first major difference between the two algorithms is again a matter of global versus local view. Since the modified DFS backtracks only to the previously explored node, it is possible to save in a stack all of the logic implications that resulted from this exploration. When backtracking, all of the implications are simply popped from the stack and the previous situation is restored. Usually, very few nodes are affected and this operation does not require much CPU time. The modified A* algorithm does not backtrack to a close node but switches to a completely different path. This change implies that all of the logic implications related to the previous path must be removed, and all of the logic implications related to the new partial path must be installed. It appears that this operation takes about 40% of the total CPU time.

The second difference between the algorithms is more subtle. The modified A* algorithm switches to another path each time an incompatibility causes the esperance of the current path to fall below the esperance of any alternative partial path. Our experiments have proven that the density of the path lengths is high, which implies that such a switch occurs almost every time an incompatibility is encountered. The modified DFS, on the contrary, tries to find a long path in the region of the graph currently searched before switching to a different part of the graph. The result is that the lower bound increases rapidly and allows pruning of large portions of the graph as soon as an incompatibility is encountered. In Fig. 11, which

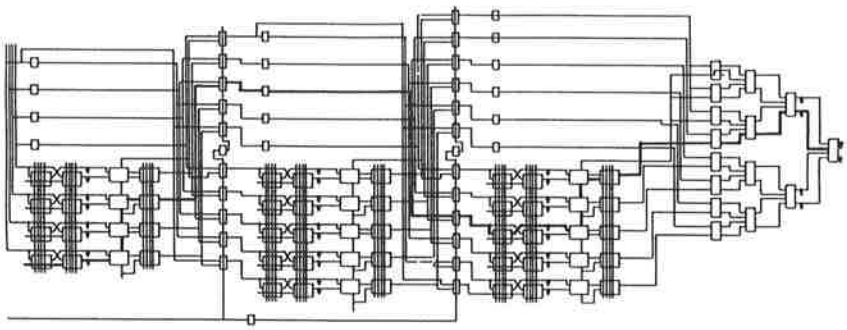

Fig. 10. LSP in presence of an explicit incompatibility.

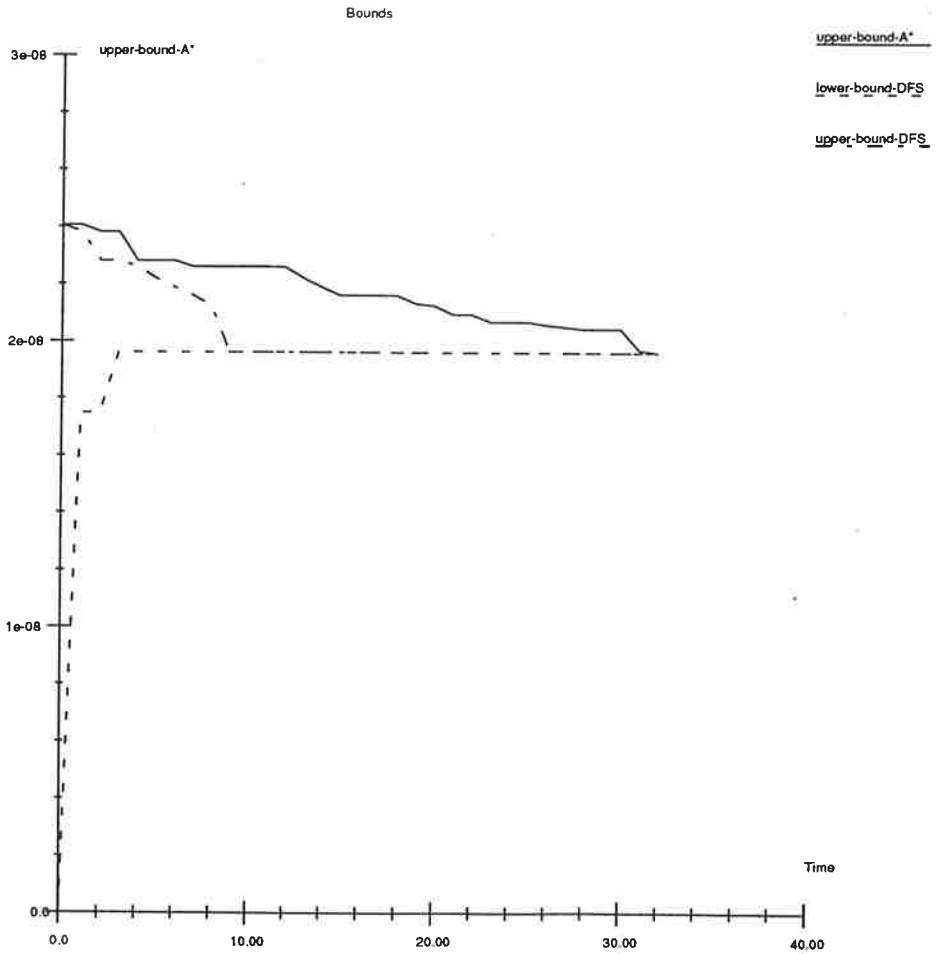

Fig. 11. Convergence of the bounds in the *modified DFS with pruning* and modified A* algorithms.

shows the comparison between the progression of the lower and upper bounds of the two algorithms, it can clearly be seen that the modified A* upper bound decreases in small steps while the modified DFS lower bound quickly rises and then causes the upper bound to drop much faster.

As already mentioned previously, the time spent searching for the longest statically sensitizable path is in direct relation to the difference between the length of this path and the length of the longest PERT path. The reason for this is quite obvious since the PERT analysis guides our algorithms. If there is no false path, neither of the algorithms needs to backtrack and the search is trivial. In more difficult cases, we can look at the time spent backtracking as related to the quality of the information obtained from the PERT analysis. The larger the difference between the statically sensitizable and the nonsensitizable longest path, the more times the information given by the esperance will be misleading and the longer the search will wander. This relationship is unfortunately not constant. We have observed a large difference between the CPU time spent on the 12-b ALU with by-pass circuitry

TABLE VI
THE PRUNING TECHNIQUE GAIN

circuit	# of nodes w/pruning	CPU	# of nodes w/o pruning	CPU	false
4 bit by-pass ALU	14	0.1s	176177	2723.4s	no
8 bit comparator	25	0.5s	1756	21.7s	yes
8 bit counter	26	0.3s	1147	19.1s	yes
7 bit carry-sel add	14	0.1s	3018	26.8s	no
combinatorial logic	73	0.4s	132	0.8s	yes
12 bit by-pass ALU	1144	6.9s	42351	741.3s	yes

starting at the LSB and a 14-b ALU in which the by-pass circuitry was introduced after the 4th bit. The reason for this difference lies in the effectiveness of the pruning technique. In the 12-b ALU, after a long path has been found, the remaining false paths can be discarded as soon as the incompatibility located in the first by-pass circuitry is discovered. In the 14-b ALU, however, the algorithm has to reach the 4th bit before false paths can be discarded. Not only is the search going deeper but the number of paths in the first 4 bits is already large and requires many backtrackings.

The final D-algorithm step also demonstrates this gain versus the CPU relationship. If the first path passes the final D-algorithm, little time is spent. In the third example of Table IV, the final D-algorithm must eliminate several paths, which results in a much larger difference in computation time. In both Tables IV and V, it is important to note that the local propagation of the implications eliminates almost all of the paths which are not statically sensitizable. This suggests that the dissociation of the two parts of the D-algorithm provides a better result in terms of computation cost than if a full D-algorithm was performed during the search.

Finally, in order to evaluate the effectiveness of our pruning technique, we suppressed pruning and compared the results with the original algorithm. Table VI shows the CPU time and the number of nodes visited by the two versions while the last column indicates whether there was a false path in the circuit. In the absence of false paths, the advantage can be expected to be large, but we see that even when a relatively long search is required, the pruning technique still provides a tremendous gain.

XI. CONCLUSION

The importance of the false path problem in timing verifiers cannot be neglected. The large errors that have been proven to occur in previous timing verifiers prevent their application in optimization, where an accurate estimate of timing constraints is required. However, the need to search for the input stimuli is inherent to the problem of finding the critical path, and the cost of solving the false path problem is intrinsically high. In order to alleviate this situation, we have proposed a new approach that takes into account the logic behavior of the circuit and finds paths which are guaranteed to be sensitizable. Using simple logic models, we have developed new algorithms to search for statically sensitizable paths. We have converted the problem of searching for statically sensitizable paths into a simple longest path search associated with a local propagation of the logic implications, followed by a complete D-algorithm. The algorithms that we have developed reduce the computation cost associated with this task. By continuously updating bounds on the length of the longest path and progressively refining the search space, the new algorithms manage to curb the combinatorial explosion of the search. The search itself is guided by a preanalysis based upon the PERT algorithm, which allows pruning of subgraphs that cannot improve the bounds. In addition, the bounds provide the user with feedback on the progress of the search and permit him to interrupt the search when a satisfactory estimate is reached.

Because of the difficulty of finding the true critical path of a circuit, designers have traditionally been skeptical of the results produced by timing verifiers. By using the lower bound provided by the longest statically sensitizable path and the upper bound given by the longest PERT path or other algorithms, a much higher confidence can now be placed in timing verification.

ACKNOWLEDGMENT

The authors would like to acknowledge the anonymous reviewers of the first version of this paper for their useful comments, especially for stressing the problem of dynamic versus static sensitization. The authors are also grateful to Marko P. Chew, Tom Cobourn, Larry Pillage, and Prof. A. J. Strojwas from CMU for many stimulating discussions and reviews that helped refine this paper.

REFERENCES

[1] J. Benkoski and A. J. Strojwas, "A new approach to statistical and hierarchical timing simulations," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1039–1052, Nov. 1987.

[2] D. Brand and V. S. Iyengar, "Timing analysis using functional relationship," *IEEE ICCAD '86, Dig. of Tech. Papers*, Santa Clara, CA, Nov. 1986, pp. 126–130.

[3] H. De Man, I. Bolsens, E. Vanden Meersch, and J. Van Cleynenbreughel, "DIALOG, An expert debugging system for MOSVLSI design," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 303–311, June 1985.

[4] D. H. C. Du, S. H. C. Yen, and S. Ghanta, "On the general false path problem in timing analysis," *Proc. 26th Design Automation Conf.*, 1989.

[5] S. Even, *Graph Algorithms*. New York: Computer Science, 1979.

[6] R. B. Hitchcock, G. L. Smith, and D. D. Cheng, "Timing analysis of computer hardware," *IBM J. Res. Develop.*, vol. 26, no. 1, pp. 100–105, Jan. 1982.

[7] N. P. Jouppi, "TV: An nMOS timing analyzer," *Proc. Third Caltech VLSI Conf.*, 1983, pp. 72–85.

[8] —, "Derivation of signal flow direction in MOS VLSI," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 480–490, May 1987.

[9] P. C. McGeer and R. K. Brayton, "Efficient algorithms for computing the longest viable path in a combinational network," *Proc. 26th Design Automation Conf.*, 1989.

[10] T. M. McWilliams, "Verification of timing constraints on large digital systems," in *Proc. 17th Design Automation Conf.*, 1980, pp. 139–147.

[11] J. K. Ousterhout, "A switch-level timing verifier for digital MOS VLSI," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 336–348, June 1985.

[12] S. Perremans, L. Claesen, and H. De Man, "Static timing analysis of dynamically sensitizable paths," *Proc. 26th Design Automation Conf.*, 1989.

[13] M. Pomper, W. Beifuss, K. Horninger, and W. Kaschte, "A 32-bit execution unit in an advanced NMOS technology," *IEEE J. Solid-State Circuits*, vol. SC-17, pp. 533-538, June 1982.
[14] J. P. Roth, "Diagnosis of automata failures: A calculus and a new method," *IBM J. Res. Develop.*, pp. 278-281, Oct. 1966.
[15] E. Vanden Meersch, L. Claesen, and H. De Man, "SLOCOP: A timing verification tool for synchronous CMOS logic," *in Proc. ESSCIRC '86*, Delft, 1986, pp. 205-207.
[16] P. H. Winston, *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1984.

*

Jacques Benkoski (S'86-M'89) received the B.Sc. degree (cum laude) in electrical and computer engineering from the Technion-Israel Institute of Technology, Haifa, Israel, in 1985, and the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University in 1987 and 1989, respectively.

During 1985, he worked at the IBM Scientific Center, Haifa, Israel. In 1987, he was a visiting research scientist at the Interuniversity Micro-Electronics Center (IMEC) in Leuven, Belgium. Since 1989, he has been with SGS-Thomson, Central Research and Development in Grenoble, France. His research interests include timing modeling and simulation, design verification methodologies, and performance-driven synthesis.

Erik Vanden Meersch received the degree in electrical engineering from the University of Leuven, Belgium, in 1983.

He joined the ESAT CAD team, where he worked on algorithms for design analysis, including topology check and timing verification with the goal of integrating these different tools into an interactive design system. In 1985 he joined the IMEC Laboratory, where he continued this research. In 1987, he joined CPqD Telebras, Campinas, Brazil where he worked on new algorithms for transistor netlist comparison. Since 1989, he has been with the CAD Department, MIETEC, Brussels, Belgium.

*

Luc Claesen (S'77-M'85) received the Electrical Engineering degree and the Ph.D. degree from Katholieke Universiteit Leuven, Belgium, in 1979 and 1984, respectively.

In 1979, he joined the ESAT Laboratory, Katholieke Universiteit Leuven, as a Research Assistant, where he worked in the field of computer-aided design of integrated systems for digital and analog signal processing. In 1984, he joined the IMEC Laboratory, where he headed research in the Design Management and Verification group, VLSI Systems Design Methodologies Division. Currently, he is an Associate Professor at Katholieke Universiteit Leuven. His research interests include computer-aided design, formal design and verification methods, integrated digital signal processing, and image synthesis systems.

Dr. Claesen received a Best Paper Award at the ICCD 1986 Conference. He is a member of IFIP working group 10.2.

*

Hugo De Man (M'81-SM'81-F'86) for a photograph and biography, please see page 937 of the September 1990 issue of this TRANSACTIONS.