SLOCOP_ II :A versatile timing verification system for MOSVLSI.*

P. Johannes, P. Das, L. Claesen, H. De Man †

IMEC, Kapeldreef 75, B-3030 Leuven (Belgium)

Abstract

The new SLOCOP-II timing verification system for the accurate performance analysis of MOSVLSI circuits is being presented. The algorithms in SLOCOP-II solve the serious problem of "false paths" that occur in all existing timing verifiers, by taking into account the logic functionallity of the circuits at hand. To allow this for custom MOSVLSI designs, new event determination algorithms based on binary decision tree (BDT) have been developed and are presented in this paper. The algorithms to avoid the indication of "false longest delay paths" can take a long calculation time. Therefore two new techniques have been developed: 1) by preprocessing the constrained event graph, compiled code can be generated that can execute orders of magnitude faster and 2) by exploiting the hierarchy available in circuits. These algorithms have been implemented in the SLOCOP-II timing verification system. Results and comparative cpu times on parameterised modules in the CATHEDRAL-II library are presented in the paper.

Key Words: Static timing verification

1 Introduction.

The fact that integrated circuit designs must be verified for their correctness before they are produced is generally accepted. Simulation on all levels is still used to a large extent for this verification. However, analytic methods such as timing verification tools [1] for the verification of the delay characteristics are more and more used. Timing analyzers determine the critical delay paths, time slacks and violations against set-up and hold times. This is done in a way that is independent of specific logical values of the signals. Timing verifiers specifically adapted for MOS circuits have been published [2,3,5]. Alternative methods have been proposed in order to increase the accuracy of the delays [4,6]. However, existing timing verifiers [3,2,5] do not take logical propagations conditions into account and compute the longest delay paths with PERT like algorithms. This may cause problems, as can be seen in figure 1. The longest path in the circuit on the left can not be sensitised: the '1' on both circuit inputs make the '0' on the multiplexer input impossible and thus the output is not sensible to transitions on the bold input. The real longest path is given on the right. This situation

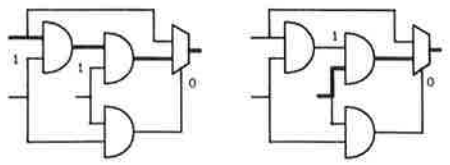

Figure 1: A circuit with an unsensitisable path

occurs in time optimised circuits, such as the carry bypass ALU [9] in the CATHEDRAL-II library [13] where the delay is overestimated by a factor of 2.

*Research sponsored under the EEC ESPRIT-1058 project.

†Professor at Katholieke Universiteit Leuven Belgium

Algorithms that assist in solving the false path problems have been proposed in [8,11,12]. These new algorithms require modeling the logic functionality and the event graphs for the subcircuits. This can be accomplished by using characterised library cells [11,12] or by using rule-based pattern matching techniques [6]. However, these approaches are limited to either the cell libraries, or to the circuit patterns described in the rule-base and are not applicable to general MOSVLSI circuits, without each time extending the rule base and/or the library to new circuit patterns and design styles. Traditional timing verifiers [3,2,5] are not directly restricted to specific circuit patterns or design styles but do not allow to model the logic functionalities and input-output event dependencies as required by the new algorithms [8,11,12] for false path avoidance. New event and propagation condition determination algorithms for general MOSSVLSI circuits are presented in this paper.

Unfortunately, the application of these algorithms on MOSVLSI circuits is very time consuming. In order to accelerate the evaluation of the longest delay paths, the concept of code generation has been introduced [5]. The calculation of the delay is deferred to when the code is being evaluated. This can be done by implementing the delay evaluation by means of abstract data-types [7].

Another way of increasing the speed is to introduce hierarchy. This avoids the problem of long preprocessing times that occur when handling large circuits with the code generation method.

In section 2 the SLOCOP timing verification environment is introduced. In section 3 the event graph with logical signal propagation constraints will be explained, together with the methods that are used to extract the propagation conditions from the logic functions. Section 4 describes the algorithms to determine the longest sensitizable path. Thereafter section 5 describes the algorithms that transform an event graph with logical constraints into a compacted event graph without logical constraints. In section 6 the use of hierarchy is explained. The practical results are discussed in section 7.

2 The SLOCOP-II system

The timing verification environment that has been realised consists of two systems: one that deals with the verification in an hierarchical way and one that operates on flattened circuits. Both systems are shown in figures 2 and 3. The difference between the systems is mainly due to different input mechanisms. Once the input has entered the system, the functionality is largely the same.

- The non-hierarchical system takes in a file that has been preprocessed by the DIALOG [10] preprocessing and electrical verification program. This file contains the flattened circuit information and the result of a preprocessing step. The preprocessing splits up the circuit in DC connected components (DCN's) and determines the logic functionality of each of these subnetworks as a function of their inputs. This logic functionality is used to determine the relations between transitions (events) on the inputs of the DCN's and the transitions on the outputs. The necessary conditions on the other inputs for these transitions to occur are also determined. The necessary delay for a given input transition to reach the output can then be calculated and the result is an event graph (see section 3) that can be analysed in several ways:

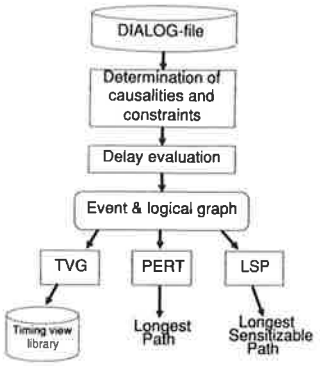

Figure 2: The SLOCOP-II system.

- TVG: timing view generation. Timing views are event graphs in which no false paths remain and where some non essential events have been eliminated to increase the efficiency.
- LSP: the longest sensitisable path is searched.
- PERT: a simple but fast PERT analysis is performed.

- The hierarchical system is connected with various structure generators (schematic editors, module generator, symbolic layout editor ...) through standardised SPI (structure procedural interface) routines [14]. It has two kinds of inputs:
  - the circuit composition, with each of the components given as a black box and the interconnection of these boxes.
  - the timing views of the back boxes that contain their event graphs.

  With these inputs we can construct the event graph and perform the same analyses as we did in the non hierarchical system.

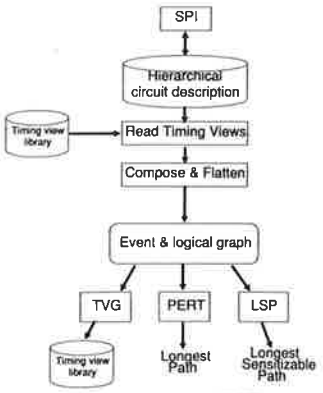

Figure 3: The hierarchical SLOCOP-II system

3 Event and propagation condition determination

To illustrate the method of event generation we show a small example in figure 4. The gate that is shown will, after analysis, have 6 edges in the event graph. Delays and propagation conditions will be associated with them in a similar way as with the one shown in the figure.

3.1 Definitions

We need the following definitions:

- Events: these are logical transitions at circuit nodes, they can be either UP or DOWN. Events are connected through edges, which are weighted with the delay between them. This is the only information used in traditional timing verifiers [3,2,5]. For the false path algorithms it is also necessary to store the propagation conditions with the edges. In figure 4 these propagation conditional are A=1 and B=1 for the one edge that is shown.
- DCN: (DC connected network) a network we want to verify is split up in DCN's, these are parts of the network that are connected through the source-drain nodes, with the exception of Vdd and Gnd nodes. Their inputs are gate nodes, the transistors or external inputs of the circuit under consideration. Their outputs are nodes that are connected to gate nodes of transistors that are not in the DCN. With these nodes we can associate a logic pull-up and pull-down function that we obtain from DIALOG [10].
- BDT: a Binary Decision Tree that represents a logic function [17]. The vertices of the tree have the following properties:
  - Node: a field that contains a reference to a circuit node.
  - Value: a field with the logic value $\in \{0, 1, X\}$ of the vertex. Only leaf vertices have values different from 'X'.

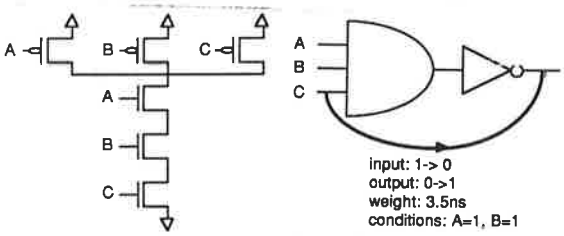

Figure 4: Illustration of the event graph

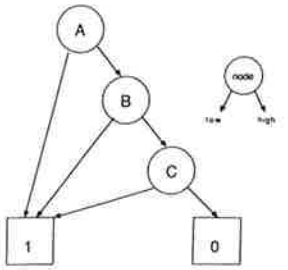

Figure 5: The binary decision tree for (NOT (AND A B C))

  - High: a field that contains a pointer to the vertex we go to if the node to which the Node field refers is set to '1'.
  - Low: a field that contains a pointer to the vertex we go to if the node to which the Node field refers is set to '0'.

  The edges in this tree are the High and Low pointers of each vertex. An example of such a tree is given in figure 5, where the tree for the pull-up function of the three input NAND gate of figure 4 is given.

3.2 Analysis of the BDT's

Motivation Most existing methods for timing verification do not use a logic model so they do not have problems with the determination of propagation conditions associated with certain events. To the authors' knowledge, no general method does yet exist to determine the event dependency and propagation conditions. In cell based design the logic functionality can be included in the library. This was done in [6].

However, with custom design we need a general method for the determination of events and their associated propagation conditions. We can obtain the logic functionality of the circuit via DIALOG [10]. The logic function of a subcircuit is then translated into a BDT and analysed.

A new and general algorithm to derive the events and the propagation conditions on the non-switching inputs as in figure 4 is presented in the following paragraph.

Algorithm We shall first introduce the algorithm intuively, and then present it in a more formal way. Consider the BDT node α in figure 6. If we want to find the events caused by the switching of this node, we start with a begin value, say '0'. The tree is searched via Low until an output is reached, and while doing so the path we follow in the tree is recorded, see figure 6.a. Recording the path amounts to recording the propagation conditions for the output value that is reached. We then trace back until we are at input α and go the other way, in this case via '1'. This represents an input transition on input α. The tree is then searched again, under the constraint that whenever a node that has been recorded in the path of the first traversal is reached, the same direction as recorded (Low or High) must be followed. In figure 6.b the case that we arrive at an opposite output is shown. An event at the output caused by an event at the input has thus been detected. In this case we add the events and the propagation conditions to the event graph, while in the case of figure 6.c no change in the output occurs and no event is recorded.

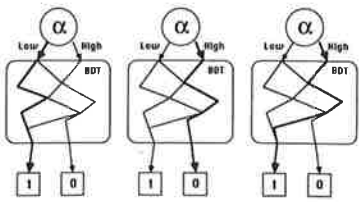

Figure 6: An intuitive explanation

To analyse the logic functions represented in a BDT we use a tree traversal algorithm based on depth first search. There are two basic parts:

1. First we look for a starting value and the input vector associated with it. A pseudo code description of this procedure is given below.

```
void run_tree(tree)
{
if (tree→value != 'X')/* the vertex is a leaf */
  {
  output=tree→value
  find_transition(output,top_tree)
  }
  else{/*the vertex is in the middle of the tree*/
      record_value(1)
      run_tree(tree→High)
      record_value(0)
      run_tree(tree→Low)
      record_value(X)
      }
}
```

   The procedure *record_value(logical value)* records the way in which we proceed through the graph.

2. The procedure *find_transition(value,tree)* that appears in the previous algorithm is used in finding transitions that can occur with the recorded input vector, given the output value arrived at. It proceeds in the following way:
   - Consecutively mark each input that is not recorded with value 'X' as switching.
   - Start processing the tree again. Each time a vertex for which a value has been recorded in the input vector is encountered, continue the traversal in the given direction. That is, proceed through the Low edge if a '0' has been recorded and vice versa. However, in the case an input that is marked switching is reached, follow the opposite path. That is, proceed through the Low edge if a '1' has been recorded. If a vertex for which no value has been recorded in the input vector occurs, both available paths have to be investigated. In doing so the followed direction has, at least temporarily, to be recorded in the input vector.
   - Eventually, a leaf vertex is reached. If the logic value of this vertex is opposite to that recorded in 'output', we can conclude that there is an event that starts at the marked input and causes a transition at the output. An edge can then be added to the event graph between these nodes, together with their input vector.

Once we have all the edges and their propagation conditions the delay associated with each edge is determined. This is done using RC-models proposed by Pennfield, Rubinstein and Horowitz [16] or by circuit simulation if higher accuracy is needed.

Example For the BDT shown in figure 5, corresponding to the NAND gate of figure 4, this works as follows:

- Assume we search for events on the output caused by the switching up of input A.
- When A is Low, the output is '1'. We thus have an event if the output goes to '0' when A switches.
- Since A is the switching input, we backtrack until we arrive at A and go the other way, that is through High.
- The only way to arrive at '0' on the output is to go via B.High and C.High.
- Thus the only way that a rising transition on A can cause an event on the output is with conditions B='1' and C='1' and it has to be a falling event.

4 The longest sensitizable path

We now have a weighted directed graph representing the timing behaviour of the circuit.

During the search for the longest sensitizable path according to the method presented in [8], before adding a new edge to the path it is necessary to check that the propagation conditions are compatible with the propagation conditions of all the other edges already in the path. Those conditions must not only be checked locally, but they must be propagated through the logical model of the circuit to all the related nodes.

The problem is thus equivalent to searching the longest path in a conditional graph. A depth first search is used, with following properties :

1. The search is guided by a low cost heuristic, namely PERT.

2. The search space is reduced by pruning : if the longest pert-path through a node is shorter than the longest sensitizable path already found, then the subgraph of that node must not be searched because the longest sensitizable path through that node can never be longer than the path already found.

With respect to the CPU-times of this longest sensitizable path algorithm, the following remarks can be made :

- When the longest PERT-path is not a false path, the algorithm is almost as fast as PERT.

- For small circuits where the number of paths to check is small, the CPU-times are of course also manageable.
- But for large, "real life" circuits with many false paths, the CPU-times explode and it takes too long before the longest path is found. For a 24-bit ALU with bypass circuitry, it takes more than one CPU-hour to search the longest sensitizable path [8].

5 Code generation

An event graph with propagation constraints as shown with the dotted lines in figure 7 can not be used as such for code generation as presented in LEADOUT [5], because a sequential algorithm, similar to existing algorithms [8] would have to be generated including backtracking and iterations. It is more desirable that code can be generated that can be evaluated in one pass such as is the case with event graphs without propagation constraints.

A first attempt to solve the problem could consist of an explicit path enumeration technique [1], followed by a removal of false paths. In this case the amount of code to be generated corresponds to the expanded number of events in the enumeration. The disadvantage of this approach is that the expanded number of events can grow exponentially in terms of the number of events and causalities.

A first improvement is possible by sharing non-conflicting initial path segments and eliminating the subgraphs that are not sensitisable. This results in a graph without propagation constraints, of which code can be generated. However in the case of reconvergent false paths such as in optimized ALU structures [9] the resulting event graph will still grow exponentially in terms of the number of events and causalities.

In the aforementioned approach all events downward from the root are expanded in order to obtain an event graph without propagation constraints. We implemented an algorithm that tries to build a *compacted event graph without propagation constraints*, where compatible subgraphs are merged again. The fact that certain event sequences in the (unconstrained) expanded event graph can be shared is used to reduce the amount of generated code. It should be noticed however that there are several ways of compacting the resulting unconstrained event graph. Therefore the heuristics described are applied. For the example in figure 7 the result of the compacted event graph is show in figure 8. This results in event graphs without constraints, where the amount of events is only doubled. This allows to generate code with an efficiency comparable to the PERT algorithm that does not consider logic compatibilities. The compaction algorithm is based on a depth first search event graph with logical propagation constraints. The results are presented in table 1. Note that the generation of an unconstrained event graph becomes very time consuming when larger examples are used. However, the evaluation of the generated code remains very fast.

6 Hierarchy

6.1 Motivation

By looking at the occurrences of false paths in the circuit, it can be observed that most false paths occur due to one of the following two reasons :

- false paths due to local logical incompatibilities.
- circuits where the designer intended to create false paths by adding bypass circuitry for speeding up the global circuit.

Examples of local logical incompatibilities can be found in the fulladder circuit of figure 9. Figure 10 gives the event graph and the logical conditions of the carry generation part of this fulladder cell. On this graph can be seen that the logical conditions of some paths are incompatible and these paths are false paths. E.g. the path from 17 ↓ through 14 ↓ to 8 ↓ is false due to the incompatible conditions on input 19.

If all these local logical incompatibilities could be eliminated, there would remain less false paths and this would speedup the LSP-algorithm.

This can be done by using the hierarchy of the circuit.

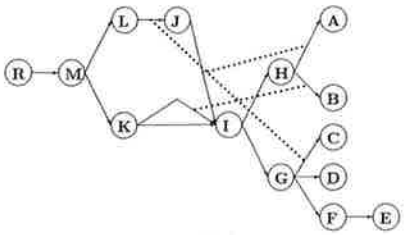

Figure 7: An event graph with logical constraints on signal propagation.

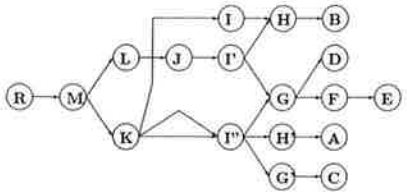

Figure 8: Compacted event graph without explicit logical constraints

The new hierarchical [15] method can be summarized as follows :

- *Generate the timing views for all the basic cells and for the bypass circuitries together with the bypassed cells.*
- *Compose these timing views to become an event graph for the whole circuit.*
- *Run the LSP-algorithm on this event graph to find the longest sensitizable path in the circuit.*

6.2 Timing view generation

The timing view generation method consist of 2 new graph manipulation techniques : the elimination of local logical incompatibilities and a graph reduction by event elimination.

Elimination of local logical incompatibilities As shown in figure 2 the circuit description and the logical functions in the circuit are derived by DIALOG [10]. Also the event graph [5] of the circuit is set up. An event corresponds either to a falling or a rising transition on an electrical node in the circuit. Each edge in the graph, the causality relationship between two events, has a corresponding delay, calculated with a Horowitz type of RC-models[16] or a simulation, and a corresponding set of logical propagation conditions for signal propagation. The total graph is a combined logical and event graph.

False paths can occur in the graph and have to be eliminated. Out of the old event graph, a new event graph without false paths has to be generated. This can be done in several ways [18,19]:

Path enumeration : All the paths are enumerated and the paths with logical incompatibilities are deleted. This method is very easy and would work, but is not efficient because the memory requirements are too large.

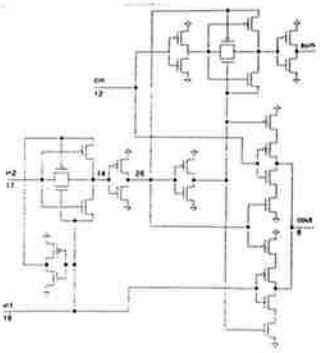

Figure 9: A fulladder circuit

Path enumeration with optimal compaction : Take as much events as possible together so that a minimal graph results. This method is not usefull because it would take to much CPU-time to find the optimal solution.

The SLOCOP-method : A method in between the two previous methods is developed. A depth first search is applied with checking for logical incompatibilities as in the LSP algorithm. During the forward search, the path is created and checked for sensitizability. While backtracking, if a sensitizable path is found, the following algorithm is used for compaction : *Two events in the graph are taken together if they refer to the same circuit node, have the same transition, and have the same subgraph.*

Algorithm :

1. Initialise : ev = rootevent;
2. while (∃ NOT INVESTIGATED out-edge of ev) {
   /* FORWARD */
   Take edge and mark it INVESTIGATED;
   if (edge compatible with current path) {
   Push current state on stack;
   Add edge to current path;
   ev = out-event of edge;
   }
   }
3. /* BACKWARD */
   if (ev == rootevent) exit;
   Try to combine ev with other events;
   Pop previous state from stack;
   go to 2.

This results in a graph without logical incompatibilities where some events have been duplicated. The event graph in figure 10 of the fulladder circuit in figure 9 becomes after elimination of the logical incompatibilities the graph in figure 11.

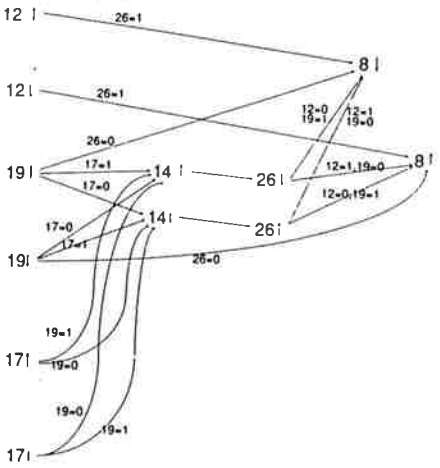

Figure 10: The event graph of the carry generation part of the fulladder

Graph reduction by event elimination As described in the previous section, the number of events has increased due to the elimination of logical incompatibilities. In this section, a method is described to compensate this effect by event elimination in a post processing step.

If an event is eliminated, all the in-edges and all the out-edges of that event are replaced by edges from all the in-events of the eliminated event to all the out-events of the eliminated event. The resulting delay of a created edge is the sum of the delays of the two replaced edges and the resulting logical propagation conditions are the conjunction of the two replaced edges.

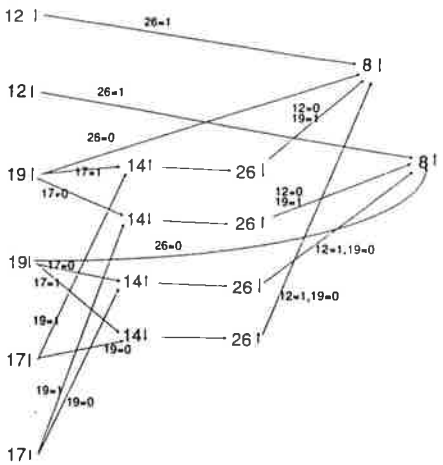

Figure 11: The event graph with logical incompatibilities eliminated

Because the events are eliminated in a post processing step, the CPU-time required by the method must be very low and a very simple algorithm has to be used. The following simple criterion for elimination is used : *eliminate an event if the number of created edges is smaller than the number of replaced edges.* In other words, eliminate an event if the sum of the in-edges and out-edges of the event is larger than their product.

6.3 The hierarchical composition

The timing views can be hierarchically composed and an event graph for the higher level cell is generated. This event graph has less (or none) false paths because all the local logical incompatibilities are eliminated during the timing view generation.

The LSP-algorithm can run on this event graph and the required CPU-times will be much lower than running it directly on the whole flat circuit because all the local logical incompatibilities are eliminated. In many circuits there will be no false paths any more in the hierarchically composed event graph.

Results of this approach are presented in table 2.

7 Conclusion

In this paper a *new and general* timing verification environment SLOCOP-II for MOSVLSI circuits has been presented that allows to avoid indication of *false paths* that is a well known problem in all currently existing timing verifiers. This has required to take into account the logic propagation conditions on individual events in subcircuits, for which new algorithms have been worked out and implemented. Efficiency in longest path searches in SLOCOP-II is obtained by using compiled code generation and the exploitation of the hierarchy. The SLOCOP-II program is illustrated with the timing verification results on a number of CATHEDRAL-II modules.

circuit	cpu time(s)		
	compiled code		interact.
	step 1	step 2	LSP
falsef	0.3	0.1	0.3
8 bit comp.	8.3	0.3	0.3
8 bit counter	4.9	0.3	0.2
7 bit carry-sel.	14.1	0.4	0.1
4 bit alu	16.0	0.3	0.2
12 bit alu	347.5	1.0	15.8
14 bit alu	5109.6	1.8	1258.2
18 bit alu	18272.4	2.2	1903.2
24 bit alu	73587.3	2.5	4235.1

Table 1: Code generation

circuit	interact.	Hierarchical		longest
	LSP (s)	prep.	analysis	path
alu12	0.6	-	-	65.4ns
alu14	760	6.0s	0.5s	55.9ns
alu16	1070	6.0s	0.6s	58.3ns
alu18	1574	6.0s	0.7s	58.3ns
alu20	2275	6.0s	0.8s	61.2ns
alu22	3280	6.0s	0.9s	61.2ns
alu24	4020	6.0s	0.9s	63.7ns

Table 2: Hierarchical verification

References

[1] Robert B. Hitchcock, G.L. Smith, D.D. Cheng, "Timing Analysis of Computer Hardware", *IBM Journal of Research and Development*, Vol. 26, No.1., Jan. 1982, pp. 100-105.

[2] Norman P. Jouppi, "TV: an nMOS Timing Analyser", Proceedings of the Third Caltech VLSI Conference, 1983, pp. 72-85.

[3] John K. Ousterhout, "Crystal: a Timing Analyser for nMOS VLSI Circuits", *Proceedings of the Third Caltech VLSI Conference*, 1980, pp. 58-69.

[4] Seung H. Hwang, Young H.Kim, A.R. Newton, "An accurate delay modeling technique for switch-level timing verification", *23rd Design Automation Conference*, June 29 - July 2, 1986, pp. 227-233.

[5] Thomas G. Szymanski, "LEADOUT: A Static Timing Analyser of MOS Circuits", *IEEE ICCAD'86, Santa Clara, CA*, November 1986, Digest of Technical papers, pp. 130-133.

[6] E. Vanden Meersch, L. Claesen, H. De Man, "SLOCOP: a Timing Verification Tool for Synchronous CMOS Logic", *Proceedings European Solid State Circuits Conference ESSCIRC '86*, Delft, pp. 205-207.

[7] D.E. Wallace, C.H. Séquin, "Plug-in Timing Models for an Abstract Timing Verifier", *Proc. 23 Design Automation Conference*, 1986, pp. 683-689.

[8] J.Benkoski, E. Vanden Meersch, L. Claesen, H. De Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verifiers", *Digest of technical papers ICCAD'87, Santa Clara, CA*, November 1987, pp. 24-28.

[9] M. Pomper, W. Beifuss, K. Horninger, W. Kaschte, "A 32-bit Execution Unit in an Advanced NMOS Technology", *IEEE Journal of Solid-State Circuits*, Vol. SC-17, No. 3, June 1982, pp. 533-538.

[10] I. Bolsens, W De Rammelaere, C. Van Overloop, L. Claesen, H. De Man, " A Formal Approach Towards Verification of Synchronous MOS Circuits",*Proceedings IEEE ISCAS-88*, Vol. 3, pp. 2113-2116, june 1988.

[11] Daniel Brand, Vijay S. Iyengar," Timing Analysis Using Functional Analysis",*IEEE Transactions on Computers*, Vol. 37,No. 10,October 1988.

[12] S. Perremans, L. Claesen, H. De Man, " Static Timing Analysis of Dynamically Sensitisable Paths",*DAC-89*

[13] H. De Man, J.Rabaey, P. Six, L. Claesen, " Cathedral-II: A Silicon Compiler for Digital Signal Processing" ,*IEEE Design & Test*, December 1986.

[14] J. Cockx, " ESPRIT 1058: SPI version 2.3" ,*Silvar-Lisco*, Revision 2302, 6 February 1989

[15] P. Das, P.Johannes, L.Claesen, H.De Man, " Hierarchical timing view generation including accurate modeling for false paths", *Proc. IEEE CICC-89*, May 15-18, 1989, San Diego CA.

[16] M. A. Horowitz, " Timing Models for MOS circuits", *Departement of Electrical Engeneering, Stanford University, Stanford* CA94305, Technical report No. SEL83-003, december 1983.

[17] R. E. Bryant, " Graph-Based Algorithms for Boolean Function Manipulation," ,*IEEE Transactions on Computers, Vol. C-35, no. 8*, august 1986.

[18] J.P. Schupp, " STIVITS: a high performance timing verification system for VLSI-chips based on compiled code generation", *Engineering thesis Kath. Univ. Leuven Belgium*, July 1988 (in Dutch).

[19] L. Claesen, J.P. Schupp, H. De Man, " Accelerated Sensitisable Path Algorithms for Timing Verification based on Code Generation", *Proc. IEEE ISCAS-89*, May 9-11, 1989, Portland Oregon.