CELL ORIENTED FAULT SIMULATION

J. Hartmann, B. Schieffer and U. Sparmann
Fachbereich 14.1 - Informatik
Universität des Saarlandes
D-W6600 Saarbrücken, Germany

Abstract

Currently, in most fault simulators physical defects have to be modeled as stuck-at faults in networks of primitive gates such as AND, NAND etc. Especially for complex gates this modeling technique is rather inexact. In this paper, a fault simulation approach supporting a more powerful and variable fault model is presented. Specification of faults is done in a cell oriented manner. To achieve maximum flexibility, the set of basic cells and their failure modes are not coded into the algorithm but can be specified by the user in a library. For the special case of compiler driven fault simulation it is shown that this increased flexibility does not degrade the performance of the simulator but, in contrast, results in a considerable speed-up (up to factor 35).

1 Introduction

As has been observed by many authors a large amount of the physical defects typical of today's VLSI technologies can not be correctly modeled by stuck-at faults [GCV80, BART82, Mal87]. In order to overcome this deficiency, techniques are under development for automatic extraction of fault models for moderate sized basic cells which take into account aspects like physical layout or defect statistics of the manufacturing process [Mal87, FS88]. But most of the algorithms for test generation and fault simulation can not take advantage of such an approach since they have been tailored to the stuck-at fault model. Thus, it is necessary to generalize these algorithms such that they can also be used in the above setting. In this paper we consider this generalization with respect to fault simulation algorithms for combinational circuits. It is shown that the increased flexibility of the fault model does not result in performance degradations.

In the stuck-at fault model, faults can only be attached to lines. In contrast, our fault modeling technique is cell oriented. Arbitrary combinational failures of basic cells can be described by giving the function of the faulted cell. This fault modeling technique is more powerful since stuck-at faults can be looked upon as cellular faults affecting only cell pins. The set of basic cells with their corresponding fault models is specified in a library. Thus, it is very easy to add a new basic cell or to modify the fault model of an existing one. In particular, the library could be generated by a tool which automates the process of fault modeling like [FS88].

Fault simulation is applied to evaluate the capability of a given set of input patterns to detect the faults of the fault model under consideration. For complex circuits and large pattern numbers it is a very time consuming task. Thus, the increased capability and flexibility in fault modeling must not degrade performance of the simulator. Surely, in cell oriented fault simulation we need additional preprocessing time to analyze the library's fault descriptions and a more expensive fault injection procedure during the simulation phase (for stuck-at fault injection only a line's value has to be inverted). But, on the other hand, the library contains important information about the basic subnetworks of the circuit and their fault mechanisms. By an appropriate preprocessing of this information combined with an intelligent fault injection procedure it can be achieved that the cell oriented approach becomes superior to line oriented fault simulation even for the special case of stuck-at faults.

To investigate the time savings implied by the cell oriented fault simulation approach, the compiler driven fault simulator of [DG87] has been implemented in both versions. For large circuits the cell oriented approach always performed better. The level of improvement compared to the original algorithm depends on the use of complex cells like multiplexers or full-adders, i.e. the amount of information that can be taken from the library. Thus, simulation of a realistic number of input patterns yields for circuits consisting only of simple gates (AND, NAND, etc.) a speed-up factor of approximately 2 while for circuits heavily using complex cells speed-up factors of even up to 35 were gained.

The paper starts by motivating and defining our cell oriented fault modeling technique. Then, the basic modifications necessary to generalize fault simulation algorithms to this approach are given. Section 4 explains the effects which speed-up cell oriented fault simulation compared to the line oriented approach. Section 5 gives experimental results by comparing implementations of a compiler driven fault simulator in both versions.

2 Cell oriented fault modeling

In the stuck-at fault model it is assumed that exactly one input or output line of a gate is fixed to a logical value (0 or 1). Internal faults of complex cells can only be modeled by replacing these cells by networks of simple gates (AND, NAND etc.). As an example of the inadequacy of this modeling technique consider the NMOS complex gate and its corresponding gate level representation shown in figure 1. The function of the correct gate is $\overline{(i1 \vee i2) \wedge (i3 \vee i4)}$. If the gate is faulted by the open marked f_{open}, the faulty gate computes the function $\overline{(i1 \wedge i3) \vee (i2 \wedge i4)}$. This fault can not be modeled correctly by a stuck-at fault in the gate level representation, since none of these faults results in the functional behaviour given above.

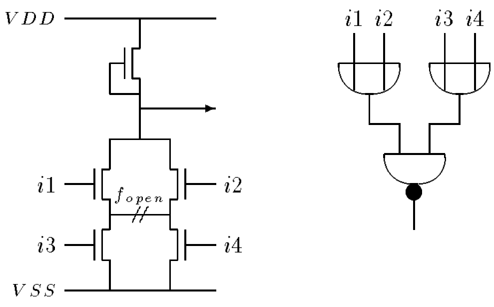

Figure 1: NMOS cell faulted by an open

Applying an inadequate fault model for test generation and fault simulation implies two main deficiencies:

- Some of the physical failures are not correctly modeled by the fault model and therefore may not be detected by the generated test.
- Computing time is wasted for faults which only occur in the model and not in the actual circuit.

In order to overcome these deficiencies, new fault models have been derived for technologies like NMOS and CMOS [GCV80, Wad78]. But the fault modes of circuits depend not only on the technology used for fabrication but also on defect statistics of the manufacturing process and the physical layout. Since physical failures depend on so many conditions (and thus change whenever one of these factors changes), it is important to automate the process of fault modeling. In [Mal87, FS88] a method called *Inductive Fault Analysis* (IFA) is proposed which can be used to derive realistic fault models for circuits by taking into account fabrication technology, defect statistics of the actual process, and physical layout. By analyzing examples of CMOS circuits the following results were obtained in [FS88]:

1. Only less than 50% of the faults extracted by IFA could be correctly modeled by stuck-at faults.
2. No more than 7% of the extracted faults caused sequential behavior.

From the above observations it follows that the most urgent demand on algorithms for fault simulation and test generation is the ability to cope with *arbitrary combinational* fault models.

To achieve this flexibility in fault modeling, in our approach the set of basic cells and their corresponding fault models are not coded into the algorithm but are specified in a library. As a consequence the algorithm can be easily adapted to a specific manufacturing process or cell library utilizing a method like IFA. For each cell type t the set of possible faults F_t is specified. A fault $f \in F_t$ can alter the cell's behavior in an arbitrary manner as long as it remains combinational. It is described by its *set of test alternatives (TA)* which specifies the input/output pairs for which correct and faulty cell differ, i.e.

$$TA_t(f) := \{(i,o) | out_t(i) \neq out_t^f(i) = o\},$$

where out_t and out_t^f denote the functions of the correct and faulted cell, respectively. As an example let us consider the fault of figure 1. It can be described by its test alternatives

$$TA_t(f_{open}) = \{(0110,1),(1001,1)\}.$$

(The value of input i_k is given by the k^{th} bit from the left.) Of course, stuck-at faults can be described in our model as well. For example the stuck-at 1 fault at input i_1 of the above cell has the following TA

$$TA_t(i1 \text{ s-a-1}) = \{(001*,0),(0001,0)\},$$

where $(001*,0)$ is an abbreviation for the tests $(0010,0),(0011,0)$ ('$*$' stands for an arbitrary element from $\{0,1\}$).

Surely, the above modeling technique is only adequate for moderate sized primitives. (The representation of test alternatives can grow exponentially in the number of inputs.) Since exact fault modeling using a method like IFA is a very expensive process, it is only applicable to small cells. Thus, the above restriction is natural.

3 Cell oriented fault simulation

The task of fault simulation is to evaluate the capability of a given set of input patterns to detect a prescribed set of faults. This is normally done in the following manner:

```
foreach input pattern
begin
   simulate correct circuit;
   foreach undetected fault
   begin
      inject fault and simulate circuitry depending on
      fault location;
      if output of correct and faulty circuit differ
         then mark fault detected;
      fi;
   end;
```

end;

To speed up this process, simulations are normally executed for w (= word length of the given computer) input patterns in parallel, utilizing bitwise logical operations [WEF+85, AS87, DG87, BHKS91]. Thus, in each loop w input patterns are treated in parallel.

The main difference between line and cell oriented fault simulation consists in the way faults are injected. In stuck-at fault simulators, fault injection is done simply by inverting the w-bit assignment of a line. Thus, at most two faults (stuck-at 0 and stuck-at 1 of the given line) are considered. In cell oriented fault simulation fault injection is done at cell outputs, considering all internal faults of a basic cell simultaneously.

The following four sets comprise all the information necessary for fast cell oriented fault injection. They are computed in the preprocessing phase during parsing of the fault library.

$$INPUTS_t := \bigcup_{f \in F_t} \{i \mid \exists o : (i,o) \in TA_t(f)\}$$

is the set of input combinations that allow a fault injection at a cell of type t. For an $i \in INPUTS_t$ the corresponding faulty output combinations form

$$OUTPUTS_t(i) := \{o \mid \exists f \in F_t : (i,o) \in TA_t(f)\}.$$

The set

$$FAULTS_t(i,o) := \{f \in F_t \mid (i,o) \in TA_t(f)\}$$

contains those faults that can be tested with input i and then lead to the faulty output o, and

$$FAULTS_t(i) := \bigcup_{o \in OUTPUTS_t(i)} FAULTS_t(i,o)$$

comprises all faults that can be tested by input combination i. For a fast realization of set manipulations by bitwise logical operations the sets $FAULTS_t(i)$ and $FAULTS_t(i,o)$ are stored as bit vectors.

Let g be the cell under fault injection, F_u the set of undetected faults at g and t the type of g. A bit position from $\{1,\dots,w\}$ is said to *match* the input combination i at gate g if g has i as input for this bit position. Then, the fault injection works as follows:
If a pattern $i \in INPUTS_t$ is able to detect an yet uncovered fault (i.e. $FAULTS_t(i) \cap F_u \neq \emptyset$) the bit positions which match i at g are determined. After for all matched i's, an element o from $OUTPUTS_t(i)$ (with $FAULTS_t(i,o) \cap F_u \neq \emptyset$) has been inserted at the matching bit positions, fault simulation is started. If a faulty signal reaches a primary output in a bit position that matched i and where o was inserted, the faults from $FAULTS_t(i,o)$ are removed from F_u. Injection of faulty output combinations, subsequent fault simulation, and contingent fault removal are repeated until for all matched i's each element from $OUTPUTS_t(i)$ has been taken into account. Thus, there are at most $\max\{\,|OUTPUTS_t(i)|\mid i \text{ input to } g\}$ simulation runs for cell g.

Note that word parallel operations can be heavily applied to speed up the above process. Thus, they are not only utilized for set operations with respect to F_u, $FAULTS_t(i)$ and $FAULTS_t(i,o)$ but also for identification of matching bit positions, fault insertion, and determination of bit positions where faults have been detected. (More detailed information about the fault injection procedure can be found in [HSS91].)

4 Advantages of cell oriented fault injection

Surely, cell oriented fault injection is more expensive in terms of running time than line oriented fault injection. On the other hand it substantially reduces the number of necessary simulation runs. This reduction results from three different effects: Event checking at cell borders, fault collapsing, and better utilization of word parallel processing. In the following these effects will be clarified by examples comparing cell oriented and line oriented fault injection for the special case of stuck-at faults.

Event checking at cell borders Fault simulators based on line oriented fault injection insert a stuck-at fault if its effect can be observed at the faulty line. As an example consider the gate level implementation of a multiplexer shown in figure 2. For input combination 111 the stuck-at-0 fault at the select-input will be injected and a simulation run will be started.

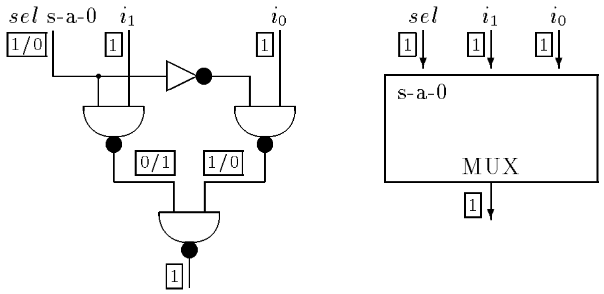

Figure 2: Multiplexer with input 111

In contrast, in the cell oriented approach for a given input combination internal faults of a basic cell are only injected if they actually propagate to the cell's outputs. Thus, in the example above no fault will be injected at the select input of the MUX-cell since there is no propagation to its output (i.e. sel s-a-0 $\notin FAULTS_{MUX}(111)$).

Fault collapsing In line oriented fault injection at most one fault is injected per simulation run at a single word position. Thus, for the EXOR-cell given in figure 3 with input combination 11 the faults a s-a-0, b s-a-0, c s-a-1 and d s-a-1 are all simulated separately.

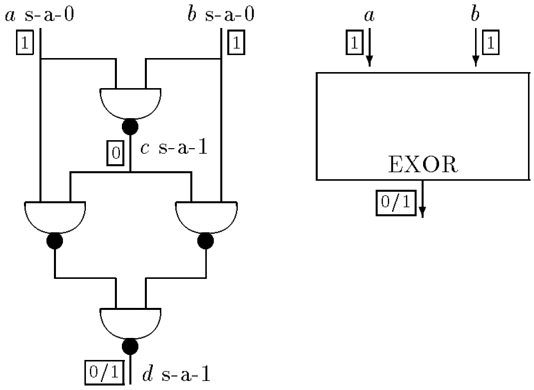

Figure 3: Exor-gate with input 11

In cell oriented fault injection one simulation run identifies for a bit position whether for a given input i a faulty output o propagates to a primary output. If this is the case, the elements of $FAULTS_t(i,o)$ are removed from the fault list. When $FAULTS_t(i,o)$ contains more than one element, several faults are processed by one simulation run. Thus, in our example the four faults a s-a-0, b s-a-0, c s-a-1 and d s-a-1 are simulated in parallel, since they all result in the same faulty response, i.e. belong to $FAULTS_{EXOR}(11,1)$.

Better utilization of word parallel simulation In the line oriented approach at most two faults are injected for word parallel fault simulation, i.e. the stuck-at-0 and stuck-at-1 fault of the line under fault injection. Consider the situation of figure 4 where only the faults b s-a-1, c s-a-1 and d s-a-1 remain to be simulated. Applying line oriented fault injection on a hypothetical 3-bit machine, for each of these faults a separate simulation run will be started.

On the other hand, since for the given assignment the effects of each of these faults are confined to different word positions, one simulation run would be sufficient. In the cell oriented approach all internal faults of the cell are considered in parallel for fault injection. Thus, word positions can be better utilized and in our example all three faults are simulated in parallel (b s-a-1 in bit position 1, c s-a-1 in bit position 2, and d s-a-1 in bit position 3).

5 Experimental evaluation

To investigate the running time improvements implied by cell oriented fault simulation, the compiler driven fault

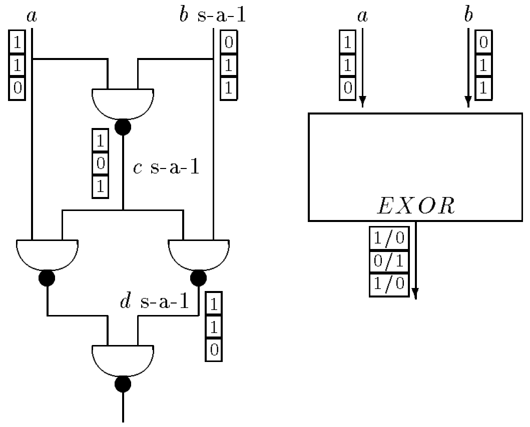

Figure 4: Word-parallel fault simulation

simulator of [DG87] has been implemented in a line and a cell oriented version. In compiler driven simulation, a software routine is generated for simulating the circuit. For each gate it contains an instruction block calculating the gate's function. To ensure that no gate is evaluated before its inputs have been computed, the order of instruction blocks has to respect a topological order[1] of the gates.

Besides reducing the number of simulation runs, cell oriented fault simulation also naturally implies a method to speed-up the actual simulation process, which has only been applied to event driven simulators [SSB89] before:
In compiler driven simulators a fault is injected by inverting the correct assignment of a line. Then, fault simulation is performed by jumping to the first statement evaluating a gate that depends on the faulty line. Thus, the gate level representation used for modeling the internal faults of a basic cell is also applied to simulate the correct cell. In contrast, in the cell oriented approach fault injection is done at cell boundaries. As a result any code can be used for simulating the correct cell as long as it computes the correct logic function. Consider the EXOR gate as an example. In the line oriented simulation routine a gate level representation as given in figure 3 has to be applied for simulation. Thus, four AND and four NOT operations have to be executed, and the values of all internal lines have to be stored in main memory. In the cell oriented simulation routine executing one single exor operation and storing the result in main memory is sufficient.

The following tables summarize the results of our implementation, more detailed information can be found in [HSS91]. Measurements are given for the largest benchmark circuits [BF85] (c6288 and c7552) and some arithmetic circuits (a 16-bit integer multiplier im16, a 64-bit

[1] The gates of a circuit are topologically ordered, iff for all gates g its predecessors occur earlier than g itself. Since we consider combinational circuits which contain no loops such an order always exists.

circuit	patterns	simulation runs line or.	cell or.	red. factor
c6288	30,016	22,967	18,381	1.2
	192	6,306	2,538	2.5
c7552	30,016	345,266	183,175	1.9
	1,024	25,333	12,335	2.1
im16	30,016	263,165	16,727	15.7
	448	19,980	2,778	7.2
ia64	30,016	637,183	3,266	195.0
	832	24,579	1,346	18.3
fa32	30,016	1,203,119	269,564	4.5
	1,248	65,765	16,082	4.1
fm32	30,016	8,922,105	1,393,124	6.4
	1,568	674,016	101,349	6.7

Table 1: Number of simulation runs

circuit	simulation time line or.	cell or.	speed-up factor
im16	8,030 μs	1,360 μs	5.9
ia64	2,860 μs	400 μs	7.2
fa32	5,020 μs	1,110 μs	4.5
fm32	19,600 μs	6,100 μs	3.2

Table 2: Times taken for one simulation

integer adder ia64, a 32-bit floating-point adder fa32 and a 32-bit floating-point multiplier fm32). c6288 and c7552 are only built from primitive gates AND, NAND, OR, NOR, NOT. The arithmetic circuits mainly consist of complex cells as full-adders, EXORs or multiplexers. For cell oriented fault simulation, circuit representations with complex cells as primitives were used if they were available (not so for c6288 and c7552). In contrast, in the line oriented approach, complex primitives had to be expanded for the modeling of internal faults.

Table 1 illustrates the reduction of simulation runs due to the effects described in section 4. Measurements are given for simulations with 30016 random patterns and a dynamic abort where fault simulation has been stopped as soon as 64 subsequent patterns did not test a not yet covered fault.

The acceleration of simulation runs is shown in table 2. Times required for a fault free simulation of the circuits with complex cells are opposed to those of their expanded counterparts.

Both effects (accelerated simulation routine and reduced number of simulation runs) are summarized in table 3 which gives running times for complete fault simulations (with pattern numbers as in table 1) measured on a SUN SPARCstation 2.

The additional running time for the complicated fault injection phase is more than outweighted by the positive effects of cell oriented simulation. The speed-up grows with increasing complexity of the basic cells. For the benchmark circuits c6288 and c7552 which consist of simple gates speed-up factors of only 1.1 up to 2.0 were measured. For the remaining circuits, which are mostly built from complex cells, these factors range from 11.2 to 35. On the other hand, as becomes clear from the entries for im16 or ia64, speed-up can not be predicted by considering only the complexity of the basic cells. This is due to the effect that the reduction in simulation runs also depends on the patterns simulated and the structural properties of the overall circuit. Finally, the overall acceleration is not equal to the product of the speed-up factors in tables 1 and 2, since the costs of a fault simulation run depend on the location of the fault simulated. Thus, saving a simulation run for a fault near to a primary output is of little effect (as an example consider the 30016 measurement for ia64).

circuit	running time line or.	cell or.	speed-up factor	fault coverage
c6288	32.8 s	30.6 s	1.1	99.6%
	4.6 s	2.3 s	2.0	99.6%
c7552	290 s	199 s	1.5	95%
	23.7 s	14.3 s	1.7	91.9%
im16	271 s	16.6 s	16.3	98.6%
	64.6 s	2.4 s	26.9	98.3%
ia64	620 s	17.9 s	35.0	90%
	23.2 s	1.1 s	21.0	89.3%
fa32	23 min	116 s	11.7	82.8%
	87.7 s	7.8 s	11.2	81%
fm32	24:28 h	1:08 h	21.5	78.3%
	1:56 h	314 s	22.3	67.4%

Table 3: Running times for complete fault simulation

6 Conclusions

In this paper, a cell oriented fault specification technique for arbitrary combinational failures has been proposed. To achieve maximum flexibility in fault modeling, the set of basic cells and their failure mechanisms are not coded into the test algorithm but are read in from a library. For the special case of fault simulation we have argued that this increased flexibility does not degrade performance but in contrast results in a considerable speed-up.

Our results are encouraging for further research in this direction. A test generation algorithm supporting the fault modeling technique proposed in this paper is under development. Both algorithms will be integrated in the design system CADIC [BHK+87] as components of an interactive tool for test generation. (First steps in this direction have been described in [BBH+90].)

Clearly, in order to derive high quality test sets, combinational fault modeling alone will not be sufficient in many applications. Thus, we are currently developing specification and simulation techniques for failure mechanisms which turn a combinational cell into a sequential circuit.

Let us conclude by some critical remarks about compiler driven fault simulation. Of course, compiler driven simulation has the advantage of easy implementation. The generation of the most critical part of the algorithm (the software routine which simulates the circuit) is straightforward. On the other hand, there is one major drawback. For big circuits, the simulation routine gets extremely large. Thus, compilers or linkers may not be able to handle it. The code optimization described in chapter 5 is a first step to reduce the size of the generated program but further research concerning this topic is necessary. In this context, utilizing hierarchical representations of circuits to generate compact simulation routines [KS91] is promising.

Acknowledgements

The authors gratefully acknowledge Prof. Dr. Günter Hotz for his support and encouragement.

References

[AS87] K. J. Antreich and M. H. Schulz. Accelerated fault simulation and fault grading in combinational circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):704–712, September 1987.

[BART82] C.C. Beh, K.H. Aray, C.E. Radke, and K.E. Torku. Do stuck-at fault models reflect manufacturing defects? In *Proceedings 1982 International Test Conference*, pages 35–42, 1982.

[BBH+90] B. Becker, Th. Burch, G. Hotz, D. Kiel, R. Kolla, P. Molitor, H. G. Osthof, G. Pitsch, and U. Sparmann. A graphical system for hierarchical specifications and checkups of VLSI circuits. In *Proceedings of the 1st European Design Automation Conference (EDAC90)*, pages 174–179, 1990.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings IEEE International Symposium on Circuits and Systems*, 1985.

[BHK+87] B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.G. Osthof. Hierarchical design based on a calculus of nets. In *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC87)*, pages 649–653, June 1987.

[BHKS91] B. Becker, R. Hahn, R. Krieger, and U. Sparmann. Structure based methods for parallel pattern fault simulation in combinational circuits. In *Proceedings of the 2nd European Design Automation Conference (EDAC91)*, 1991.

[DG87] W. Daehn and M. Geilert. Fast fault simulation for combinational circuits by compiler driven single fault propagation. In *Proceedings 1987 International Test Conference*, pages 286–292, 1987.

[FS88] F.J. Ferguson and J.P. Shen. Extraction and simulation of realistic CMOS faults using inductive fault analysis. In *Proceedings of the International Test Conference*, pages 475–484, 1988.

[GCV80] J. Galiay, Y. Crouzet, and M. Vergniault. Physical versus logical fault models MOS LSI circuits: Impact on their testability. *IEEE Transactions on Computers*, C-29(6):527–531, June 1980.

[HSS91] J. Hartmann, B. Schieffer, and U. Sparmann. COFS - A cell oriented fault simulator. Internal Report, Universität des Saarlandes, 1991.

[KS91] R. Kolla and B. Serf. The virtual feedback problem in hierarchical representations of combinational circuits. *Acta Informatica*, 28:463–476, 1991.

[Mal87] W. Maly. Realistic fault modeling for VLSI testing. In *Proceedings 24th ACM/IEEE Design Automation Conference*, pages 173–180, 1987.

[SSB89] M. H. Schulz, B. H. Seiss, and F. Brglez. Hierarchical fault simulation in combinational circuits. In *Proceedings 1989 European Test Conference*, pages 33–40, 1989.

[Wad78] R. L. Wadsack. Fault modeling and logic simulation of CMOS and MOS integrated circuits. *The Bell System Technical Journal*, 57:1449–1474, 1978.

[WEF+85] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault simulation for structured VLSI. *VLSI Systems Design*, pages 20–32, 1985.