Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

A New Approach to Distributed Functional Fault Modeling

Dr. Sumit Ghosh

AT&T Bell Laboratories Research

Holmdel, NJ 07733, USA.

Abstract

A new approach to functional deductive fault simulation is presented in this paper. In this approach, fault models of complex functional digital components are derived using a new modeling technique and a decomposition principle. Also this approach utilizes the deductive technique [AD72] and the fault simulation algorithm is distributed in all the fault models. Every model is independent and is capable of scheduling itself for execution when it receives the input vectors and fault lists at all its input ports. As a result, parallelism may be utilized with relative ease. Functional fault models are also observed to be invariant to their internal implementation and performance measurements indicate that functional fault simulation is significantly faster than gate-level simulation. The CPU time rises linearly with the increasing number of devices simulated as shown by a limited set of experiments. This approach has been verified in the RDV [GS84] at Stanford University.

1. Introduction

1.1 Previous Work

Before Seshu [SS65] introduced the parallel technique, fault simulation was based on the serial technique where, corresponding to every fault, a faulty circuit is simulated and its output compared against that of a good circuit. As a result, serial fault simulation was slow. The parallel technique is faster and is based on the following principle. In it, the fault-free circuit and a number of faulty circuits are simulated simultaneously. The number of faults simulated during a single pass is a function of the word length of the host machine; for multiple faults, many passes may be required. The performance improvement of the parallel over the serial technique is, therefore, proportional to the word length of the host computer. TEGAS [SS72] implements the parallel technique. However, modeling of functional blocks is very complex because, for fault simulation, each block must be broken into an equivalent gate-level representation for fault simulation.

The SALOGS [CG78] fault simulator implements the serial technique to fault simulate gates and flip-flops. Multiple simulation passes are required, one for each fault, and the CPU time increases quadratically with the number of devices simulated; however, a "states-applied analysis" which is equivalent to fault collapsing is carried out prior to fault simulation, which identifies all equivalent faults and eliminates them from consideration. Consequently, the CPU time is less than it would have been without the analysis.

Armstrong [AD72] introduced the deductive technique to further ed in a linked list and, therefore, the limitation of finite computer word length does not apply. Faulty circuits are not explicitly simulated; however, given an input vector(s) to a circuit, output fault lists are deduced from knowledge of circuit behavior. [2] Because of its deductive nature, this technique is complex to implement.

The concurrent fault-simulation technique was introduced by Ulrich and Baker [UE74] around the same time as the deductive technique and was intended to be an improvement over the parallel method. It is ideally suited for a multiprocessor machine. The fault-free and (multiple) faulty circuits are concurrently simulated. During simulation of a faulty circuit, however, if the output at any stage is identical to that of the faultless circuit, the faulty one is dropped from consideration because the corresponding fault cannot be detected. In a single-processor machine, the concurrent evaluation of multiple copies is emulated by lists and list processing techniques.

The SIMDE fault-simulator system [MA76] uses the deductive principle to fault simulate gate-level circuits. A few general techniques for functional-level fault simulation have been reported in [SS73] that may enhance fault simulation speed; these techniques are specific to parallel fault simulation. A functional concurrent fault simulator was introduced by d'Abreu [dM80] where, a new scheduling approach is capable of accurately simulating timing and stuck-at faults simultaneously.

Hirschhorn [HS83] described an extension of concurrent fault simulation to the functional level incorporated in the FANSIM3 logic simulator. Fault-simulating storage elements (such as memory) normally requires every fault associated with each element to be stored separately. For a large number of faults, this resource requirement becomes excessively large. FANSIM3 solves this problem by storing the response from the faultless element and only the differences (typically a few words) generated by the faults; it

[1] This research was performed while the author was associated with the Computer Systems Laboratory, Stanford University, CA - 94305.

[2] Therefore, the word simulation is not appropriate; however the terminology "deductive fault simulation" will continue to be used in this investigation as it is widely known in the design-automation community.

also uses a concurrent storage evaluation algorithm where elements whose state differs because of storage faults only are simulated concurrently with the faultless element. This reduces the number of faulty functional simulations and simplifies the problem of convergence and divergence of functional-fault effects. Hajj [HI83] introduced a fault simulator for MOS that extracts the logic expressions for subcircuits from layout information and uses a combination of concurrent and parallel methods to evaluate these expressions. Moorby [PM83] reported the "parallel value list" method that combines the parallel and concurrent fault simulation techniques. This approach uses multiple fault lists similar to concurrent fault simulation but they are compacted into computer words as in a parallel fault simulator.

In summary,

- The fault modeling knowledge and the functions for evaluating gate-level primitives are intertwined with the centralized simulation scheduler. Consequently, addition of fault models of new primitives into the fault simulator is often very difficult.
- Fault models are devoid of any scheduling which is completely contained in a centralized scheduler. Consequently, utilization of parallelism that might be present is difficult.
- fault modeling of functional devices is difficult. Usually a functional device is synthesized from gate-level primitives which are then fault simulated. Problems associated with this approach are
  - Difficulty in achieving accuracy of the functional device through synthesis of gate-level primitives.
  - The CPU time taken for fault simulation at the gate-level is usually very large and therefore such simulations are expensive.

1.2 A Novel Approach to Fault Simulation

First, a few concepts and definitions are presented after which the new fault simulation approach used in RDV is detailed.

The concept behind **fault simulation** is that of simulating a circuit under various fault conditions, in order to evaluate an input (test vector) in terms of its fault detection and location capability. Mathematically, fault simulation may be described as follows. Let C^g denote an arbitrary good circuit and f = {f1,f2,...,fn}, is the set of all faults of interest in C. Assuming a 0,1 value system in RDV, let the good output response from using a test vector T be C^g(T). Assume C^f denote the circuit under fault condition f in {f1,....,fn}. A fault in {f1,...,fn} is said to be detected by T if $C^f(T) \neq C^g(T)$.

A fault list at an output of a circuit, under fault simulation, is a list of faults that will force a faulty value at the output, for a given input. Mathematically, if a circuit denoted by C has the following set of relevant faults, f = {f1,...,fn}, then the fault list at an output O (good value O^g) will be FO = {fi,...,fk}, such that, for each fault in FO, the value at the output O will be $O^f \neq O^g$.

A fault model, corresponding to a device, is defined as an abstraction that contains adequate knowledge to determine the output fault lists, given the input vectors and the input fault lists to the device. For such a fault model structure, the deductive fault simulation technique was chosen as appropriate in RDV, because, both the fault model and the deductive technique operate in a dataflow manner. Fault models are expressed in Ada [AD83].

This investigation is limited to single stuck-at logical 0 and logical 1 faults, because, a high percentage of all faults in a circuit may be detected when modeled in terms of stuck-at faults [BM76]. It is also limited to zero-time-delay fault models. An advantage of the zero-time-delay approach is that the fault models and lists are less complex, as compared to the case where every model has an unique delay value. The disadvantage, however, is that delay faults may not be simulated.

In contrast to conventional fault simulators where scheduling is concentrated in a centralized scheduler, in RDV a large degree of the scheduling is distributed among the fault models. The remaining small part of scheduling is a part of the Ada environment. The models are independent and distinct from each other and the Ada scheduler, and, therefore, to fault simulate a new type of device it is only required to add a fault model.

Because a large degree of scheduling is distributed among the fault models and only a small remaining part is contained within the Ada environment, parallelism can be utilized with relative ease. Every fault model is capable of scheduling itself when input vectors and fault lists are asserted at all its input ports. Consequently, at any instant during simulation more than one fault model may be executing simultaneously. Fault models, in RDV, do not introduce any new technique for parallelism into the Ada environment; they simply preserve and utilize the parallelism that is already in Ada. A limitation of the above approach, however, is that models are complex and deriving them may be difficult.

Fault list propagation through a functional device with a complex input-output relationship is difficult. However, fault models for functional devices including sequential devices with memory may be derived in RDV. The capability is made possible by a combination of three factors.

- The high-level language Ada in which the complex behavior of functional devices may be expressed.
- A decomposition principle which may structurally decompose most commonly used digital devices into smaller units, where each unit is fault simulated in succession and the intermediate fault lists are propagated from one simulated unit to another. This principle is not detailed here and the reader is referred to [GS84].
- Fault information related to storage is stored using Ada variables. This technique is not presented here and the reader may refer to [GS84].

In summary, the fault simulator in RDV has the following characteristics :

- For every device type, the fault modeling knowledge and the function to evaluate the device are stored in a single fault model. Fault models are independent entities and distinct from the simulator nucleus.
- To a large degree, scheduling is distributed in every fault model as opposed to conventional fault simulators where the entire scheduling is contained in a centralized scheduler. As a result, parallelism can be utilized with relative ease.
- Fault models for functional devices and sequential blocks with memory may be derived with relative ease.
- A fault model of a device may be integrated with its functional and timing counterparts in the rule-based design verifier RDV.

Section 2 presents the basic concepts of RDV followed by the derivation of fault models for gates. The fault models for gates explain the basic constituents of a model – the fault modeling knowledge, the evaluation function, and the distributed scheduling and also how it is independent of the rest of the simulator. In section 3, functional-level models are derived for several commonly used digital devices (using the decomposition principle) such that they can each be integrated with their respective functional and timing models into the RDV. Also, the limitation of the decomposition principle is presented followed by the invariance principle which states that the results of functional fault simulation are invariant to the implementation details. Section 4 contains an analysis of the measurements obtained from running fault simulation on several example circuits in RDV.

2. Basic Concepts and Gate Fault Models in RDV

2.1 Basic Concepts

The deductive fault simulation algorithm [AD72] incorporated in RDV deduces the output fault lists based on the input vectors, input fault lists, and the input-output behavior of the device being simulated without any explicit simulation that usually involves scheduling of good and faulty events.

In RDV, prior to initiating fault simulation of a digital design, input vectors are assigned to all the primary inputs and any required initialization is performed. When a device as a result of initialization has input vectors and fault lists asserted at all its inputs, the corresponding fault model is executed. Because scheduling is distributed, at any instant during simulation, the number of such models that are executing simultaneously may be greater than one. The execution process is described as follows.

Every path or net in a circuit is identified in RDV by an integer. A net may connect several nodes of many devices and has a single logical value at any instant during simulation. When a fault model of an n-input device, where the input ports are identified by I1,...,In is executed, first, fault lists F(I1), ...,F(In) are created. This process is termed "creating list." When the input value at net Ik is logical 0, F(Ik) contains a single fault entry Ik stuck-at 1. If the value at net Ik is 1 instead then F(Ik) will contain the fault entry Ik stuck-at 0. A fault list is implemented as a linked list of fault records or entries, where each record consists of three fields. In RDV, a fault list for net Ik is pointed at by F(Ik), where F identifies that part of the simulation database that relates to fault simulation. The first field contains the identifier of the net where the fault originates, the second field identifies the nature of the fault – whether stuck-at 0 or stuck-at 1. The third field points to the next fault record and when none is present it points to nil. Figure 1 shows a fault list corresponding to the net identifier X, where the entry is X stuck-at 1.

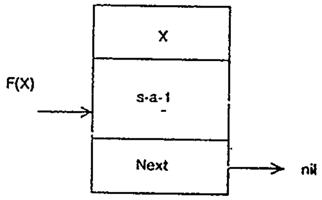

Figure 1: FAULT RECORD FOR NET X.

Once input fault lists are created at the inputs of a device, then, the output fault lists are deduced based on the input lists and the input-output behavior of the device. The process of deduction consists of selectively merging the input fault lists to form output fault lists at the output of the device and is termed "merging lists," as discussed in section 2.2.

Because of reconvergence in a circuit, it is sometimes necesary in the deductive technique to subtract selected fault entries from a fault list. In the procedure responsible for subtracting fault lists, the fault entries that are selected for subtraction are usually those that are common beween two or more fault lists. The two procedures that are responsible for extracting the common entries from two or more lists, and subtracting selected entries from a fault list are respectively termed as "common-entry," and "subtract-lists."

A fault model completes execution after the output fault list is deduced and is propagated to the inputs of other fault models that are connected to it. In RDV, fault models are successively activated, executed, and then the fault lists are propagated to the primary output of the circuit. The fault simulation process terminates when output fault lists are available at all primary outputs of the circuit; the contents of these lists identify all faults that are detectable for the given input vector.

2.2 Fault Model of a two-input AND Gate

For the AND gate shown in Figure 2, the fault model is derived in Figure 3.

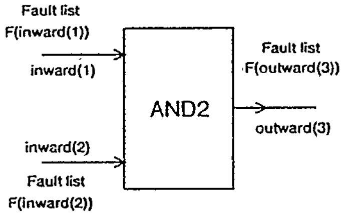

Figure 2: AND GATE.

First, the model checks whether inputs have been asserted at its inputs and when true it activates itself for execution. The output fault list is deduced based on the input fault lists, input vectors, and the input-output behavior of the gate. Therefore, deducing the fault list is completely independent of other models and the rest of the simulator. At the end of execution, the fault model propagates output fault lists to all other fault models that are connected to its output port. Subsequent models activate themselves when inputs are asserted at their input ports and, the fault lists are propagated in the direction of the primary output of the circuit. The self-activation and propagation of fault lists to other models at the end of execution constitutes the distributed scheduling of the model. The underlying Ada scheduler simply places active fault models in a queue for execution by a single processor and terminates the executed models.

Also, because the input fault lists are accessed from the simulation database indirectly through the input identifiers input (1) and input (2) and since the computation on input fault lists to derive the output fault lists is based on these identifiers, the fault model is independent of the functional and timing models and may, therefore, be integrated into RDV.

In the device model shown in Figure 3, the input identifiers are input (1) and input (2) and the output identifier is output (3). The values at input(1), input(2), and output(3) are stored in t(input(1)), t(input(2)), and t(input(3)). Until vectors are assigned to the inputs, the fault model is not executed. During execution, first, the fault lists F(input (1)) and F(input (2)) are created by the create-list based on the input vector. Then, the output value is determined using the Boolean relationship and the fault list F(output (3)) is created by the create-list. Based on the input vector and considering only single stuck-at faults, the output fault list is determined using the merge-lists are in the following way.

If the inputs input (1) and input (2) are both assigned logical 1, then every record (fault) in the fault lists F(input (1)) and F(input (2)) will force the respective input to 0. Both F(input (1)) and F(input (2)) must be merged into F(output (3)) because for all faults that force an input to 0, the output will be 0 – different from the correct output and, therefore, these faults are detectable at the output. When input (1) and input (2) are both logical 0, then, every record in the fault lists F(input (1)) and F(input (2)) will force the respective input to 0.

```
Task body AND2 is
 input   : array(1..2) of node;
 output  : array(3..3) of node;
 var k is integer;
 begin
  if t(input(1))=one and t(input(2))=one then
   t(output(3)):=one; else t(output(3)):=one; end if;
             --  output determined by input vector

  if t(input(1))=zero then create-list(input(1),s-a-1);
  else create-list(input(1),s-a-0); end if;
  if t(input(2))=zero then create-list(input(2),s-a-1);
  else create-list(input(2),s-a-0); end if;
  if t(output(3))=zero then create-list(output(3),s-a-1);

  else create-list(output(3),s-a-0); end if;
     -- create input fault lists based on input vector

 common-entry(input(1),input(2),k);
 if t(input(1))=zero and t(input(2))=zero then
  if F(k) <> nil then merge-lists(output(3),k); end if;
 elsif t(input(1))=zero and t(input(2))=one then
  merge-lists(output(3),input (1));
  if F(k) <> nil then subtract-lists(output(3),k); end if;
 elsif t(input(1))=one and t(input(2)=zero then
  merge-lists(output (),input ());
 if F(k) <> nil then subtract-lists(output(3),k); end if;

 else merge-lists(output(3),input (1));
 merge-lists(output (3),input (2));
   -- create output fault list based on test vector,
   -- input fault lists, and functionality of the gate

  end if;
 end;
```

Figure 3: FAULT MODEL FOR AN AND GATE.

None of the F(input (1)) or F(input (2)) are merged into F(output (3)) because, for all faults that force an input to 1, the output is 0 – indistinguishable from the correct output and, therefore, these faults are not detectable at the output. However, if there are common entries between fault lists at input(1) and input(2), then, for each entry, both inputs will be forced to 1 causing a 1 at the output which is detectable. The output fault list will contain all common entries between F(input(1)) and F(input(2)). If input (1) and input (2) are 0 and 1 respectively, every record in F(input (1)) and F(input (2)) will force the two inputs to 1 and 0 respectively. Only the F(input (1)) is merged with F(output (3)), because, for all faults that force a 1 on input (1), the output is 1 – different from the correct output and, therefore, these faults are detectable at the output. However, all faults common between F(input(1)) and F(input(2)) must be eliminated from F(output(3)) because each of them will force a 0 on input(2), causing a 0 on the output which is undetectable.

When input (1) and input (2) are 1 and 0 respectively, the F(input (2)) is merged into F(output (3)) and the faults common between F(input(1)) and F(input(2)) are eliminated, by a similar reasoning. The fault model therefore determines the output fault list and the AND gate is fault simulated.

The above fault model consists of rules expressed through "if" clauses. These rules are but explicit calls to the three procedures, based on the Boolean relationship that defines the AND gate, and are executed based on the input vector. Fault models for other gates such as OR and NOT contain different sets of rules based on their respective Boolean relationships and executed depending on the input vectors. Fault models of an Inverter and OR gate may be derived analogous to the AND gate.

3. Functional Fault Simulation in RDV

In this section, the new approach is applied to functional devices and functional fault models are derived.

Definition of functional fault models in RDV for combinational blocks : Let CB denote a functional combinational component with n inputs I1,I2,..,In, and k outputs O1,O2,...,Ok, where Oj = Mj(I1,I2,..,In) for all j from 1 to k, and M1,...,Mk are Boolean mappings. A functional fault model in RDV is required to consider all faults in the fault lists, F(I1),...,F(In) including the stuck-at faults at I1,..,In, simulate them assuming unfaulted Boolean mappings M1,..,Mk, and propagate them to the output ports O1,...,Ok, for a given set of input test vectors.

Definition of functional fault models in RDV for sequential blocks : Let SB denote a functional sequential component with n inputs I1,I2,..,In, r states S1,...,Sr, and k outputs O1,O2,...,Ok. Oj = Mj(I1,I2,..,In,S1,..,Sr) for all j from 1 to k, and Sl_{new} = Nl(I1,I2,..,In,S1,..,Sr) for all l from 1 to r where, M1,..,Mk and N1,...,Nr are Boolean mappings. A functional fault model in RDV is required to consider all faults in the fault lists, F(I1),..., F(In) including the stuck-at faults at I1,..,In, and the internal stuck-at faults at S1,..,Sr, simulate them assuming unfaulted boolean mappings M1,.., Mk,N1,..,Nr and propagate them to the output ports O1,...,Ok, for a given set of input test vectors. A stuck-at fault associated with a state implies that the initialization is faulty.

The principal advantages of deriving functional fault models are :

- Because the CPU time required to simulate a circuit is generally a function of the number of devices simulated, it may imply considerable savings when fault simulation is carried out at a much higher – functional level, where a single device represents many gates.
- Functional fault models of flipflops and other sequential devices are more comprehensive than their gate-level equivalent.
- Typical digital designs contain a substantial number of functional blocks such as adders, counters, and multipliers.
- For some large circuits such as microprocessors, the manufacturer supplies the functional model only.

The principal disadvantage of functional fault models is loss of detail particularly with respect to internal faults. Section 3.1 presents a functional fault model of a one-bit adder and section 3.2 presents the functional fault model of a four-bit adder. Section 3.3 briefly presents the limitation of the decomposition principle.

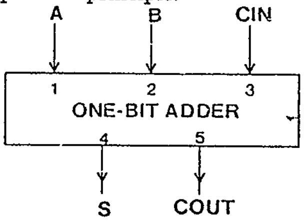

Figure 4: A ONE-BIT ADDER FUNCTIONAL REPRESENTATION.

The Boolean equations describing the functional block are :

S = CIN.(-A.-B + A.B) + -CIN.(-A.B + A.-B);

COUT = CIN.(A + B) + A.B;

For the one-bit adder shown in Figure 4, the functional fault model is derived assuming the general case that A, B, and CIN are secondary inputs. The boolean equations that define the adder are given below the Figure 4, where the symbols ".", "+", and "-" represent Boolean AND, OR, and NOT respectively. Therefore, fault lists F(A), F(B), and F(CIN) may be nonempty. Figure 5 contains the truth tables for COUT(carry out) and S(sum) outputs. For a given input set of values of Cin, A, and B, the COUT and S outputs may be obtained from the tables 1 and 2. Assuming CIN = 1, A = 0, and B = 1, the following fault lists are created by the procedure create-list and appended to the already existing lists. Therefore, F(CIN) = F(CIN) ∪ CIN(s-a-0), F(A) = F(A) ∪ A(s-a-1), and F(B) = F(B) ∪ B(s-a-0). For the input vector, the correct outputs are read from the tables 1 and 2 to be S = 1 and COUT = 0; therefore, the fault lists for these outputs are F(S) = S(s-a-0) an F(COUT) = COUT(s-a-1). For all faults in F(CIN), the equivalent faulty input vector is CIN = 0, A = 0, and B = 0 for which COUT is 0 (table 1) and S is 0 (table 2). The value of COUT is indistinguishable from the correct value; however, any fault common between F(A), F(B), and F(CIN) will be cause a 1 at COUT which is same as the good output. Therefore, F(COUT) = F(A) ∩ F(B) ∩ F(CIN). The value of S is different and, therefore, the faults in F(CIN) are potentially detectable at the output S. Hence, F(S) = F(S) ∪ F(CIN). All fault entries that are common between F(A) and F(CIN) will cause a 1 at the S output and hence must be eliminated. Therefore, F(S) = S(s-a-0) ∪ {F(CIN) - F(A) ∩ F(CIN)}. For all faults in F(A), the output values are CIN = 1 and S = 0. Both of these output values are different from the correct output and all faults in F(A) are detectable at both S and COUT. Therefore, F(S) = F(S) ∪ F(A), and F(COUT) = F(COUT) ∪ F(A). All fault entries common between F(A) and F(B) and between F(A) and F(CIN) must be eliminated from F(S) because they cause a 1 on the S output which is indistinguishable from the good output. The fault entries common between F(A) and F(B) and between F(A) an F(CIN) must also be eliminated from F(COUT) because they will force a 0 on the COUT output which is same as the good output. Therefore, $F(S) = F(S) \cup F(A) - F(A) \cap F(B) - F(A) \cap F(CIN)$, and $F(COUT) = F(COUT) \cup F(A) - F(A) \cap F(B) - F(A) \cap F(CIN)$. By a similar reasoning process, for all faults in F(B), the following final fault lists are obtained.

$$F(S) = S(\text{s-a-0}) \cup F(CIN) - F(A) \cap F(CIN) \cup F(A) - F(A) \cap F(B) - F(A) \cap F(CIN) \cup F(B) - F(A) \cap F(B).$$

$$F(COUT) = COUT(\text{s-a-1}) \cup F(A) \cap F(B) \cap F(CIN) \cup F(A) - F(A) \cap F(B) - F(A) \cap F(CIN).$$

The fault model (Figure 6) for the one-bit adder must produce output fault lists for all possible vectors at the inputs A, B, and CIN. In it, first, the input fault lists F(A), F(B), and F(CIN) are determined based on the given input vector. Then, the S and COUT are determined using the truth tables and based on these, the F(S) and F(COUT) are deduced. All faults in F(A), F(B), and F(CIN) correspond to respective faulty vector for which, the S and COUT outputs are determined again using the truth tables. If the value of S for all faults in F(A) differ from the correct value, then, the fault list F(A) is appended to F(S), because, they are potentially detectable at the S output. Similarly, where the value of COUT for all faults in F(A) differ from the good value, the fault list F(A) is added to F(COUT). When the fault lists F(A), F(B), and F(CIN) are being appended to F(S) and F(COUT), care is taken to isolate all faults that are common between the input fault lists. Each of these faults imply a faulty value on more than one input and, consequently, the output must be evaluated using the truth tables. Where such a fault causes an output that is identical to the good value, that fault must be eliminated from appropriate output fault list, because, it will not be detectable. The fault model terminates execution when the output fault lists are completely determined and are propagated to other fault models that are connected to its output port.

CIN	A	B	COUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 5: TRUTH TABLES FOR COUT AND S.

```
Task body one-bit-adder is
input  : array(1..3) of node;
output  : array(4..5) of node;
var
 ab,acin,bcin,abcin : integer;
function inv(a : intnet-val) return intnet-val is
begin ..... end;
function Table1(g,h,m:intnet-val) return intnet-val is
begin ....... end;
function Table2(g,h,m:intnet-val) return intnet-val is
begin ....... end;
begin
 common-entry(input(1),input(2),ab);
 common-entry(input(1),input(3),acin);
 common-entry(input(2),input(3),bcin);
 common-entry(input(1),input(2),abcin);
 common-entry(abcin,input(2),abcin);
 t(output(4)):=Table1(t(input(1)),t(input(2)),t(input(3)));
 t(output(5)):=Table2(t(input(1)),t(input(2)),t(input(3)));
 if t(input(1))=zero then create-list(input(1),s-a-1);
 else create-list(input(1),s-a-0); end if;
 if t(input(2))=zero then create-list(input(2),s-a-1);
 else create-list(input(2),s-a-0); end if;
 ...................................
 if t(output(5))=zero then create-list(output(5),s-a-1);
 else create-list(output (5),s-a-0); end if;
 if t(output(4)) <> table1(inv(t(input(1))),t(input(2),
  t(input(3)) then
  merge-lists(output(4),input(1));
 if F(ab) <> nil and T(output(4)) =
  table1(inv(t(input(1))),inv(t(input(2))),t(input(3)))
  then subtract-lists(output(4),ab); end if;
 if F(acin) <> nil and T(output(4)) =
  table(inv(t(input(1))),t(input(2)),t(inv(input(3))))
  then subtract-lists(output(4),acin); end if;
 if F(abcin) <> nil and T(output(4)) =
  table(inv(t(input(1))),inv(t(input(2))),inv(t(input(3))))
  then subtract-lists(output(4),abcin); end if;
 end if;
  ...................................
end;
```

Figure 6: FAULT MODEL FOR ONE-BIT ADDER.

3.2 Functional Fault Model of Four-Bit Adder

In the above functional fault model for a one-bit adder, the truth table size is a function of the number of inputs and outputs. Reading a table is equivalent to determining the outputs logically using the Boolean equations and with a rise in the number of inputs and outputs as in functional blocks, these equations become complex. This section describes methods of subdividing functional blocks into modules to simplify fault simulation.

Figure 7 is a functional diagram of a four-bit adder. The outputs S0 through S3 and C3 are functions of A0 through A3,B0 through B3, and Cin. The truth-table for nine inputs and five outputs would be prohibitively large and, equivalently, a set of Boolean equations with some of them involving nine variables may be quite complex. The fault model for such a component will require complex fault list manipulation which may be difficult.

A technique is introduced whereby the four-bit adder is functionally decomposed into four one-bit adders as shown in Figure 7. The four one-bit adders are identified by 1 through 4. The carry output from a previous one-bit adder is connected to the carry input of the subsequent one. For 1, the carry input is primary and for 4, the carry output is primary. 1 and 4 are the least - and the most significant adders respectively. The decomposition principle may be formally expressed as follows. Let C denote a circuit that may be structurally decomposed into n units, C = C_1,....,C_n such that, either each unit may be fault simulated completely independent or they must be fault simulated in sequence. In the latter case, intermediate results and fault lists are propagated from C_j to C_{j+1} immediately after the unit C_j has been fault simulated.

In the fault model of the four-bit adder, the four one-bit adders 1 through 4 are fault simulated sequentially using the truth table for a one-bit adder repeatedly. Because all three inputs A0, B0, and CIN of 1 are primary, they receive input vectors first and 1 is fault simulated. Results of simulation are fault lists F(S0) and F(C0), where S0 and C0 are the sum and intermediate carry outputs. Both output fault lists will be some function of the input fault lists F(A0), F(B0) and F(CIN). Therefore, F(S0) = fl(F(A0), F(B0), F(CIN)) and F(S0) = gl(F(A0), F(B0), F(CIN)), where fl and gl are mappings. Then, 2 is fault simulated and the output fault lists F(S1) and F(C1) are determined as some function of F(A1), F(B1), and F(C0). Therefore, F(S1) = f2(F(A1), F(B1), F(C0)) = f2(F(A1), F(B1), gl(F(A0), F(B0), F(CIN))) = $f2_1$(F(A1), F(B1), F(A0), F(B0), F(CIN)), where f2 and $f2_1$ are mappings. Similarly, F(C1) = $g2_1$(F(A1), F(B1), F(A0), F(B0), F(CIN)), where $g2_1$ is a mapping. Fault models for 3 and 4 are executed in order and finally, the fault lists F(S3) and F(C3) are obtained as follows. F(S3) = $f4_1$(F(A0), .., F(A4), F(B0),.., F(B4), F(CIN)) and F(C3) = $g4_1$(F(A0), .., F(A4), F(B0), .., F(B4), F(CIN)), where $f4_1$ and $g4_1$ are mappings.

Figure 8 contains the functional fault model of a four-bit adder. The input(1) through input(9) and output(10) through output(14) represent the nine inputs and five outputs respectively. The procedure p-fault-simulation corresponds to the model section for fault simulation. The procedure adderl is responsible for fault simulating a one-bit adder and has been detailed in an earlier section. The p-fault-simulation calls adderl four times; during each call, the input fault lists are passed to the adderl and the resulting output fault lists are sent back to the p-fault-simulation when adderl completes execution.

An Invariance Principle

While deriving the functional fault model for the adder, the implementation details of these functional blocks have either been ignored or a specific implementation have been assumed. It is reasonable, however, to assume that there could be more than one implementation or a different set of Boolean equations for the same functional block. Functional fault simulation in RDV will always yield the same result despite the implementation or Boolean equations chosen while deriving the fault model. Intuitively, because the fault lists obtained include stuck-at faults at only the input and output nodes, they should be unaffected by the internal implementation. A formal proof of this invariance principle, as applied to combinational and sequential circuits is detailed in [GS84].

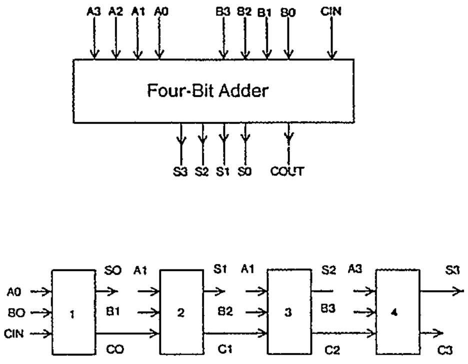

Figure 7: DECOMPOSED FOUR BIT ADDER.

```
-----------------------------------------------
|Task body four-bit-adder is                  |
| input: array(1..9) of node;                 |
| output: array(10..14) of node;              |
| procedure one-bit-adder(..);                |
|  begin .. end;                              |
|                                             |
| begin                                       |
|       -- fault simulate first one-bit adder |
|  one-bit-adder(...);                        |
|  .....                                      |
|       -- fault simulate fourth one-bit adder|
|  one-bit-adder(...);                        |
-----------------------------------------------
```

Figure 8: FAULT MODEL OF A FOUR-BIT ADDER.

3.3 Fault Models for other Functional Blocks and Limitation of the Decomposition Principle

Functional fault models for shift-registers, synchronous and asynchronous counters, memory elements, ALU units, and multipliers are derived and detailed in [GS84]. Models for multiplexers, combinational PLA's and other functional sequential blocks that may be represented by a truth table or a set of Boolean equations, may be derived using the decomposition principle and is not presented in this paper. For cases such as random logic and arbitrarily large PLA's whose Boolean equations may not be decomposed structurally, the fault model must contain the entire truth table and, therefore, may be complex.

4. Analysis of Results Obtained from Fault Simulation in RDV

This section reports the CPU times taken for fault simulating circuits at the gate- and functional-level level in RDV.

4.1 Fault Simulation of Adders

Figure 9 shows the normalized CPU times obtained from executing fault simulation on a four-, eight-, sixteen-, and thirty-two-bit adder. Because it is difficult to manually describe the interconnection database for a large gate-level circuit, the sixteen- and thirty-two-bit adder circuits are not simulated at the gate-level in RDV. However, they are simulated at the functional-level in RDV.

	Adder Size			
	4	8	16	32
RDV Gate-Level	35.6	54.28	-	-
RDV Functional-Level	6.8	9.9	11.4	17.6

Figure 9: CPU TIMES (IN SECONDS) FOR DIFFERENT FAULT SIMULATORS.

For each of these simulations, a set of 240 test vectors generated by a random 0,1 generator is used and the fault coverage is 100 percent. For the thirty-two-bit adder consisting of 336 gates, the total number of faults is 448 all of which are detected by the test vector set. It is observed in Figure 10 that the graphs are linear.

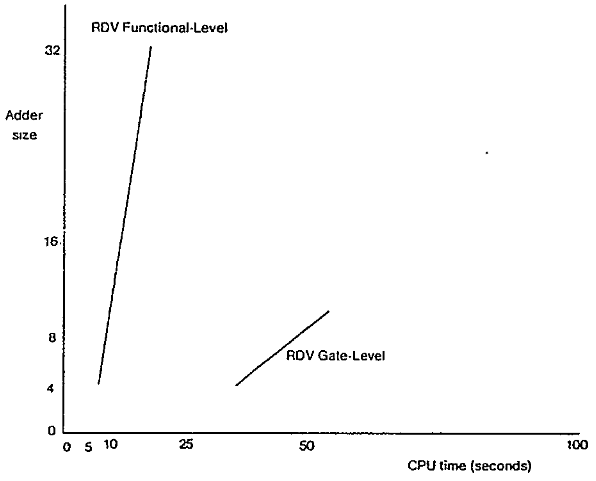

Figure 10: GRAPHS OF PERFORMANCE OF DIFFERENT FAULT SIMULATORS.

The linearity of the fault simulator in RDV may be explained as follows. Because each component in RDV is simulated only once per vector, and as all necessary computations are done within the component model, the total CPU time will be proportional to the total number of devices simulated.

It may be observed from Figure 9 that functional fault models are faster than their equivalent gate-level models. The functional fault models in RDV of the four- and eight-bit adders are faster than the corresponding gate-level models in RDV, by factors of 5.2 and 5.4 respectively. The ratio is likely to increase for fault models of functional devices that represent larger number of gates.

4.2 Fault Simulation of the AMD2903 in RDV

A simplified AMD2903 bit-slice processor is functionally fault simulated in RDV using the DEC-20 computer system. The simplified architecture and the functional fault models of the constituent components of the AMD2903 are given in [GS84]. A set of 240 input test vectors are used and the fault coverage achieved, is 62%. The CPU time taken is 29.6 seconds.

5. References

[AD72] D.B.Armstrong, "A Deductive method of Simulating Faults in Logic Circuits," IEEE Transactions on Computers, May 1972.

[AD81] D.C.Luckham, H.J.Larsen, D.R.Stevenson and F.Von Henke, "ADAM - An ADA Based Language for Multi-processing," Program Verification Group Report PVG-20, CSD Report STAN-CS-81-867, Stanford University, July 1981.

[AD83] U.S. Department of Defense, "Reference Manual for the Ada Programming Language," January 1983.

[BM76] M.A.Breuer and A.A.Friedman, "Diagnosis and Design of Digital Systems," Computer Science Press. Potomac. Maryland. 1976.

[CG78] G.R.Case and J.D.Stauffer, "SALOGS-IV A Program to Perform Logic Simulation and Fault Diagnosis," Proceedings of the 15th Design Automation Conference, Las Vegas, June 1978.

[dM80] M.A.d'Abreu, "An Accurate Functional Level Concurrent Fault Simulator," Proceedings of the 17th Design Automation Conference, 1980.

[HI83] Ibrahim Hajj and D.Saab, "Fault Modeling and Logic Simulation of MOS VLSI Circuits based on Logic Expression Extraction," ICCAD 1983.

[GS84] Sumit Ghosh, "RDV: A Rule-Based Generalized Design Verifier," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, CA - 94305, 1984.

[HS83] S. Hirschhorn, "Fault Propagation Techniques in a Functional Level Concurrent Fault Simulator," ICCAD 1983.

[MA76] A.Miara etal, "Fault Propagation in digital networks using a deductive analysis," CAD Seminar, Budapest, 1976.

[PM83] P.R. Moorby, "Fault Simulation Using Parallel Value Lists," ICCAD 83.

[SS65] S.Seshu, "On An Improved Diagnostic Program," IEEE Transactions on Electronic Computers, Vol EC-14, 1965.

[SS72] Szygenda, S.A. and Thompson, E.W., "Fault Insertion Techniques and Models for Digital Logic Simulation," FJCC 1972.

[SS73] S.A. Szygenda and A.A. Lekkos, "Integrated Techniques for Functional and Gate Level Digital Logic Simulation," Proceedings of the 10th Design Automation Workshop,June 1973. pp 159-172.

AUTHOR'S BIOGRAPHY

Sumit Ghosh graduated from the Indian Institute of Technology, Kanpur in May 1980 with a B.Tech. degree in Electrical Engineering and the received the M.S. and Ph.D. degrees from Stanford University, California, in 1981 and 1984 respectively. While at Stanford, he was affiliated with the Computer Systems Laboratory and the Center for Integrated Systems. Sumit then joined the Knowledge Systems Research Department at Bell Laboratories as a Principal Investigator-Member of Technical Staff. His interests are in computer-aided design of digital systems and VLSI, distributed expert systems for design verification and synthesis, behavior-level fault modeling, multiprocessor architecture, and hardware description languages.

Dr. Sumit Ghosh
AT&T Bell Laboratories Research
Room 4G-634, Holmdel, NJ 07733, USA.
Tel. 201-949-8689.