1986

Fault simulation using binary decision diagrams /

William Bader
Lehigh University

FAULT SIMULATION USING BINARY DECISION DIAGRAMS

by

William Bader

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

September 20, 1986

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science

9/22/86

(date)

[illegible]

Professor in Charge

[illegible]

Chairman of Department

CONTENTS

Fault Simulation

using

Binary Decision Diagrams

Abstract

This paper describes an algorithm for performing fault simulation on binary decision diagrams with a computational complexity similar to that of good fault simulation algorithms for gate networks. The algorithm can use fault dropping and can handle multiple observable outputs in a single pass, so that substantial reductions in simulation are obtained, as was confirmed in trail runs of a limited number of test cases.

Fault Simulation

using

Binary Decision Diagrams

1 Introduction

1.1 BDD Definitions

A binary decision diagram, or BDD, is a type of acyclic directed graph that can be used to determine the value of a boolean function [1]. The graph is constructed from nodes and edges. Constant nodes have an out degree of 0. All other nodes have an out degree of exactly 2 and contain the name of a variable. I will call these nodes variable nodes or decision nodes to distinguish them from constant nodes. The outward edges are called the exits of the node. In particular, one edge is called the 0-exit and the other is called the 1-exit. Unless otherwise labelled, the left edge is the 0-exit. Exactly one node in a BDD has an in degree of 0. This node is called the <u>entrance</u> <u>node</u> or <u>root</u> of the BDD. To determine the value of the function represented by a BDD for a given input, start at the entrance node and work downward through the nodes. If the node is a constant node, the value of the function is the value of that node. Otherwise, take the 0-exit if the value of the node's variable is false or the 1-exit if it is true. Figure 1 shows a BDD with its parts labelled. The numbers in parentheses are node numbers that correspond to numbers in a later example.

Figure 1. A BDD

```
                          |
                        -----
                  (1)   | A |  <--- entrance node
                        -----
                         /   \
        0-exit --->     /     |  <--- 1-exit
                    -----     |
               (2)  | B |     |
                    -----     |
                     /   \    |
                    /     \   |
              (3) 0   (4)  1     <--- constant node
```

1.2 BDD Data Structures

The input to the BDD fault simulation algorithm is a set of BDD's. We will call each diagram in the set of BDD's a <u>graph</u> and the entire set of graphs a <u>network</u> to avoid confusion between a single diagram and the entire set of diagrams. The BDD fault simulation algorithm requires two types of data structures: graphs and nodes. Each graph contains a graph descriptor, a pointer to a node, and a graph value. Each node contains a left node (0-exit) pointer, a right node (1-exit) pointer, a variable pointer, and a node value. There are two types of graphs: input graphs and output graphs. Each input graph corresponds to a primary input to the network. The value of an input graph is the current value of the input it represents. Each output graph corresponds to a BDD and has a pointer to the entry node of the BDD. The value of a BDD is the same as the value of its entry node.

There are two types of nodes: constant nodes and variable nodes. Constant nodes have a fixed value of either 0 or 1. The value of a variable node depends on the value of the graph that represents its variable and on the values of the nodes at its exits. If the value of the graph is 0, then the value of the node is the value of the node pointed to by the node's left pointer; otherwise, the value of the node is the value pointed to by the node's right pointer. The data structure for the BDD from the previous example is shown with comments in Figure 2. The node values correspond to an input with A set to 0 and B set to 1.

Figure 2. Data Structures for a BDD

Graph	Descriptor	Value	Node Pointer	Comments
1	input	0	-	variable A
2	input	1	-	variable B
3	output	0	1	entrance of a BDD

Node	0-exit	1-exit	Variable	Value	Comments
1	2	4	1	0	entrance node depends on A
2	3	4	2	1	depends on B
3	-	-	0	0	constant 0
4	-	-	1	1	constant 1

When a BDD is excited with an input, either the left or the right branch of each node is <u>activated</u>. Among the set of activated branches, there is a unique path from the entrance node of the BDD to exactly one of its exit nodes. The path is called the <u>active</u> <u>path</u> for that input.

It is possible to simulate a BDD network by simulating every node from the constant nodes to the root node in much the same manner that gate networks must be simulated gate by gate. However, since the value of a BDD only depends on the value of the constant node at the end of the active path, it is much more efficient to simulate only the active path by starting at the entrance node and working towards the exit node. Note that whenever a decision node depends on the value of an output graph, that graph must be evaluated before making the decision on the node.

1.3 Pseudo-Code Notation

We will describe our gate conversion algorithm and several versions of our fault simulation algorithm in a Pascal-like pseudo-code. Each procedure will be declared as

```
procedure example(exampleargument);
   begin
      do something
   end;
```

Comments will be delimited by braces, for example

```
{ this is a sample comment }.
```

IF statements have the syntax

```
if some condition then
   do action A1
   do action A2
else
   do action B1
   do action B2
```

while FOR statements have the syntax

```
for each item in a set do
   do something with the item
```

Note that IF and FOR statements can be nested inside each other and that statements in the body of another statement are marked by their indentation. As in Pascal, we will use square brackets to denote array subscripts and periods (".") to reference elements in structures. The declarations of the variables that we will use in pseudo-code to represent BDD networks are shown in Figure 3. For example, Nodes is a data structure that represents all the nodes in a BDD network; Nodes[5] represents node number 5, and Nodes[5].LeftNode represents a pointer to the 0-exit of node number 5.

Figure 3. Declarations

```
Nodes: array [1 .. MaximumNumberOfNodes] of
         record
           Value { value of this node,
                   0, 1, or unknown }
           Critical { whether this node is critical,
                      Yes, No, Unknown }
           Graph { pointer to the graph that
                   contains this node }
           DecisionGraph { pointer to the graph that
                           represents the nodes's
                            variable }
           LeftNode { pointer to the 0 exit
                      of this node }
           RightNode { pointer to the 1 exit
                       of this node }
         end;
Graphs: array [1 .. MaximumNumberOfGraphs] of
          record
            GraphType { InputGraph, OutputGraph, or
                        ObservableOutputGraph }
            Value { value of this graphs,
                    0, 1, or unknown }
            Critical { whether this graph is critical,
                       Yes, No, Unknown }
            EntranceNode { pointer to entrance node of
                           this graph }
          end;
```

2 Conversion of Gate Networks to BDD's

2.1 Goals of the Conversion Algorithm

When doing fault analysis on gate networks, the stuck-at fault model is typically used. In this model, a fault causes a line to be stuck at either 0 or 1. A fanout branch can be stuck without having its stem or other branches stuck. So that we may do fault simulation on a BDD network, we must be able to convert the gate networks into BDD networks in such a manner that faults in the original gate network can be mapped into faults in the BDD network. To be completely general, the conversion algorithm must also be able to handle any combinational circuit, including circuits with multiple outputs.

In order to maintain the correspondence between faults, BDD networks are created in an algorithmic fashion such that each BDD node corresponds to an input lead in the gate network and each graph corresponds to a primary input, fanout stem or observable output. In the BDD, gate inputs are represented by nodes, and the value of each input is represented by the value of the graph corresponding to the node's variable. Thus, stuck-at-0 or 1 faults in a gate network correspond to stuck-left or right faults in a node. This was first described in [9].

2.2 The Conversion Algorithm

The process of BDD construction starts by creating an input graph for each primary input in the gate network. Then, the gates in the gate network are examined in order by increasing level. For each gate, if the first input is a primary input, a fanout stem, or an observable output, then a decision node is created with the graph corresponding to that line as the decision variable. The new node is labelled with the line number of the input. Otherwise, the input must be the output of a gate with a fanout of exactly 1. Since the gates are processed by increasing level, a BDD was already created for this gate in an earlier step. This BDD will now be used to represent the input line of the current gate. The same is done for all the other inputs. If the gate is an OR gate or a NOR gate, the new input is combined with the existing BDD by replacing all 0 exits of the existing BDD with the entrance node of the BDD for the new input. If the gate is an AND gate or a NAND gate, the new input is combined by replacing all 1 exits of the existing BDD with the entrance node of the BDD for the new input. After all the inputs are processed, if the gate is an INVERTER, a NOR gate or a NAND gate, all the 0 and 1 constant nodes of the BDD are complemented. If the gate output is a fanout stem or an observable line, a graph entry is made for it. An example of a gate network is shown in Figure the resulting BDD network is shown in Figure 5, and its data structure is shown in Figure 6. The values of the nodes in the data structure correspond to an input of 0 for A and 1 for B. A more precise statement of the algorithm follows in Figure 7. Note that in the statement of the algorithm, we treat primary inputs as a type of gate just as we treat them as a type of graph in BDD networks.

Figure 4. A Gate Network

```
  a.O                   d.1 ------
A---------------------------|    | d.O
     |                      |NAND|------
     | c.1 ------      ------|    |     |     ------
     ------|    | c.O | d.2 ------      ------|    |
           |NAND|------                      |NAND|------F
     ------|    |     | e.1 ------      ------|    |
     | c.2 ------      ------|    |     |     ------
     |                      |NAND|------
B---------------------------|    | e.O
  b.O                   e.2 ------
```

Figure 5. Resulting Binary Decision Diagrams

```
BDD 1   BDD 2     BDD 3     BDD 4         BDD 5
  |        |         |         |             |
  A        B      ------     ------        ------
              (3) |BDD1| (4) |BDD2|    (5) |BDD3|
                  |a.0 |     |b.0 |        |c.1 |
                  ------     ------        ------
                  /    \     /    \        |    \
             (1) 0  (2) 1   0      1       |   ------
                                           |(6)|BDD4|
                                           |   |c.2 |
                                           |   ------
                                           |  /     \
                                            1        0

           BDD 6
             |
           ------
       (7) |BDD3|
           |d.1 |
           ------
           /    \
           |   ------
           |(8)|BDD5|
           |   |d.2 |
           |   ------
           |  /     \
        ------       \
    (9) |BDD5|        \
        |e.1 |         \
        ------          |
        |     \         |
        |      ------   |
        | (10) |BDD6|   |
        |      |e.2 |   |
        |      ------   |
        |     /      \  |
          1            0
```

Figure 6. Resulting Data Structure

Graph	Descriptor	Value	Node Pointer	Comment
1	input	0	-	variable A
2	input	1	-	variable B
3	output	0	3	A
4	output	1	4	B
5	output	1	5	output of gate C
6	obs. output	1	7	output of network

Node	0-exit	1-exit	Variable	Graph	Value	Comment
1	-	-	0	-	0	constant 0
2	-	-	1	-	1	constant 1
3	1	2	1	3	0	A
4	1	2	2	4	1	B
5	2	6	3	5	1	C.1
6	2	1	4	5	0	C.2
7	9	8	3	6	1	D.1
8	9	2	5	6	1	D.2
9	1	10	5	6	1	E.1
10	1	2	4	6	1	E.2

Figure 7. Conversion Algorithm

```
for each primary input do
  create an input graph

for each gate in order of increasing level do
  if it is a primary input then
    create a node with the input as the decision point
    label the node with the line number of the input
  else if the gate is an inverter then
    if its input is represented by a graph then
      create a node with the graph as the decision point
      label the node with the line number
        of the inverter's input
    else
      take the partial BDD for the input
  else if the gate is an AND gate or a NAND gate then
    for each fanin do
      if its input is represented by a graph then
        create a node with the graph as the decision point
        label the node with the line number
          of the current input
      else
        take the partial BDD for the input
      if this is the first fanin then
        make the node or input BDD the current partial BDD
      else
        set the 1 exits of the current partial BDD to be the
          input BDD
  else if the gate is an OR gate or a NOR gate then
    for each fanin do
      if its input is represented by a graph then
        create a node with the graph as the decision point
        label the node with the line number
          of the current input
      else
        take the partial BDD for the input
      if this is the first fanin then
        make the node or input BDD the current partial BDD
      else
        set the 0 exits of the current partial BDD
          to be the input BDD
  if the gate is an INVERTER, a NAND or a NOR then
    complement the values of all the constant nodes
  if the gate is a primary input or
     the output of the gate is a fanout stem or
     the output of the gate is observable then
    create a graph the for the constructed BDD and mark
      the gate with the index of the graph
  else
    mark the gate with the index of the entrance node
      of the constructed BDD
```

2.3 Modification to the Conversion Algorithm

For reasons that will be apparent later, two minor modifications to the conversion algorithm allow it to produce BDD's that require slightly less overhead for fault simulation. First, when the gate network is levelized, a gate may be assigned to level L with an output that is required only by gates at level L+K or higher, where K > 1. If the output of the gate is not observable, then the level of the gate is increased to L+K-1. If there happens to be an output between levels L+K and L+K-2, no overhead will be required for the changed gate. For similar reasons, whenever a gate with an observable output and a gate with a nonobservable output have the same level, the gate with the observable output is assigned to a BDD first.

3 The BDD Fault Simulation Algorithm

3.1 Basic Fault Simulation Algorithm

Given a BDD network and a set of values for the input graphs, the goal of fault simulation is to find the set of critical nodes. A critical node is a variable node such that switching the node's input from its left exit to its right exit would change the value of an observable graph. Switching the exits corresponds to provoking a fault in a gate network, while changing an observable graph corresponds to propagating a fault in a gate network. To tell if a node is critical according to the definition of a critical node, we could mark the node as stuck and resimulate the network. We will take another approach. To test if a node is critical, we will replace the node with a constant node that has a value complementary to the good-simulation value of original node. For the node to be critical, two conditions must hold: I) at least one observable graph in the new network has a different value, and II) the value of the node pointed to by the left exit of the original node differs from the value of the node pointed to the right exit of the original node. Condition I tests if the fault will propagate to an observable output if it is provoked, while condition II tests if the fault really is provoked. Note that a necessary, but not sufficient, condition for I) to hold is that the node be on the active path of its BDD. Clearly changing a node not on the active path of its graph will never change the graph so being on the active path is necessary, while changing a graph in a BDD network will change nodes in all the other graphs that depend on the changed graph but the net result of the changes is as unpredictable as changing a fanout stem in a gate network, so being on an active path is not sufficient.

The two conditions suggest the algorithm shown in Figure 8. This algorithm examines each node on the active path. To test if a node is critical, it cuts the node out of the graph and replaces it with a constant node with the complement of the value of the node. The altered network is then simulated. Note that although the value of the graph that contains the node will change, the values of other graphs may not. If the value of an observable graph changes, the other exit of the current node is simulated as if it were an entrance node of a graph. If this exit has a value different from the exit that was taken during the good simulation, the node is marked as critical.

Figure 8. Basic Algorithm

```
{ EvaluateNode - evaluate the value of a node }

procedure EvaluateNode(node);
begin
   if the node has not been evaluated then
     if the node is a constant 0 node then
       Nodes[node].Value := 0
     else if the node is a constant 1 node then
       Nodes[node].Value := 1
     else { if the node is a decision node then }
       EvaluateGraph( Nodes[node].DecisionGraph )
       if Graphs[ Nodes[node].DecisionGraph ].Value
            = 0 then
         EvaluateNode( Nodes[node].LeftNode )
         Nodes[node].Value :=
           Nodes[ Nodes[nodes].LeftNode ].Value
       else
         Evaluate( Nodes[node].RightNode )
         Nodes[node].Value :=
           Nodes[ Nodes[node].RightNode ].Value
end;

{ EvaluateGraph - evaluate the value of a graph }

procedure EvaluateGraph(graph);
begin
  if the graph has not been evaluated then
    EvaluateNode(Graphs[graph].EntranceNode)
    Graphs[graph].Value :=
      Nodes[ Graphs[graph].EntranceNode ].Value
end;
```

```
{ FindCritical - find all critical nodes }

procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  for each node do
    if the node is on an active path then
      { test for condition I }
      flip the value of the node
      clear the value of all graphs and all other nodes
        { the flip and clear effectively replaces the node
          with a constant node with a complemented value }
      evaluate the observable graphs
        {i.e. a faulty simulation}
      if the value changes {condition I is met} then
        { test for condition II }
        evaluate the exit not taken during
          the good simulation
        if it has a different value then
          {condition II is met}
          mark the node as critical
end;
```

3.2 Modification I: Saving Good Values

Note that when the other exit of a node is evaluated while testing for condition II, the simulation can use values already computed by the good simulation. FindCritical is revised as shown in Figure 9:

Figure 9. Algorithm after Modification I.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  for each node do
    if the node is on an active path then
      flip the value of the node
      clear the value of all graphs and all the other nodes
      evaluate the observable graphs
        {i.e. a faulty simulation}
      if the value changes {condition I is met} then
        restore good values to all nodes and graphs{*new*}
        evaluate the exit not taken during
          the good simulation
        save good values                         { *** new *** }
        if it has a different value then
          {condition II is met}
          mark the node as critical
end;
```

3.3 Modification II: Critical Graphs

Observe that for a given graph, either all sensitive nodes meet condition I or they all fail condition I. A graph is called critical if and only if it has a sensitive node that meets condition I. This means that simulation from the observable graphs need only be done once per graph instead of once per sensitive node. FindCritical may be altered as shown below in Figure 10:

Figure 10. Algorithm after Modification II.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  set Critical to Unknown for each graph { *** new *** }
  for each node do
    if the node is on an active path then
      if Graphs[ Nodes[node].Graph ].Critical is
          Unknown then { * new * }
        flip the value of the graph
        clear the value of all graphs and all other nodes
        evaluate the observable graphs
            {i.e. a faulty simulation}
        if an observable graph changed then { ** new ** }
          set Graphs[ Nodes[node].Graph ].Critical := Yes
        else
          set Graphs[ Nodes[node].Graph ].Critical := No
      if Graphs[ Nodes[node].Graph ].Critical = Yes then
          { * new * }
        restore good values to all nodes and graphs
        evaluate the exit not taken during
            the good simulation
        save good values
        if it has a different value then
          {condition II is met}
          mark the node as critical
end;
```

3.4 Modification III: Levelized Graphs

This modification and the following modifications require that the graphs and nodes be in a level organized form. Level organized form means that nodes in a graph do not use any higher graphs as decision variables and that all the nodes for a graph follow the nodes of all lower graphs. Since our conversion algorithm examines the gate network in order by increasing level, the BDD networks it produces have their graphs in order by increasing level also. The nodes, however, are not always in the proper order. The reordering procedure shown in Figure 11 collects the nodes for each graph. The values in the array NodeMap contain the new index for each node. When the conversion program writes out the BDD data structure, all it must do is write the value of NodeMap[n] instead of n whenever n is a pointer to a node.

Figure 11. Procedure to Reorder Nodes.

```
{ ReOrder -
    create a mapping to put the nodes in levelized order }

procedure ReOrder;
begin
  i := 0
  for g := 1 to NumberOfGraphs do
    for n := 1 to NumNodes do
      if Node[n].Graph = g then
        i := i + 1
        NodeMap[i] := n
end;
```

Since a change in a node in graph I cannot effect graph J, (for J less than I), the good values for these graphs do not have to be cleared. Thus, FindCritical can be revised again as shown in Figure 12.

Figure 12. Algorithm after Modification III.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
      {i.e. do a good simulation}
  set Critical to Unknown for each graph
  for each node do
    if the node is on an active path then
      if Graphs[ Nodes[node].Graph ].Critical is
          Unknown then
        flip the value of the graph
        clear the value and nodes of this graph and
          all higher graphs    { *** new *** }
        evaluate the observable graphs
            {i.e. a faulty simulation}
        if an observable graph changed then
          set Graphs[ Nodes[node].Graph ].Critical := Yes
        else
          set Graphs[ Nodes[node].Graph ].Critical := No
      if Graphs[ Nodes[node].Graph ].Critical = Yes then
        restore good values to all nodes and graphs
        evaluate the exit not taken during
            the good simulation
        save good values
        if it has a different value then
          {condition II is met}
          mark the node as critical
end;
```

3.5 Modification IV: Change Sets

When a given graph changes, decision nodes in other graphs may be affected. If no sensitive nodes in the affected graph are changed, the value of the graph will not be changed. Just as in a gate network, one can envision the effects of a fault propagating through the graphs in a BDD network. Instead of blindly simulating the entire network to determine if condition I holds for a particular fault, we will maintain a list of graphs on the propagation frontier. A graph is on the <u>propagation frontier</u> if no faulty simulation has been done for it and at least one of its nodes uses as its variable a graph that was changed due to the fault. Initially, the list will be set to contain all graphs with a node that uses the graph to be tested as its variable. From then on, the lowest level graph in the list is removed from the list and simulated. If the faulty value of the graph differs from the good value, all graphs with a node that uses the just simulated graph as its variable are added to the list. This process continues until either 1) a graph that is an observable output changes, in which case the original graph is critical, or the list is empty, in which case the original graph is not critical. As a speed up, note that if the the list ever contains exactly one graph and that graph changes then the original graph is critical if and only if the new graph is critical. This suggests that the nodes should be checked from highest to lowest. With this in mind, FindCritical can be recoded as shown in Figure 13. Note that this process parallels methods used to calculate the D-cube c(T,F) for a test T and a fault F in gate networks. For example, a proof that the speed up is valid is given as Lemma A in [5].

Use of change sets also has the desirable side effect that multiple outputs can be traced at once. The only differences between tracing from a single output and tracing from multiple outputs are that an extra good simulation must be done for for each additional output and that more graphs are initially labelled critical. Even if N outputs share a common graph, that graph will only be evaluated once, so the time to simulate all N outputs will be much less than N times the amount of time needed to simulate the first output. Thus, the extra cost of cost of simulating several outputs at once depends more on the number of unique graphs on critical paths from all the outputs than on the number of outputs.

Figure 13. Algorithm after Modification IV.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
      {i.e. do a good simulation}
  set Critical to Yes for each observable graph
    and Unknown for all other graphs { *** new *** }
  for each node from highest to lowest do { *** new *** }
    if the node is on an active path then
      if Graphs[ Nodes[node].Graph ].Critical is
          Unknown then
        flip the value of the graph
        GraphsToCheck := (all graphs that have a decision
                          node that depends on
                          Nodes[node].Graph)    {* new *}
        while (GraphsToCheck not empty) and
            (Critical = Unknown) do        { *** new *** }
          g := lowest graph in GraphsToCheck
          remove g from GraphsToCheck
          clear all nodes in graph g
          EvaluateGraph(g)
          if graph g changes then
            if (GraphsToCheck is empty) and
               (Graphs[g].Critical <> Unknown) then
              Critical := Graphs[g].Critical
            else if graph g is observable
              Critical := Yes
              clear GraphsToCheck
            else
              GraphsToCheck := GraphsToCheck +
                (all graphs that have a decision node that
                 depends on g)
        if Critical = Unknown then Critical := No
      if Graphs[ Nodes[node].Graph ].Critical = Yes then
        restore good values to all nodes and graphs
        evaluate the exit not taken during
            the good simulation
        save good values
        if it has a different value then
            {condition II is met}
          mark the node as critical
end;
```

3.6 Modification V: Graph Jumping

The previous change reduced the work needed to tell if a graph is critical from evaluating most of the graphs from the observable output down to evaluating only graphs on the path to the observable output. However, when adding graphs to the list, it treats all nodes identically. The goal of this modification is to maintain information about the types of nodes that have changed in a graph. First, note that the graph will not change unless at least one sensitive node changes. Thus, we will only enter a graph onto the list when a sensitive node on that graph is affected.

Given set of nodes in a graph (including at least one sensitive node), we want to tell if the graph's output will be changed if the given nodes all use their other exit. Suppose exactly one node changes on a critical graph. Then the graph changes if and only if condition II holds for the node, i.e. the node is on the sensitive path and the exits of the node have different values. Since the graph is critical, all of its nodes were checked for meeting condition II as part of the test for criticality, so this information is already known and requires no extra work to calculate. To generalize this case, note that if the other exit of a node has the same value as the graph, the node cannot affect the graph, while if the other exit has a different value than the graph, the node may affect the graph. For example, if the value of the other exit of all the affected nodes is identical to the value of the graph, the graph will remain unchanged. By the same token, if the graph is on the list and if the value of the other exit of all the affected nodes differs from the value of the graph, then the graph will change. If some mixture of these two types of nodes changes, the graph must still be simulated. Sometimes the value of the graph or of the other exit of a node is not known. In this case, the graph must be simulated also. Figure 14 shows the results of this modification.

Figure 14. Algorithm after Modification V.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  set Critical to Yes for each observable graph
    and No for all other graphs
  for each node from highest to lowest do
    if the node is on an active path then
      if Graphs[ Nodes[node].Graph ].Critical is
          Unknown then
        clear CriticalCount, NonCriticalCount, and
            UnknownCount values for each graph { * new * }
        flip the value of the graph
        for every decision node that depends on
            Nodes[node].Graph do
          if the node is on an active path then
            add its graph to GraphsToCheck
          if the other exit of the node is unknown or
             the value of the graph is unknown then {new}
            increment UnknownCount for its graph
          else if the other exit differs from the
        value of the graph then   { *** new *** }
      increment CriticalCount for its graph
    else if the other exit is identical to the
        value of then graph then   { *** new *** }
      increment NonCriticalCount for its graph
  while (GraphsToCheck not empty) and
      (Critical = Unknown) do
    g := lowest graph in GraphsToCheck
    remove g from GraphsToCheck
    if CriticalCount[g] = 1 and
        NonCriticalCount[g] = 0 and
        UnknownCount[g] = 0 then
      changed := Yes
      flip Graphs[g].Value
    else if CriticalCount[g] = 0 and
        NonCriticalCount[g] = 1 and
        UnknownCount[g] = 0 then
      changed := No
    else
      clear all nodes in graph g
      EvaluateGraph(g)
      changed := (Graphs[g].Value changed)
    if changed then
      if (GraphsToCheck is empty) and
         (Graphs[g].Critical <> Unknown) then
        Critical := Graphs[g].Critical
      else if graph g is observable
        Critical := Yes
        clear GraphsToCheck
      else
        for every decision node that depends on
            Nodes[node].Graph do  { * new * }
          if the node is on an active path then
            add its graph to GraphsToCheck
          if the other exit of the node is unknown or
              the value of the graph is unknown then
            increment UnknownCount for its graph{new}
          else if the other exit differs from the
              value of the graph then   { * new * }
            increment CriticalCount for its graph
          else if the other exit is identical to the
              value of then graph then { * new * }
            increment NonCriticalCount for its graph
  if Critical = Unknown then Critical := No
if Graphs[ Nodes[node].Graph ].Critical = Yes then
  restore good values to all nodes and graphs
  evaluate the exit not taken during
      the good simulation
  save good values
  if it has a different value then
    {condition II is met}
    mark the node as critical
```

```
end;
```

3.7 Modification VI: Stem Approximation

The previous version of the algorithm does a faulty simulation for every graph evaluated during the good simulation. Each graph corresponds to a fanout stem in the gate network. Abramovici, Menon and Miller (AM&M) [2] have discovered that fanout stems are rarely critical if none of their branches are critical. They claim that

> the approximation occurs seldom and consists in not marking as detected some faults that are actually detected in the evaluated set. ... This approximation does not affect the usefulness of the method.

At worst, this approximation is slightly pessimistic about the coverage of a test set. Even if a critical stem is not marked as critical for one test vector because none of its branches are critical, there will usually be another test vector with both the stem and some of its branches critical. Thus, when determining the coverage of a test set as a whole, this approximation has the desirable effect of reducing the number of faulty simulations without loosing much accuracy. Figure 15 shows the algorithm after this modification.

Figure 15. Algorithm after Modification VI.

```
procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  set Critical to Yes for each observable graph
    and No for all other graphs
  clear list of stems to check   { *** new *** }
  for each node from highest to lowest do
    if the node is on an active path then
      if Graphs[ Nodes[node].Graph ].Critical is
          Unknown then
       if Nodes[node].Graph not in list of
           stems to check then
        Graphs[Nodes[node].Graph].Critical := No { new }
       else
        clear CriticalCount, NonCriticalCount, and
            UnknownCount values for each graph
        flip the value of the graph
        for every decision node that depends on
            Nodes[node].Graph do
          if the node is on an active path then
            add its graph to GraphsToCheck
          if the other exit of the node is unknown or
              the value of the graph is unknown then
            increment UnknownCount for its graph
          else if the other exit differs from the
                value of the graph then
            increment CriticalCount for its graph
          else if the other exit is identical to the
                value of then graph then
            increment NonCriticalCount for its graph
        while (GraphsToCheck not empty) and
                (Critical = Unknown) do
          g := lowest graph in GraphsToCheck
          remove g from GraphsToCheck
          if CriticalCount[g] = 1 and
              NonCriticalCount[g] = 0 and
              UnknownCount[g] = 0 then
            changed := Yes
            flip Graphs[g].Value
          else if CriticalCount[g] = 0 and
                  NonCriticalCount[g] = 1 and
                  UnknownCount[g] = 0 then
            changed := No
          else
            clear all nodes in graph g
            EvaluateGraph(g)
            changed := (Graphs[g].Value changed)
          if changed then
            if (GraphsToCheck is empty) and
               (Graphs[g].Critical <> Unknown) then
              Critical := Graphs[g].Critical
            else if graph g is observable
              Critical := Yes
              clear GraphsToCheck
            else
              for every decision node that depends on
                  Nodes[node].Graph do
                if the node is on an active path then
                  add its graph to GraphsToCheck
                if the other exit of the node is unknown or
                    the value of the graph is unknown then
                  increment UnknownCount for its graph
                else if the other exit differs from the
                    value of the graph then
                  increment CriticalCount for its graph
                else if the other exit is identical to the
                    value of then graph then
                  increment NonCriticalCount for its graph
        if Critical = Unknown then Critical := No
      if Graphs[ Nodes[node].Graph ].Critical = Yes then
        restore good values to all nodes and graphs
        evaluate the exit not taken during
            the good simulation
        save good values
        if it has a different value then
          {condition II is met}
          mark the node as critical
          enter Nodes[node].DecisionGraph in
            list of stems to check    { *** new *** }
end;
```

3.8 Modification VII: Fault Dropping

When determining the fault coverage of a set of test vectors but the exact coverage of each individual vector is not required, we do not have to simulate faults for one test that already have been detected by another test. Skipping these faults is called <u>fault dropping</u>. Since only the remaining faults must be simulated for each vector, fault dropping reduces the time spent per vector. Figure 16 shows the algorithm with fault dropping. In order to keep track of which nodes have been tested for which faults, we have added a new field in each node that tells which faults have been tested and a new procedure Initialize that must be called once for each test set (while FindCritical is called once for each test vector) to initialize the new field. Note that the stem approximation cannot be made when doing fault dropping because stem faults will not be detected unless the stem happens to be critical on the first test for at least one branch that detects a fault on that branch.

Figure 16. Algorithm after Modification VII.

```
Nodes: array [1 .. MaximumNumberOfNodes] of
         record
           { same as before plus }
           TestedForStuck: array [0 .. 1] of
               { Yes, or No}
         end;

procedure Initialize;
begin
  for each node do
    set TestedForStuck[0] to No
    set TestedForStuck[1] to No
end;

procedure FindCritical;
begin
  clear the value of all nodes
  evaluate the observable graphs
    {i.e. do a good simulation}
  set Critical to Yes for each observable graph
    and No for all other graphs
  for each node from highest to lowest do
    if the node is on an active path then
     if TestedForStuck[ Graphs[Nodes[node].Graph].Value ]
    = No then   { *** new *** }
if Graphs[ Nodes[node].Graph ].Critical is
    Unknown then
  clear CriticalCount, NonCriticalCount, and
      UnknownCount values for each graph
  flip the value of the graph
  for every decision node that depends on
      Nodes[node].Graph do
    if the node is on an active path then
      add its graph to GraphsToCheck
    if the other exit of the node is unknown or
       the value of the graph is unknown then
      increment UnknownCount for its graph
    else if the other exit differs from the
        value of the graph then
      increment CriticalCount for its graph
    else if the other exit is identical to the
        value of then graph then
      increment NonCriticalCount for its graph
  while (GraphsToCheck not empty) and
      (Critical = Unknown) do
    g := lowest graph in GraphsToCheck
    remove g from GraphsToCheck
    if CriticalCount[g] = 1 and
        NonCriticalCount[g] = 0 and
        UnknownCount[g] = 0 then
      changed := Yes
      flip Graphs[g].Value
    else if CriticalCount[g] = 0 and
        NonCriticalCount[g] = 1 and
        UnknownCount[g] = 0 then
      changed := No
    else
      clear all nodes in graph g
      EvaluateGraph(g)
      changed := (Graphs[g].Value changed)
    if changed then
      if (GraphsToCheck is empty) and
         (Graphs[g].Critical <> Unknown) then
        Critical := Graphs[g].Critical
      else if graph g is observable
        Critical := Yes
        clear GraphsToCheck
      else
        for every decision node that depends on
            Nodes[node].Graph do
          if the node is on an active path then
            add its graph to GraphsToCheck
          if the other exit of the node is unknown or
              the value of the graph is unknown then
            increment UnknownCount for its graph
          else if the other exit differs from the
```

```
                    value of the graph then
                  increment CriticalCount for its graph
                else if the other exit is identical to the
                    value of then graph then
                  increment NonCriticalCount for its graph
        if Critical = Unknown then Critical := No
      if Graphs[ Nodes[node].Graph ].Critical = Yes then
        restore good values to all nodes and graphs
        evaluate the exit not taken during
            the good simulation
        save good values
        if it has a different value then
          {condition II is met}
          mark the node as critical
          if the node's variable = 0 then { *** new *** }
            set TestedForStuck[0] = Yes
          else
            set TestedForStuck[1] = Yes
end;
```

4 Advantages over Gate Representations

4.1 Qualitative Comparison

This method has several advantages over fault simulation based on gate networks: it can run with or without making approximations; it can use fault dropping, and it can handle multiple observable outputs in a single pass.

Fault simulation in gate networks is an algorithmically hard problem because faulty simulation is needed, either explicitly or implicitly, to determine the criticality of stems. If the proportion of stems to gates is constant for a given class of networks, then doubling the size of the network would double the number of stems, which would double the number of faulty simulations required. If the time required per simulation is proportional to the number of gates, the time to do the total simulation would increase by four. An algorithm that increased in time this fast would run too slowly on large networks to be useful for things such as grading test sets. Conventional gate oriented algorithms rely on making approximations or fault dropping in order to handle large networks. For example, the AM&M algorithm [2] may miss detecting a critical stem if it has no critical branches. The BDD algorithm can run in reasonable time without approximations. In this mode, it will never miss a critical node or label a noncritical node as critical. When it is allowed to make approximations, it runs between 3% and 5% faster, depending on the topology of the network.

Some algorithms for fault simulation on gate networks cannot use fault dropping. To reduce the amount of simulation for each test vector, they require that the criticality of all stems with a level higher than L be known before attempting to determine the criticality of a stem at level L. The BDD algorithm exploits criticality information when it is present, but does not require it. Thus, the algorithm inherently compensates for missing information due to fault dropping.

Gate-based fault simulation can only handle one observable output at a time. A normal circuit may have a large number of outputs. Fault-simulation algorithms on gate networks usually trace from each output to the primary inputs, one output at a time. On every BDD network tested so far, tracing all the outputs at once took only slightly longer than tracing a single output. Thus, on classes of circuits that have a fixed proportion of observable outputs to gates, such as some iterative circuits, an algorithm that can trace all the outputs in one pass will run an order of magnitude faster than an algorithm that must trace each output one at a time.

4.2 Experiments

The BDD fault simulation algorithm outlined above, a simplified version of the AM&M critical path tracing algorithm [2] and a gate-network-to-BDD conversion algorithm were coded in VAX-Pascal on a VAX-11/750 under VMS V4.1. Each program was run on two classes of networks and the 74S181 ALU, and run-time statistics were collected that show the relative time complexity of each algorithm. All networks were entered in gate network form and then converted to BDD's.

Modified versions of the published programs AUGMENT and ANALYZE were used to get timings of the programs. AUGMENT reads a Pascal program and inserts calls to timing procedures at the beginning and end of each subroutine. As the program runs, the timing procedures collect timing data. When the program finishes, it writes a file containing the timing information. ANALYZE reads the timing information and a symbol table created by AUGMENT to produce a report showing the number of times each procedure was called, the percentage of the calls to the total number of procedure calls, the average number of milliseconds per call (not counting calls made to other procedures), total number of milliseconds and percentage of the total run time, the average number of milliseconds per call (including calls made to other procedures), total number of milliseconds and the percentage of the total run time. The report can be sorted in forward or reverse order by procedure name or by any of the columns. The last line of the report shows the total run time in seconds and the total number of procedure calls. A report from a single run shows where the program spends its time. It shows which procedures would improve the performance the most it they were optimized. A report from several runs on similar, but increasingly larger, inputs shows where the algorithm encoded by the program should be optimized.

The first network simulated was the Signetics 74S181 4-bit ALU. It had 14 primary inputs, 8 outputs and 87 gates. Exclusive-or gates were replaced with four NAND gates. This circuit was chosen because it is well-known and has been analyzed by other methods.

The exclusive-or trees were generated by a program. For a given number of levels, N, the program generates a tree with 2 ** N inputs, 1 output, and 4 * (2 ** N - 1) NAND gates. This circuit form was chosen because its large amount of reconvergent fanout typically gives worst cases for fault-simulation in gate networks. The corresponding BDD should also be a worst case for BDD fault simulation.

The adders were also generated by a program. For a given number of cells, N, the program would generate an adder with 2N + 1 inputs, N + 1 outputs and N cells. Each cell is a 1-bit full adder implemented in two-level form. This circuit was chosen because it has a large number of outputs and it has several logic levels for each cell due to the carry ripple. Large adders test the ability of a fault simulator to handle a large number of outputs and to propagate faults through many levels of logic.

4.2.1 The XOR Tree

The BDD algorithm may not be as efficient as AM&M [2] for this network. Trees were tested with up to 7 levels and 128 inputs. Although the number of procedure calls in the BDD fault simulator increases with an order of 1.02 (i.e. almost linearly) with respect to the number of nodes as the number of levels is increased from 5 to 7, the total time spent simulating increases with an order of 1.8. The total time spent by AM&M increases with an order of about 1.65.

4.2.2 The Adder

From a network with 5 cells to a network with 50 cells, the total time increases with an order of 1.49 for the BDD algorithm and 1.48 for AM&M. When the BDD algorithm was run with fault dropping and allowed to trace all outputs at once, its total time only increased with an order of 1.40 and its time per input vector decreased by a factor of 4. With the stem approximation enabled (in addition to fault dropping and multiple output tracing), the total time increased with an order of 1.26 with a loss in precision of about 2.6% of the critical nodes not marked as critical.

4.2.3 Results

The Tables below show typical timings for the tested networks. The times are all in real-time seconds spent determining which lines are critical; times for start-up, termination, and printing output are not included. The BDD fault simulator was run with fault dropping and multiple output tracing enabled. Table 1 shows run times comparing the BDD fault simulator to a simple implementation of the AM&M gate-network critical-path tracer.

The gate network path tracer implemented the 'basic algorithm' of [2], but not equal parity cover lines, FFR jumping, 'no overlap' mode or start and stop lines; however, they claim that stem analysis requires 50 percent of their total run time and that using equal parity cover lines reduces stem analysis time by 10 percent and FFR jumping reduces it by another 10 percent. They also claim that 'no overlap' mode reduces run time by about 20 percent. We also suspect that some of the additional time spent by the critical path tracer is due to the implementation, not the algorithm. For example, frontiers were implemented as sets, so operations of the form

i = lowest level gate in Frontier

were implemented as loops. AM&M do not tell how they implement frontiers, but it is conceivable that implementing frontiers as several linked lists of gates, with one list for each level and with each list ordered by gate number would reduce the run time, so that the apparent order-of-magnitude advantage of the BDD algorithm is perhaps exaggerated.

Even though our algorithm seems to increase in time slightly faster than the AM&M critical path tracer, critical path tracing cannot handle multiple outputs or fault dropping. On the other hand, the BDD algorithm can use fault dropping and can trace from multiple outputs with little extra cost.

Table 1. BDD Simulation Time vs. Gate Network Tracing Time

Circuit	BDD Time	Gate Network Time
32 input XOR tree	537	4570
64 input XOR tree	1830	13390
128 input XOR tree	6730	44643
5 cell ADDER	420	5910
50 cell ADDER	14780	180430
74SN181 4-bit ALU	870	28920

Table 2 shows the effects that different modifications have on the algorithm. The results show that graph jumping reduces the number of node evaluations by at least an order of magnitude. Tracing all the outputs at once reduces the number of node evaluations by a factor that depends on the number of outputs and the number of graphs that are used by more than one output. With graph jumping enabled, fault dropping reduced the number of node evaluations by half. Fault dropping had more effect on the adder because the test set was longer and most faults were detected by the first few tests. The approximation algorithm reduced the number of node evaluations from 2 to 5 percent. Since the approximation cannot be used when using fault dropping, it should only be used when complete fault lists are required for each test.

Table 2. Effect on BDD Algorithm of Different Options

Circuit	G	O	D	A	Node Evaluations
32 input, single	N	-	N	-	459 051
output parity	Y	-	N	N	5 136
tree	Y	-	N	Y	5 136
all 436 faults	Y	-	Y	-	3 786
detected					
50 cell adder	N	-	N	-	2 344 689
2232 of 2302 faults	N	-	Y	-	398 703
detected by a set	Y	N	N	N	217 629
of 20 random test	Y	N	N	Y	212 677
vectors	Y	N	Y	-	149 422
	Y	Y	N	N	48 411
	Y	Y	N	Y	46 707
	Y	Y	Y	-	22 775

G = Graph jumping enabled O = All outputs traced at once
D = Fault dropping enabled A = Approximation enabled

4.3 Comparisons with Other Methods

Most timings reported in the literature are times to calculate test sets. Some test-set generation programs create test sets by selecting an untested fault and then attempting to find a test for it. Once a test is found, the programs runs a fault simulator to determine other faults detected by that test. The process is repeated until a suitable number of faults have been tested. Thus, the time to generate a test set of N vectors for a given circuit is at least an upper bound on a reasonable time that a fault simulator could spend on those same N vectors. On a single run for a circuit with 50 logic blocks, Roth et al. [5] report a time of 45 seconds in DALG-II deriving tests and 30 seconds in TEST-DETECT doing fault simulation. Wang [6] claims his algorithm runs about twice as fast as Roth's. Our algorithm can simulate a test set of 20 vectors with a coverage of about 97% on a 50 cell adder with about 600 logic blocks in 31 seconds, so we can do several times the work in about the same amount of time. Cha et al. [7], however, claim to have generated 864 patterns for a 32-bit adder in 422 seconds on an IBM 370-168, or about 2 patterns per second. They do not say if they combine several patterns into each test vector. On the 50 cell adder, we can simulate 2 vectors in about 3 seconds. Accounting for the difference between 50 and 32, we are slightly slower if each pattern is a complete test vector, or several times faster if they combine patterns. In addition, their IBM mainframe is probably faster than our VAX.

More recently, Waicukauski et al. [8] claim to have simulated 50,000 patterns on an 890 gate network in 3.2 seconds on an IBM 3033, or about 72 nsec/gate-pattern. We require about 2.5 msec/gate-pattern, a factor of 30,000 times longer. However, it would be fairly straight forward to modify our algorithm to trace 256 vectors at a time as they do. This would speed up our algorithm by a factor of 100, but if it does not, going to 512 or 1024 vectors should. Also, their IBM 3033 is at least 10 times as fast as our VAX-11/750. This leaves a factor of 30 remaining. The current implementation is entirely in Pascal and contains extra instructions for counting calls to different procedures and for disabling certain features in order to produce information that was useful when modifying the algorithm. Rewriting the critical procedures in assembly language and deleting the extra instructions should make the program run at least twice as fast, leaving us only about 10 times slower. We expect to start work on these modifications in the near future.

5 Differences from Other Works

Although Akers [1] first suggested using BDD's for testing, he used diagrams that matched the gate network at a functional level only. According to Akers, a test set that fully exercises all of the nodes and branches of a diagram would be useful in testing almost any reasonable implementation of its function. Villar and Bracho [3] use a modified BDD that they call an atomic digraph; however, they mainly use the digraph to assist in calculating the ring sum of the good function with a faulty function to derive test vectors. The distinguishing features of our BDD fault simulation algorithm are that 1) the BDD is related to the original network in such a way that faults in the BDD can be mapped back to the network and 2) faulty simulation is not required for each node.

6 Conclusions

A new algorithm was given for fault simulation on binary decision diagrams. The algorithm can trace from several outputs in one pass at little extra cost per output, can use fault dropping, and can make stem approximations when not using fault dropping. For one test vector on a circuit with one output the algorithm has approximately the same computational complexity as fault simulation algorithms based on gate-network tracing. When performing fault dropping and handling multiple observable outputs in a single pass, the algorithm has a considerable advantage over gate-based path tracing.

REFERENCES

[1] S. B. Akers, "Binary Decision Diagrams," IEEE Trans. Comp., vol. C-27, pp. 509-516, June 1978.

[2] M. Abromovici, P. R. Menon, and D. T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation," IEEE Design and Test of Computers, vol. 1, pp. 83-93, Feb. 1984.

[3] E. Villar and S. Bracho, "Fault Simulation and Test Generation in Combinational Circuits Using Atomic Digraphs," Int. J. Electronics, vol. 59, pp. 461-470, 1985.

[4] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," IEEE Trans. Comput., vol. C-24, pp. 242-249, Mar. 1975.

[5] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," IEEE Trans. Elect. Comput., vol. EC-16, pp. 567-580, Oct. 1967.

[6] D. T. Wang, "An Algorithm for the Generation of Test Sets for Combinational Logic Networks," IEEE Trans. Comput., vol. C-24, pp. 742-746, July 1975.

[7] C. W. Cha, W. E. Donath, and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," IEEE Trans. Comput., vol. C-27, pp. 193-200, Mar. 1978.

[8] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, "A Statistical Calculation of Fault Detection Probabilities by Fast Fault Simulation," Proceedings of the 1985 International Test Conference, pp. 779-784.

[9] A. K. Susskind, "Logic Simulation Based on Binary Decision Diagrams," Memorandum dated April, 1986.

VITA

Name:	William Bader
Place of Birth:	Bethlehem, PA
Date of Birth:	September 10, 1964
Parents:	Dr. and Mrs. Morris Bader

Institutions attended:

Fall 1982 to Summer 1986:	Lehigh University

Degrees:

May 1985:	Bachelor of Science in Computer Engineering (highest honors)

Professional Experience:

July 1982 to Present:	Programmer/Analyst at Software Consulting Services. Full time since May 1985.