HIERARCHICAL TEST GENERATION
FOR CMOS CIRCUITS

by

S. Wayne Bollinger

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical Engineering

APPROVED:

[Signatures]

Dr. S. F. Midkiff, Chairman

Dr. J. R. Armstrong
Dr. F. G. Gray
Dr. B. G. Kafura
Dr. J. G. Tront

March, 1992
Blacksburg, Virginia
LD
5655
V856
1992
B.644
e.2
HIERARCHICAL TEST GENERATION
FOR CMOS CIRCUITS
by
S. Wayne Bollinger
Dr. Scott F. Midkiff, Chairman
Electrical Engineering
(ABSTRACT)

As advances in very large scale integration (VLSI) technology lead to higher levels of circuit integration and new design styles and fabrication processes, traditional test generation techniques fail to adequately address the problems of how to (1) accurately represent the structure of design styles and physical faults, and (2) manage the high computational costs and memory resource requirements caused by the complexity of VLSI. This research investigates a modular, hierarchical approach to test generation for combinational complementary metal oxide semiconductor (CMOS) circuits that effectively deals with these issues. Circuits are modeled using multi-level descriptions to handle large circuit sizes while maintaining an effective balance between accuracy and complexity.

Object-oriented analysis and design techniques are used in the development of a hierarchical test generation application implemented using C++. In doing this, the primary objectives were to produce a easily maintainable system, provide an extensible framework for test generation supporting the straightforward incorporation of new types of circuit primitives and faults, and retain the same level of computational efficiency that can be achieved using a procedural language such as C. Characteristics of the object-oriented hierarchical test generation application, such as expandability and run-time efficiency, are compared to those of a standard gate-level test generation program implemented using C and a procedural design approach.

Traditional test generation is extended using the object-oriented test application as a framework to support the testing of circuits containing both structural hierarchy and elements modeled with multiple levels of abstraction. In a structural sense, the inherent hierarchy present in VLSI circuits is exploited by partitioning circuits into modules, and modeling a circuit as a collection of
interconnected module instances. In addition to the structural description, hierarchy is used to represent multiple layers of abstraction. For example, a single module may be represented by a high-level functional description to support high-speed logic simulation and fault propagation, as well as a detailed switch-level description to provide for the use of accurate test generation algorithms. A key feature of this research is the inclusion of a switch-level model that can efficiently represent realistic physical faults occurring in CMOS that the classical gate-level stuck-at fault model cannot.

An extended implementation of the hierarchical test generation application supports test generation for bridging faults in CMOS circuits using quiescent supply current observation (IQDQ testing). This implementation supports the complete set of operations required for performing fully automatic test pattern generation, including fault sensitization, fault simulation, and test compaction. The emphasis of this work is on the efficient generation of IQDQ test sets that achieve very high fault coverage of unrestricted bridging faults, including both gate-level and switch-level bridging faults, with reasonable computational requirements. Fault coverage results are presented for tests of realistic bridging faults derived directly from the CMOS layouts of benchmark circuits.

As a CMOS circuit increases in size and complexity, the computational effort required to analyze and generate tests for the circuit increases dramatically. Limits on the memory capacity and processing power of uniprocessors place severe constraints on the size of circuits that can be considered. This research explores the feasibility of using a circuit partitioning approach to deal with large circuits and reduce the run-time complexity of test generation via parallel processing. Characterization of the major phases of test generation shows how the inherent parallelism existing in test generation can be exploited during forward implication and backward justification. The effectiveness of a circuit partitioning approach is evaluated with respect to its expected performance in a distributed processing environment. Upper bounds on the concurrency available in specific circuits are empirically determined by simulating the behavior of a "perfect" conflict-free test generation algorithm that operates without backtracking. Results presented for a number of benchmark circuits indicate that the average available parallelism is fairly low, limiting the potential speedup of a circuit partitioning approach to test generation.
Dedication

This dissertation is dedicated to my wife and best friend, Andrea, for her unlimited support, encouragement, and love during difficult times.
Acknowledgements

I gratefully acknowledge the support and contributions of my advisor and committee chairman, Dr. Scott F. Midkiff, and would like to express my appreciation for his continual patience and guidance throughout my graduate studies at Virginia Tech. I can think of no better compliment than to say that I highly recommend him to future graduate students who would like to work with a truly helpful, competent, and professional advisor.

I would also like to thank Dr. J. R. Armstrong, Dr. F. G. Gray, Dr. D. G. Kafura, and Dr. J. G. Tront for consenting to serve on my committee and taking the time to review this work, and Dr. D. S. Ha for agreeing with short notice to act as a substitute committee member.

Finally, a special thanks goes to my parents, family, and friends for their encouragement and prayers during the preparation and completion of this dissertation.

This work was supported in part by a Digital Faculty Program/Incentives for Excellence grant from the Digital Equipment Corporation, a SIGDA/DATC Design Automation Graduate Scholarship, and a Presidential Graduate Fellowship from the Graduate School and the Bradley Department of Electrical Engineering at Virginia Polytechnic Institute and State University.
# Table of Contents

Chapter 1. Introduction .................................................. 1

Chapter 2. Background and Motivation ................................. 7
  2.1. Circuit and Fault Representation ............................... 8
    2.1.1. Switch-level Circuit Models ................................ 10
    2.1.2. Fault Models .............................................. 12
    2.1.3. Hierarchical Representation of Circuit Structure ............ 14
    2.1.4. Description Using Multiple Levels of Abstraction ........... 17
  2.2. Test Generation ................................................. 18
    2.2.1. Traditional Testing Techniques ............................ 18
    2.2.2. Switch-level Test Generation ................................ 21
    2.2.3. Hierarchical Test Generation ............................... 27
  2.3. Distributed Techniques ......................................... 30
    2.3.1. Control Techniques ....................................... 31
    2.3.2. Distributed and Parallel Simulation ......................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3.</td>
<td>Parallel and Distributed Test Generation</td>
<td>36</td>
</tr>
<tr>
<td>2.4.</td>
<td>Summary</td>
<td>39</td>
</tr>
<tr>
<td>Chapter 3.</td>
<td>Object-Oriented Test Generation</td>
<td>41</td>
</tr>
<tr>
<td>3.1.</td>
<td>Object-Oriented Analysis</td>
<td>42</td>
</tr>
<tr>
<td>3.1.1.</td>
<td>The Test Generation Problem</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2.</td>
<td>Identifying Fundamental Abstractions</td>
<td>47</td>
</tr>
<tr>
<td>3.2.</td>
<td>Object-Oriented Design</td>
<td>50</td>
</tr>
<tr>
<td>3.2.1.</td>
<td>Specify Classes</td>
<td>52</td>
</tr>
<tr>
<td>3.2.2.</td>
<td>Specify Operations</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3.</td>
<td>Specify Dependencies</td>
<td>57</td>
</tr>
<tr>
<td>3.2.4.</td>
<td>Specify Interfaces</td>
<td>64</td>
</tr>
<tr>
<td>3.3.</td>
<td>Implementation</td>
<td>66</td>
</tr>
<tr>
<td>3.4.</td>
<td>Comparison</td>
<td>71</td>
</tr>
<tr>
<td>3.4.1.</td>
<td>Maintenance</td>
<td>71</td>
</tr>
<tr>
<td>3.4.2.</td>
<td>Extensibility</td>
<td>73</td>
</tr>
<tr>
<td>3.4.3.</td>
<td>Performance</td>
<td>74</td>
</tr>
<tr>
<td>3.5.</td>
<td>Summary</td>
<td>77</td>
</tr>
<tr>
<td>Chapter 4.</td>
<td>I\textsubscript{deg} Test Generation for Bridging Faults</td>
<td>78</td>
</tr>
<tr>
<td>4.1.</td>
<td>Background</td>
<td>79</td>
</tr>
<tr>
<td>4.2.</td>
<td>Circuit Models</td>
<td>82</td>
</tr>
<tr>
<td>4.2.1.</td>
<td>Switch-Level Circuit Representation</td>
<td>82</td>
</tr>
<tr>
<td>4.2.2.</td>
<td>Hierarchical Circuit Representation</td>
<td>84</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.2.3. Forward Propagation</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>5.2.4. Fault Simulation</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>5.3. Determining Maximum Concurrency in Test Generation</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>5.3.1. Conditions for Maximum Parallelism</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>5.3.2. Definition of Concurrency</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>5.4. Performance Models</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>5.4.1. Assumptions and Terminology</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>5.4.2. Case 1: Non-overlapped implication and justification</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>5.4.3. Case 2: Overlapped implication and justification</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>5.4.4. Case 3: Non-overlapped implication and justification with communication overhead</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>5.4.5. Case 4: Overlapped implication and justification with communication overhead</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>5.4.6. Summary of Performance Models</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>5.5. Simulating a &quot;Perfect&quot; Test Generation Algorithm</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>5.6. Results</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>5.7. Summary</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>Chapter 6. Conclusions</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>6.1. Summary of Work</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>6.2. Contributions of this Work</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>6.3. Directions for Future Research</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

Table of Contents
List of Illustrations

Figure 1. (a) Example circuit with stuck-at 0 fault, (b) Valid test that detects fault. . . . . 45

Figure 2. Object diagram for the test generation problem. ........................................... 51

Figure 3. Revised object diagram for the test generation problem. ............................... 59

Figure 4. A tree structured class hierarchy. ................................................................. 61

Figure 5. Module class inheritance tree. ................................................................. 63

Figure 6. Fault class inheritance tree. ................................................................. 65

Figure 7. Alternatives for representing multi-level behavior via inheritance. ................. 70

Figure 8. Tree structure for representing hierarchical circuit state. .............................. 72

Figure 9. Example switch-level module representation. ............................................... 85

Figure 10. Example hierarchical circuit representation. ....................................... 88

Figure 11. Switch-level module class hierarchy. ...................................................... 89

Figure 12. Bridging fault class hierarchy. ................................................................. 94

Figure 13. Control flow for PODEM algorithm. ......................................................... 97

Figure 14. Fault sensitization with unsatisfiable constraints. .................................. 105

Figure 15. Successful sensitization conditions for detection of bridging fault. ............. 107

Figure 16. I_{DDQ} test generation tool suite .............................................................. 110

Figure 17. Simplified control flow for conflict-free PODEM. ..................................... 132

Figure 18. Fan-out free binary tree circuit. ......................................................... 136
List of Tables

Table 1. Performance Comparison of Procedural and Object-Oriented Test Generation Applications ........................................... 76

Table 2. Gate-Level Bridging Fault Classification Results ........................................... 114

Table 3. Switch-Level Bridging Fault Classification Results, Part 1 ........................................... 115

Table 4. Switch-Level Bridging Fault Classification Results, Part 2 ........................................... 116

Table 5. Bridging Fault Coverage Results ........................................... 119

Table 6. $I_{DDQ}$ Test Set Size Results ........................................... 121

Table 7. $I_{DDQ}$ Test Generation Runtime Results ........................................... 122

Table 8. Single Backtrace Test Generation Results ........................................... 150

Table 9. Multiple Backtrace Test Generation Results ........................................... 151
Chapter 1. Introduction

In the design and verification of very large scale integrated (VLSI) circuits, the ability to apply tests that can detect faults, or defects resulting from imperfections in the manufacturing process, is of great practical interest. The growing complexity of VLSI chips is exceeding the ability of engineers to generate high quality tests manually, and consequently the development of fully automatic test pattern generation (ATPG) algorithms able to process large circuits is an ongoing research problem.

Given an integrated-circuit and a list of potential faults, ATPG algorithms attempt to generate a list of input signals, or a test set, that can detect the presence of all faults in the defect list. Classical test generation algorithms operate on circuits described as a netlist of interconnected logic gates such as And, Nor, or Exclusive-Or. Physical defects in the circuit are represented using a single stuck-at fault model where the inputs or outputs of gates can be stuck at a logical value 0 or 1. Although test generation is most commonly performed using gate-level circuit and fault modeling techniques, this approach has not kept pace with advances in VLSI technology. In particular, the traditional approach fails to adequately address the following problems:

- Test generation is an \textit{NP-complete} problem, exhibiting complexity that grows exponentially in circuit size. Thus performing test generation for circuits containing a
large number of elements is extraordinarily compute-intensive, making ATPG infeasible for large VLSI circuits.

- Gate-level circuit and fault models are often inadequate for representing the structure and behavior of realistic faults occurring in important VLSI technologies such as complimentary metal oxide semiconductor (CMOS) circuits.

Recent research in test generation has focused on a variety of methods that provide solutions to these problems. Some approaches move from the gate-level to higher levels of abstraction, attempting to speed up test generation by modeling circuits with a smaller number of high level primitives that can be processed efficiently. Other approaches have examined the use of lower level circuit and fault models to more accurately represent realistic faults in VLSI circuits. Of particular interest are approaches that utilize hierarchical techniques to address both the complexity and accuracy problems simultaneously. Since hierarchy has been used quite successfully in other computer-aided design (CAD) tools for VLSI circuits, there is a clear precedence for its incorporation into test generation.

A number of hierarchical approaches have been proposed for test generation [58, 71, 68]. Results reported for those that have been implemented indicate that the use of hierarchical techniques in test generation can provide significant speedup over a flat gate-level approach, especially for large circuits. Unfortunately, efficient hierarchical algorithms for manipulating multi-level circuit descriptions are extremely complicated, requiring complex data structures that can support dynamic changes in internal circuit representation in addition to the basic operations needed for test generation, such as fault simulation. As a result, traditional "flat" test generation
implementations typically require extensive modifications or a complete redesign to incorporate hierarchy.

Although reimplementing a test generation tool from the ground up is an expensive proposition, it provides an ideal opportunity to employ new tools and techniques that support the development of a superior application. The work presented here complements and extends prior research in test generation by examining the use of an object-oriented approach for the design and implementation of a hierarchical test generation algorithm that supports the testing of circuits containing elements modeled using both structural hierarchy and multiple levels of abstraction. The implementation was developed using C++, with the primary objectives being to (1) produce an easily maintainable system, (2) provide an extensible framework for test generation supporting the straightforward incorporation of new types of circuit primitives and faults, and (3) retain the same level of computational efficiency that can be achieved using a procedural language such as C.

Bridging faults, or shorts between normally unconnected signal lines, are a common failure in CMOS circuits, and as such were chosen as a target fault for this research. Studies examining realistic faults in digital CMOS circuits suggest that bridging faults account for from thirty to fifty percent of all faults [35]. Investigation into test generation for such faults has been limited, however, due to problems encountered when attempting to detect bridging faults using testing based on logic value observation.

This research considers the accurate representation of bridging faults (BF's) in CMOS, and the use of I_{DQ} testing, or supply current monitoring, an alternative technique for the detection of such

Chapter 1. Introduction
faults made possible by the normally small supply currents in CMOS circuits [2]. To allow for
the direct representation of CMOS BF's, a switch-level circuit model is employed in this work.
By modeling circuits at the transistor- or switch-level, the physical fault mechanisms present in
CMOS circuits can be represented by a general fault model incorporating both shorts and opens
in addition to stuck-at faults. The necessary topological information is described using a graph-
based circuit-model that corresponds directly to the circuit topology.

The use of a switch-level circuit representation significantly increases the amount of circuit
information that must be represented. Modeling a large VLSI circuit with a flat, switch-level
description would require enormous memory capacity. Even if such a circuit representation was
feasible, the time needed to process the circuit would increase exponentially with circuit size. In
this research the computational overhead imposed by the use of switch-level techniques is
counteracted by extending the object-oriented implementation of the modular, hierarchical test
generation algorithm.

The bridging fault test generation algorithm presented here addresses the primary issues involved
in test generation, including fault sensitization, backwards implication of logic values to
controllable inputs, and propagation of faults to observable outputs. The emphasis of this work
is solely on the generation of tests to achieve very high fault coverage of unrestricted BF's using
current observation.

Analysis of classical gate-level test generation has shown that the task of generating a test vector
to detect a specific stuck-at fault is \textit{NP-complete}, even for combinational circuits [47]. Thus the
computational requirements for test generation grow exponentially as circuit size increases. Even
in the case of a "perfect" test generator that determines tests optimally and operates without backtracking, the complexity of test generation for a combinational circuit containing \( N \) gates grows with at least order \( N^2 \) [42]. Consequently, performing fully automatic test pattern generation is infeasible for many circuits that are being produced using current technology.

Due to the importance of test generation, much effort has been expended in finding methods to speed up test generation. Heuristics are used to search the problem space in almost all test generation algorithms [43, 39, 87]. However, even algorithms that use highly optimized heuristics, such as FAN [39], can reduce the run-time only to a limited extent. Hierarchical and multi-level techniques have also been used with success, but the possible performance improvement in uniprocessor implementations of test generation algorithms is clearly limited. Given the rapid growth in the size of VLSI circuit designs, it has become apparent that the use of conventional uniprocessor computers for test generation will be inadequate.

Of the alternatives that exist for exploiting parallelism in test generation, reported work on one in particular, circuit partitioning, has been quite limited. Circuit partitioning involves dividing a circuit description and distributing it among a number of processors, which then cooperate to generate tests. A primary advantage of this approach is its ability to perform test generation without requiring every processor to maintain a complete representation of the circuit under test, a significant limitation of other approaches. This supports the handling of very large circuit sizes. However, this advantage is offset both by inherently high interprocessor communication costs and the substantial difficulty of modifying test generation algorithms that take advantage of global information so that they execute efficiently in a distributed environment. The limiting factor that determines whether or not a circuit partitioning approach to test generation will be beneficial is
the amount of concurrency that can be taken advantage of during test generation. This depends on both the amount of concurrent activity exhibited during test generation by the circuit under test, and the ability of the test generation algorithm to execute operations in parallel and utilize this concurrency.

This research explores the feasibility of using a circuit partitioning approach to deal with large circuits and reduce the run-time complexity of test generation via parallel processing. Characterization of the major phases of test generation shows how the inherent parallelism existing in test generation can be exploited during forward implication and backward justification. The effectiveness of a circuit partitioning approach is evaluated with respect to its expected performance in a distributed processing environment. Upper bounds on the concurrency available in specific circuits are empirically determined by simulating the behavior of a "perfect" conflict-free test generation algorithm that operates without backtracking. Results presented for a number of benchmark circuits indicate that the average available parallelism is fairly low, limiting the potential speedup of a circuit partitioning approach to test generation.

The following chapter discusses switch-level modeling, test generation, and distributed processing techniques, and surveys prior research. Chapter 3 describes the object-oriented test generation framework used in this research. Chapter 4 presents a full ATPG implementation for generating $I_{DDQ}$ tests for bridging faults in CMOS. The feasibility of a circuit partitioning approach to test generation is studied in Chapter 5, and conclusions and suggestions for further research are provided in Chapter 6.
Chapter 2. Background and Motivation

In recent years the switch-level structural representation for CMOS circuits has attracted considerable attention for both simulation and test generation [34, 7, 9, 17]. The switch-level approach to test generation shows promise as a means of partially solving the problem of accurate VLSI structure representation. By modeling circuits at the transistor- or switch-level, the physical fault mechanisms present in CMOS circuits can be represented by a general fault model incorporating both shorts and opens in addition to stuck-at faults. The necessary topological information is described using a graph-based circuit model that corresponds directly to the circuit topology, allowing for a realistic fault set. This approach supports the generation of test vectors that achieve a very high coverage for physically realistic faults.

The use of a switch-level circuit representation significantly increases the amount of structural circuit information that must be represented during test generation. Previous approaches to switch-level test generation often fail to address this issue, limiting their application to small circuit sizes. The use of a circuit model incorporating a hierarchy of modules or sub-graphs offers a solution to a number of problems that have limited the effectiveness of previous switch-level approaches. Studies of hierarchical approaches to test vector generation at the gate-level are promising, and show that the hierarchical approach proves to be faster than a flat-level approach for large circuits.
With the rapid growth in the size of VLSI circuits, it has become apparent that the use of conventional uniprocessor computers for performing test generation on large circuit designs will be inadequate. Parallel computing techniques have been successfully applied to logic and fault simulation, and there is also evidence that parallel processing can effectively reduce the execution time of test generation.

This chapter presents a survey of the literature related to this research and the above issues, including switch-level circuit and fault modeling, traditional and multi-level test generation, and approaches to speeding up test generation via parallel processing. Discussion of factors that motivated the development of the algorithms investigated in this research are presented here as well.

2.1. Circuit and Fault Representation

The circuit and fault models used in classical test generation algorithms evolved from the board-level design and testing of transistor-transistor logic (TTL) components [41]. These models represent circuits as connections of standard logic gates and limit faults to the line stuck-at fault model, assuming that connections between gates can be stuck at either a logic one or logic zero value. The stuck-at fault coverage measure is a widely accepted measure of test vector fault coverage. Test generation at the gate-level is a well understood problem, and a number of algorithmic methods for automated test generation have been studied. However, traditional models are inadequate for test generation for contemporary CMOS circuits for several reasons:

1. gates are unidirectional, MOS transistors are bidirectional,
2. the structure of CMOS logic designs and basic CMOS elements such as complex gates and transmission gates cannot be represented using logical gates,

3. the physical failure mechanisms in CMOS circuits are topological and cannot be accurately represented by a stuck-at gate-level model, and

4. with CMOS it is necessary to model intra-gate faults, faults within a single logical circuit block, as well as inter-gate faults occurring between gates.

In CMOS circuits, faults may occur at transistors and at electrical nodes that are either internal or external to a gate. In addition to nodes stuck-at zero or stuck-at one, common faults in CMOS circuits include electrical shorts between two non-equivalent nodes and opens between two nodes normally connected by a wire or transistor.

Behavioral-level models, another common means of representing VLSI circuits, have an advantage over lower levels of description in that they can model circuits using less structural detail and complexity. Behavioral-level models are useful when detailed knowledge of the gate structure of a circuit is unavailable. In addition, the reduced level of complexity can allow for efficient implementations of simulation and test generation tasks. As with the gate-level model, however, behavioral models alone are not suitable for modeling the structure and fault mechanisms of CMOS. Indeed, the mismatch between model and structure is even more serious when behavioral-level models are used to model CMOS circuits. The use of gate-level and behavioral-level models is inadequate for achieving very high fault coverage for large-scale CMOS circuits. Instead, a structural model must be used. Such a model, the switch-level model, is introduced in the following section.
2.1.1. Switch-level Circuit Models

Accurate test generation for CMOS VLSI circuits requires the use of techniques that are different from those commonly employed in gate-level test generation since the effects of physical failure mechanisms and certain structures commonly found in CMOS circuits do not possess a corresponding Boolean gate representation. Pass transistor logic is an example of a design style that cannot be directly modeled as a set of connected logic gates. As a result, CMOS circuits are more appropriately modeled as a set of interconnected transistors. Such a model is commonly called a switch-level representation, since transistors are basically treated as switches that are either on or off, depending on the state of their controlling gates.

Switch-level modeling ignores the lowest-level electrical aspects of circuit behavior and deals with logical behavior only. In effect, the switch-level abstraction attempts to capture the digital behavior of the transistor switch. For the purposes of test generation, a transistor can be viewed as a three terminal device, with terminals labeled source, gate, and drain. For an n-channel transistor, a logical level 1 or high voltage level applied at the gate induces a high conductance path between the source and drain, and the transistor is said to be on. A logical level 0 or low voltage at the gate isolates the source from the drain, and the transistor is said to be off. A p-channel transistor is turned on and off by applying the opposite logic values to its gate terminal. Since the switch-level model captures the digital behavior of a circuit, any voltage drop across a transistor that is turned on is ignored. Thus when the gate of an n-channel device is at logic 1, the source and drain assume a common logic value. In general, the exact logical values for each node in a switch-level circuit are determined by both the logical levels of connected nodes and the relative strengths of logic values.
In addition to its ability to match CMOS design styles by using transistors rather than gates as the primitive element, the switch-level model has a number of other advantages [46]. Since the interconnection structure of the switch-level model matches the structure of actual circuits, it can directly represent common physical failures. The use of switch models allows for greater accuracy at the cost of an increase in computational complexity compared to gate-level representations. The tradeoff between accuracy and complexity can be varied by modifying model parameters such as the number of signal types and the strengths of switches and nodes. And finally, concepts developed for both gate-level and circuit-level modeling can be incorporated into switch-level design and analysis tools.

A variety of techniques have been presented in the literature for representing a circuit at the switch-level. Matrix methods, tree structures, and graph-based techniques have been employed with varying degrees of success. These techniques are examined in detail in Section 2.2.2, covering switch-level test generation.

An early switch-level model for MOS circuits developed by Bryant [16] is mentioned here because of its widespread adoption by other researchers and its use as a basis for the models employed in this research. A circuit model is represented as a labeled, undirected graph $G = (V, E, B)$, with elements defined as follows.

- $V$: The set of graph vertices, with each non-separable electrical node represented by a unique vertex.
- $E$: The set of graph edges. A unique edge is associated with each wire connecting two separable, but electrically equivalent, nodes and with each transistor.
The set of control variables. For every edge in $E$, there is an associated control variable in $B$ that determines whether or not the edge is conducting. N-channel transistors are controlled directly by the gate signal, and p-channel transistors are assigned the inverse of their gate signals. Edges representing wires are assigned a "1" since they always conduct.

The circuit model used in this research is a subset of Bryant’s switch-level model. Transistors and nodes are modeled with a uniform strength value, and the control values for edges are explicitly enumerated. These simplifications serve to make the model more suitable for test generation purposes by eliminating the need to store information not necessary for modeling logical circuit behavior.

2.1.2. Fault Models

It has been shown in the literature that classical test generation techniques based on the assumption that all failures may be modeled by stuck-at-0 or stuck-at-1 logical faults associated with the logic diagram of a circuit are not suitable for test generation for large-scale CMOS circuits [41, 72, 8, 62]. This is due to the inability of gate-level models to account for the effect of physical failures such as shorts and opens. The following fault types have been proposed to represent the physical failure mechanisms occurring in CMOS circuits. For each fault, the switch-level fault representation is given.

Node stuck-at-0 fault: Similar to the gate-level stuck-at model, a stuck-at 0 fault occurs when a node’s logic value is permanently fixed at a logic 0. Let $e$ be a short edge with associated control variable $b$ between $V_{SS}$ and the stuck node. A stuck-at-0 fault is modeled by fixing $b=0$ in the fault-free circuit and $b=1$ in the faulty circuit.
Node stuck-at-1 fault: A stuck-at 1 fault occurs when a node’s logic value is permanently fixed at a logic 1. Let \( e \) be a short edge with associated control variable \( b \) between \( V_{nn} \) and the stuck node. A stuck-at-1 fault is modeled by fixing \( b=0 \) in the fault-free circuit and \( b=1 \) in the faulty circuit.

Transistor stuck-off: The stuck-off fault model was originally developed by Wadsack [96] to model the behavior that occurs when a transistor becomes permanently non-conducting. This fault is modeled by fixing the control variable \( b \) of the faulty transistor as \( b=0 \) in the faulty circuit and letting \( b \) assume its usual value in the fault-free circuit.

Transistor stuck-on: A stuck-on fault results when a transistor becomes permanently conducting. This fault is modeled by fixing the control variable \( b \) of the faulty transistor as \( b=1 \) in the faulty circuit and letting \( b \) assume its usual value in the fault-free circuit.

General open fault: An open fault prevents desired conduction between two circuit nodes. Let edge \( e \) with control variable \( b \) be the circuit element that contains the open. The fault is modeled by fixing \( b=1 \) in the fault-free circuit and \( b=0 \) in the faulty circuit.

General bridging fault: A bridging or short fault causes unwanted conduction between two nodes in a circuit when conduction should normally not occur. Bridging faults are modeled by adding a short edge \( e \) with control variable \( b \) to the circuit if a stuck-on transistor is not present across the fault site. The fault is modeled by fixing \( b=0 \) in the fault-free circuit and \( b=1 \) in the faulty circuit. Bridging faults may be classified as intra-gate, i.e. within a single gate, or inter-gate, i.e. between gates.

Note that line stuck-at faults and transistor stuck-on faults are a subset of the general bridging fault. A line stuck-at fault is equivalent to a short between a node and either the power or ground node, and a transistor stuck-on is equivalent to a short between the transistor’s source and drain.
Similarly, a transistor stuck-off fault is equivalent to an open between the transistor's source and drain.

In addition to fault equivalence, the concept of fault collapsing can be used to identify cases where a single test vector is a test for more than one fault. For example, in a series chain of transistors, a test that detects an open fault anywhere in the chain is also a test for every other series open fault in the chain. Likewise, a test for a short fault will also detect the presence of any parallel short faults.

The fault model for the proposed research will encompass all of the above fault types, and the test algorithm will coordinate tests for the bridging fault types. A variety of fault sensitization and propagation techniques are supported by the overall test generation approach, extending the existing state of switch-level test generation.

2.1.3. Hierarchical Representation of Circuit Structure

Unfortunately, for large CMOS circuits the complexity and storage requirements of a graph-based switch-level description pose a formidable problem. Representing a large circuit with a single-level monolithic netlist requires enormous memory capacity for storage, and exponential run time for circuit processing. Previous approaches to switch-level test generation using a strictly graph-based technique fail to adequately address the problem of handling the complexity of large CMOS circuits. Some approaches rely on the generation of a single graph to describe an entire circuit, leading to extremely complex and unwieldy graph representations [26, 27, 3, 25, 79]. To maintain reasonable memory requirements and computational complexity for processing VLSI circuits, this research will incorporate the use of a hierarchical circuit model to represent circuits.
A number of researchers have employed hierarchical methods to represent CMOS VLSI circuits, primarily for logic and fault simulation purposes. This section describes some of the models used in hierarchical simulation; hierarchical test generation models are covered in a later section.

Rogers and Abraham [83] use hierarchy in a logic and fault simulation system to provide a framework for reducing the complexity of large circuit designs to manageable levels. Their hierarchy is built up with repeated abstractions of circuit primitives, with primitives forming the lowest level atomic units of a circuit representation. Above the primitive level are macromodules, composed of invocations of primitives and/or other macromodules. This approach was used to implement CHIEFS, a concurrent hierarchical fault simulator. More recently, Rogers and Abraham have shown that concurrent hierarchical fault simulation behaves as $O(n \log n)$ for circuits described in a hierarchical fashion with $n$ primitives, as opposed to approximately $n^2$ time for flat, or non-hierarchical, concurrent fault simulation [84]. In related work, Guzolek describes WRAP, a method for hierarchical compression of fault simulation primitives, that reduces $n$ by compressing macromodules into circuit primitives [44]. By converting a large number of simple primitives into fewer, more complex primitives, the use of hierarchy compression resulted in a factor of three speedup for some circuits. Both Guzolek's work and Rogers's work, however, are limited to circuits described at the gate-level.

Motohara et al. [70] present an approach to hierarchical fault simulation for CMOS circuits that uses a combination of switch-level and behavioral models. The circuit under consideration is represented as an interconnection of macromodules, with each macromodule containing a behavioral and/or structural description. At the lowest level of the structural description hierarchy, all elements included in macromodules are logic primitives. The behavior of macromodules can
be described using a truth table, state transition table, or C-language description. Even with the additional overhead due to the use switch-level descriptions at the lowest-level, the approach was successful in reducing both the time and space requirements of fault simulation. However, the authors do not appear to provide a means for modeling the bidirectional nature of MOS transistors. This implies that certain classes of CMOS circuits cannot be modeled with their approach.

Another approach for hierarchical simulation of complex switch-level circuits is described by Saab [86]. In this work, a hierarchical switch-level simulator operates directly on a hierarchical circuit description and handles general MOS circuits. The issue of bidirectional signal flow between circuit blocks is addressed, and mixed mode modeling techniques are allowed. At any level of the hierarchy, the circuit can be described as a mixture of transistor networks, gates, and high-level functions. The hierarchy in this work is used both to reduce the memory requirements of the simulation, and to speed up the simulation algorithm through the use of high-level models. Results show that the use of hierarchy allows for the simulation of circuits that are too large to simulate at the flat-level. Experiments with the use of high-level descriptions showed promising speedups compared to a non-hierarchical approach, ranging from five for an adder circuit, up to 36 for a PLA.

A fault simulation method that considers current supply, or I_dq, testing of bridging faults in CMOS circuits is presented by Lim [60]. CMOS circuits are partitioned into connected modules based on the circuit layout, and are represented by a two-level hierarchy. At the lower level the structural detail of modules is represented with a switch-level description, and the higher level describes the global interconnection of module inputs and outputs. Some important CMOS design styles such as transmission gates are not considered. The method determines fault sensitization

Chapter 2. Background and Motivation 16
conditions by using logic simulation with a good circuit, and does not consider fault injection or fault propagation. Results from fault simulation experiments using gate-level, switch-level, and random test sets are analyzed to compare the effectiveness of the three techniques. Although results show that the improvement in bridging fault coverage gained by using switch-level test sets is on the order of three percent, only small, single-module circuits are considered. The fault simulation of multi-module circuits composed of more complex modules is not considered.

The results obtained by applying hierarchical techniques in the area of simulation show promise for similar improvements in a test generation approach that exploits hierarchy.

### 2.1.4. Description Using Multiple Levels of Abstraction

Integrated circuits and the blocks making up integrated circuits can be represented at four distinct levels of abstraction.

1. **Circuit-level**: The lowest level of abstraction, in which a circuit is modeled as a set of detailed interconnected circuit elements, such as transistors, resistors, and capacitors. Circuit behavior is expressed in the domain of analog currents and voltages.

2. **Switch-level**: A circuit is represented as a set of connected bidirectional switches that are in one of two states, open or closed.

3. **Gate-level**: A circuit is modeled as a set of connected unidirectional logic gates that operate on Boolean logic values. Typical gate primitives are And, Or, and Invert.

4. **Functional- or behavioral-level**: The highest level of abstraction, in which the output behavior of a circuit is modeled completely by the inputs and a description of the behavior of the circuit outputs as a function of the inputs.
Representation of a circuit with multiple levels of abstraction is equivalent to employing a second form of hierarchy. Although the inclusion of multiple levels of abstraction that describe the same circuit involves additional memory requirements, the higher levels can be used to achieve a speedup in algorithms that do not require the detailed information of lower levels of abstraction. Two of the approaches to simulation in the previous section used both structural hierarchy and multi-level techniques to manage the memory and computational requirements of processing large circuit descriptions [70, 86].

2.2. Test Generation

As stated in the previous section, traditional test generation techniques evolved from board-level designs using TTL components. Even though these stuck-at fault test generation algorithms are inadequate for complete testing of CMOS faults, the basic test generation strategies consist of a sequence of phases that are independent of circuit technology. These phases or elements are commonly labeled as fault sensitization, backwards line justification, forward propagation or implication, and guidance measures. In this section, classical test generation approaches that have been applied to switch-level test generation are described. A survey of techniques for switch-level test generation is presented, and previous work in hierarchical test generation is discussed.

2.2.1. Traditional Testing Techniques

Of the classical gate-level test generation techniques, such as the D-Algorithm, PODEM, FAN, and others, only the D-Algorithm and PODEM have been applied to switch-level test generation. This section briefly describes these two algorithms.
The D-Algorithm, developed by Roth, was the first widely accepted algorithm for test pattern generation [85]. The D-Algorithm is potentially capable of finding tests for all detectable faults in a gate-level combinational circuit, given sufficient memory resources and unlimited time. Based on the "calculus of the D-cube," Roth developed a complete mathematical formulation that guarantees that a test can be generated for any detectable stuck-at fault. The logic signal values used in the D-Algorithm include logic 0, 1, and X (indeterminate), and the fault signal values $D$ and $\overline{D}$. In the usual case, $D$ is defined as a logic 1 for the fault-free circuit and a logic 0 for the faulty circuit, and $\overline{D}$ is the complement of $D$. These signals are used for the forward fault propagation and backward fault sensitization of good and faulty signal values in the circuit. To sensitize and propagate circuit faults, conventional Boolean operations and certain specific fault signal interaction rules are applied to the fault signal values $D$ and $\overline{D}$, and a systematic search of the entire logic space is conducted.

The computational weakness of the D-algorithm is a result of not utilizing global network information to guide the search strategy. Since the search space for fault sensitization and propagation displays exponential growth with increasing circuit size, the search complexity is enormous. Conflicts in the search process occur when the competing requirements of the sensitization and propagation phases require complementary values for the same node. Without the ability to eliminate invalid sections of the search space, the D-Algorithm may take a long time to search the solution space only to find that no valid solution exists.

The PODEM algorithm improves on the efficiency of the D-Algorithm by taking advantage of the fact that all nodes in a combinational circuit are completely determined by the primary input
values [43]. After determining a stuck-at fault to be tested, PODEM assigns input values one at a time and immediately simulates the results. This strategy guarantees that any conflict can be resolved by complementing the latest input assignment setting, or that no solution exists for the current set of input assignments. When a conflict occurs for both logic states 0 and 1 on the first primary input pin selected for assignment, the fault is guaranteed to be undetectable.

The PODEM algorithm begins by initializing all primary inputs to unknown, or X. As logic values are assigned to inputs, the logic value of any given node in the circuit will either be X or be determined by a previous input logic assignment. The search then continues as long as no node value conflicts arise due to fault sensitization and propagation requirements. As the input logic assignment and simulation process proceeds, any conflicts that arise are a function of the latest input logic assignment and all previous input logic assignments, since all X-valued nodes are dominated by the most recent logic selections. The combinations of input logic values for unassigned inputs do not need to be searched since they cannot resolve the current conflict. Therefore a significant portion of the search space can be discarded, saving the computational effort that would have been required to explicitly explore that portion of the search space.

To test any given fault, PODEM will perform a pseudo-exhaustive combinatorial input search, but is able to immediately discard search paths when a conflict arises. In addition, gate-level PODEM algorithms use a testability analysis to help choose the order in which input pins are selected. The probability of finding redundant or undetectable faults early in the search process is improved by setting the value of the most difficult input nodes first, so that useless search combinations are not explored. The expected length of a test assignment for PODEM is always less than that of the D-Algorithm, since PODEM will on the average avoid larger portions of the search space.
2.2.2. Switch-level Test Generation

This section presents an overview of the literature on switch-level test generation for CMOS circuits. This discussion is primarily limited to approaches that utilize switch-level models to represent circuits, rather than adaptations of gate-level models. All of the methods operate on combinational circuits described at the switch-level.

In an early paper, Galiay et al. analyzed the failure mechanisms occurring in MOS circuits and discussed test generation problems at the switch-level [41]. By direct inspection of 4-bit microprocessor chips, they showed that the great majority of physical failures in MOS circuits are shorts and opens. The gate-level stuck-at fault model is shown to be inadequate for modeling these faults. Specific test generation procedures are not discussed, rather a recommendation is made to simplify the test generation problem by using layout rules that improve circuit testability by ensuring that all circuit failures are covered by the stuck-at fault model. The proposed rules have the disadvantage of imposing layout constraints, increasing chip area, and adversely affecting circuit performance.

Chiang and Vranesic consider the problem of detecting faults in CMOS combinational networks [26, 27]. They analyze the effects of open and short faults in CMOS networks, and approach the problem of detecting stuck-open faults by suitably organizing the test sequence. CMOS circuits are modeled using connection graphs, and faults are modeled as opens or shorts on branches in these graphs. Graph-based test generation techniques based on these models are reported. Chiang and Vranesic noted that for complementary pairs of transistors in a CMOS gate, tests that can detect an open fault in one transistor can also detect a short fault in its complementary counterpart. However, this observation ignores the fact that short faults may cause indeterminate values on the
outputs of CMOS gates and thus be undetectable. In addition, the test algorithms require the
generation of a single monolithic connection graph to represent circuits, making the method
unsuitable for large circuits.

Taking advantage of the low static current characteristic of CMOS, Acken shows that power
supply current measurement techniques can be used to detect shorts or bridging faults in CMOS
circuits [2]. A criteria for evaluating the effectiveness of bridging fault test vectors is presented.
Acken shows that to detect any shorts present in a circuit using power supply measurements, it
is sufficient to ensure that the shorted nodes are driven to opposite logic values during a test
sequence. The resulting current path between $V_{DD}$ and $V_{SS}$ causes the circuit to draw an excessive
amount of current. Although this work describes how to evaluate bridging fault test sets, it does
not address the issue of how to effectively generate such tests.

To determine if current measurement techniques could be used to detect stuck-open faults, Soden
et al. evaluated the electrical effects resulting from CMOS stuck-open faults, including voltage
levels, quiescent power supply current ($I_{DDQ}$), and transient response [89]. They demonstrate that
it is possible for test patterns that achieve complete logic state testing to not detect stuck-open
faults. Effects caused by the high impedance state associated with stuck-open faults were
investigated by measuring the transient responses of defective node voltage and power supply
current. Results indicated that $I_{DDQ}$ measurements can significantly enhance the detection of stuck-
open defects over the use of functional testing alone, but do not guarantee their detection.

A test generation algorithm for detecting stuck-at and transistor stuck-open faults in combinational
CMOS circuits is described by Agrawal [3]. Each CMOS gate in a circuit is modeled as a graph

Chapter 2. Background and Motivation
derived from its switch-level description. The test generation algorithm makes use of path finding algorithms within graphs, and assumes the availability of an algorithm for finding all paths between two given nodes. Other types of faults are not addressed, and no implementation details or performance results are given.

Jain and Agrawal present a procedure to convert a CMOS circuit containing both stuck-at and stuck-open faults into an equivalent circuit containing only stuck-at faults [48]. The method depends on a modeling block that represents the "memory" state caused by stuck-open transistors. After conversion, test generation is accomplished using a modified D-algorithm. The approach has the two primary drawbacks: (1) it usually generates equivalent circuits that are much larger than the original ones, and (2) it uses a complex implementation of the D-algorithm to handle memory elements. Important faults such as bridging faults are not considered in this work. Also, this approach restricts the class of circuits for which it can work. In similar work, Al-Arian and Agrawal introduce a CMOS fault model in which CMOS stuck-open faults are transformed into classical TTL gate-level stuck-at faults [5]. The test generation method is based upon the path sensitization method and the D-algorithm. As before, only stuck-at and stuck-open faults are considered. Results are presented only for very small circuits.

Reddy et al. use a switch-level model to represent MOS circuits and present a test generation algorithm that detects single transistor stuck-on and single transistor stuck-open faults [81]. The test generation algorithm is based on the classical D-algorithm, using transistor groups in much the same way as Boolean logic gates are used in the D-algorithm. A procedure is proposed that generates robust two-pattern tests for stuck-open faults that remain valid in the presence of
arbitrary circuit delays. The problem of invalidation of stuck-on tests due to indeterminate output values is not considered, and faults other than transistor stuck-on and stuck-off are not considered.

Closely paralleling Reddy’s work, Chen et al. point out the deficiency of using ad hoc extensions to gate-level models to model CMOS circuits [25]. They develop an algorithm for generating tests at the switch-level by extending Bryant’s MOSSIM-II algebra to handle sets of values, so that implication can be done during test generation. In addition, they generalize the search method used in the PODEM algorithm to search over a space-time representation for circuits. The algorithm is capable of generating multiple-pattern tests that detect switch stuck-open and stuck-on faults, and that remain valid under arbitrary circuit delays. A programmed implementation of the algorithm is used to demonstrate the feasibility of the approach. For a 238 transistor circuit, the average time to generate one test is around 100 CPU seconds on a 1 MIPS machine. This rules out the use of the basic algorithm on large circuits. Because the algorithm only considers faults that are deterministically testable in the logic value domain, the majority of transistor stuck-on faults in the benchmark circuits were not considered.

A method to generate test sets to detect stuck-open faults in very large CMOS circuits is presented by Rajski and Cox [79]. The test method is based on the application of sequences of three adjacent input vectors called trios. Stuck-open faults detected by such sequences are determined by tracing dynamic paths from primary inputs to primary outputs. This method requires no network transformation prior to processing, and generates test sets to cover stuck-open faults directly, rather than sorting a conventional stuck-at test set to make it cover such faults. The memory requirements for the method are minimal, requiring only a few variables per circuit line. Faults other than stuck-at 0, stuck-at 1, and transistor stuck-open are not considered.
Jha addressed the problem of detecting multiple stuck-open and stuck-on transistor faults in CMOS, showing that a test set that detects single stuck-open faults in a CMOS complex gate also detects most of the multiple stuck-open faults in that gate [52]. He also identifies conditions that produce 100 percent fault coverage, and that guarantee that multiple faults can be detected using only logic level tests. Results show that for any complex gate, a large percentage of multiple transistor faults are detectable. In a later work, Jha showed that a properly derived test set based only on two-pattern tests, designed to detect single stuck-open faults, can also detect all multiple stuck-open faults inside any CMOS gate in a circuit [53]. The concept of two-pattern tests is extended to three-pattern tests that under certain conditions can detect any multiple stuck-open fault in the circuit. Since the multiple stuck-open fault coverage capability in this work is obtained using test sets designed to detect single stuck-open faults only, the higher fault coverage is obtainable without the price of high test generation overhead.

Robinson and Shen describe a switch-level test generation algorithm for CMOS circuits that employs a combination of algebraic and circuit techniques to generate tests directly from the transistor level representation of a circuit [82]. A CMOS circuit is partitioned into a network of subcircuits. Each subcircuit is modeled as an arbitrarily connected relay-switch network with a set of control inputs and primary terminals. Every pair of primary terminals has an associated Boolean transmission function. To avoid run-time path tracing using graph models, the test generation procedure consults a library of subcircuits containing precomputed information. The interconnection of subcircuits provides topological constraints in the search. The search technique possesses backtracking capabilities similar to those found in the D-algorithm. The work considers test generation only for line stuck-at faults. Results indicate that the approach could be applied to other fault types to provide for expanded test generation capabilities.
Shih and Abraham propose a new methodology for test generation at the transistor level, for realistic failures such as bridging faults and transistor gate-to-source short and gate-to-drain short faults in CMOS combinational circuits [88]. A tree structure representation is used for representing fault-free CMOS complex gates and for propagating errors due to faults. The technique adapts the tree model for CMOS gates for use with a modified D-algorithm. Test generation procedures are described to generate tests for bridging faults, transistor stuck-open and stuck-on faults, and transistor gate-to-source and gate-to-drain shorts. Bridging faults are limited so that no feedback paths are created in faulty circuits. Since primitive D cubes can be generated only by using assumptions about resistances, potential logic level tests must be validated using a circuit level simulator.

In some of the most recent work reported on switch-level test generation, Damper and Burgess describe a switch-level method for generating the singular cover of an MOS primitive gate using path algebras [29]. As in the D-Algorithim, the singular cover is essentially a compact version of the gate's truth table. A primitive gate is defined as a gate in which all transistor gates are controlled by primary inputs. The method generates tests to cover all specified open and short circuit faults and all irredundant transistor stuck-open and stuck-on faults, should such tests exist. Two path algebras are used, one to generate tests for open circuit faults and a second for short circuit faults. An adjacency matrix approach is used to describe gates, supporting the representation of most realistic MOS gates. Effects such as bidirectionality and tristate behavior are taken into account. Test generation for networks of primitive gates is accomplished in two stages. In the first stage, singular covers are derived for primitive gates, and in the second, a variation of the D-algorithm with a modified D-drive procedure is used. This approach appears to be one of the most unified and general of those reported in the literature. No implementation
results are presented, but practical limits on the size of circuits that can be handled appear to be comparable with those encountered with use of the classical D-algorithm. In common with previous approaches, this work assumes that all short faults are deterministically testable using only logic level tests.

2.2.3. Hierarchical Test Generation

As with the case of hierarchical circuit representation, a number of researchers have employed hierarchical methods to generate tests for CMOS VLSI circuits. However, none of these methods model circuit structure below the gate-level. To date, no results have been published on a method combining a hierarchical, multi-level circuit description with the ability to generate tests at the switch-level.

One of the earliest hierarchical approaches to test generation is reported by Noujaim [74]. The method is based on partitioning a circuit into hierarchical elements and separately analyzing the fault propagation at each level in the hierarchy. To generate test vectors for a complete circuit, pre-determined fault characteristics are developed for each of a circuit's hierarchical elements. The fault characteristics of a higher level model are obtained by first determining a functional description for its lower level primitives and from them deriving a "test function" that defines all vector sets that achieve 100 percent stuck-at fault coverage. The low level test sets are then combined to form test sets for higher level modules, until the circuit's observable outputs are reached. All signal propagation is done with functional rather than gate-level descriptions, proceeding along boundaries between hierarchical elements. The approach considers only line stuck-at fault types, and is unable to model circuits below the gate-level.
Chandra and Patel present a hierarchical method for test generation that uses a strategy similar to that of PODEM [21]. The circuit model employed represents a circuit as a collection of interconnected logic blocks, with each logic block composed of primitive logic gates. The work uses a line stuck-at fault model. They discuss extensions to the PODEM algorithm to incorporate larger circuit primitives, and present HIPODEM, a test generation system that performs forward implication and backtracking over higher level modules. For hierarchical circuit descriptions, the algorithm makes more decisions requiring backtracks than it does for flat circuit descriptions, but backtracks are less costly than in the corresponding gate-level model. Results from test runs on adder and multiplier circuits indicate that the hierarchical test generation approach would be beneficial for large circuits.

Calhoun and Brglez discuss an algorithm called MODEM for hierarchical test generation based on module-oriented decision making [18, 19]. Their method captures modules and faults at the functional-level and the gate-level. A module describes the function of a combinational circuit at two-levels: as a minimized table of implicants and as a netlist of interior modules. At the lowest level of hierarchy, the minimized table of implicants is equivalent to the traditional And, Or, Nand, or Nor gate. Single stuck-at module faults are defined on module boundaries, and single stuck-at module-interior faults are defined on the lines that interconnect the interior modules. Whether a fault is considered to be a module or module-interior fault depends on what level of the hierarchy is being examined. MODEM uses the basic control strategies of PODEM during test generation. Results on large circuits show a consistent savings in CPU time over the traditional gate-level implementation, while maintaining an identical level of fault coverage.
Murray and Hayes describe an approach to test generation in which circuit modules and signals are manipulated symbolically using a hierarchical approach that accommodates high level descriptions [71]. In their approach, testing information for modules, including precomputed sequences of stimulus and response values, is contained in structures called test packages. The precomputed test information associated with each module is assumed to be able, if applied to the module in isolation, to reveal any internal fault of interest at the module’s output. These test packages are processed using techniques derived from artificial intelligence. Test packages are propagated along circuit paths in the same manner as conventional error or test vectors. The basic test generation routines are based on the D-algorithm. A limitation of the method arises when the algorithm attempts to propagate a test package through a module. Propagation is only successful when the module has an attainable state that allows the test package to be propagated substantially unchanged. This behavior is restrictive, and limits the number of circuits that the technique can handle. For circuits that will work, results for the approach indicate that the method greatly improves performance over conventional testing methods, displaying a potential speedup of up to three orders of magnitude. A further limitation of the approach is its inability to model circuit elements below the gate-level.

Dave and Patel propose the use of a functional-level testing methodology based on the generation of test vectors from a two-level representation of circuits under test [30]. The approach involves the generation of a two-level And-Or or Or-And implementation from a circuit’s functional description, followed by generation of test vectors using a PODEM based algorithm for the two-level implementation. No additional structural data is required for test generation. The approach assumes the presence of only stuck-at faults. Although the reported fault-coverage for the method is quite high, the effectiveness of the functional-level technique appears to decrease significantly
for large circuits due to the generation of extremely large test sets. Thus the approach is limited
to stuck-at fault test generation for circuits of small to moderate size.

2.3. Distributed Techniques

Test generation for VLSI circuits requires considerable computing resources, and much effort has
been expended to speed up test generation. One approach that has been used with success for
logic and fault simulation is the application of parallel and distributed computing techniques.
There is increasing evidence that the use of distributed processing can reduce the execution time
of test generation algorithms, but work in this direction has been limited and the problem is poorly
understood.

In this research, the term distributed processing refers to the use of a number of loosely coupled
processors working on the same problem to achieve a common goal. The characteristics of
distributed processing differ from those of tightly coupled parallel processing. In a parallel
processing system, the time required for communication between processors is comparable to local
memory access time. In a loosely coupled distributed system, communication costs are relatively
high compared to local memory accesses. A number of issues influence the performance of a
distributed application. Some of these factors are briefly mentioned below.

1. **Available Parallelism:** The amount of potential parallelism that is inherent in the
   application under consideration.

2. **Task Partitioning:** The decomposition of the problem domain into a set of smaller
   subproblems.
3. **Task Allocation:** The assignment or mapping of the partitioned subproblems onto available processors.

4. **Interprocessor Communication:** The messages that must be exchanged between processors for communication and synchronization, and the resulting communication delay imposed by the interconnection structure relative to the speed of the processors.

5. **Control Technique:** The technique used to coordinate the computation and communication activities of the processors used in a distributed application.

The first issue listed above is dependent on the application, while issues two and three are determined by the implementation used. The fourth parameter depends on the characteristics of the distributed processing system and how well the second and third issues are addressed. The next section discusses control techniques in greater detail, and the following sections review previous work in distributed simulation and parallel test generation that is applicable to this research.

### 2.3.1. Control Techniques

The major investigation into distributed techniques for switch-level algorithms has been in the area of simulation. The four techniques that have proven to be popular for distributed simulation are (1) the compiled-mode, (2) the centralized time event-driven, (3) the Chandy-Misra algorithm, and (4) the Time Warp algorithm. Each of these techniques is briefly described below.

In compiled-mode simulations [90], every logic element in a circuit is evaluated on each clock tick or time step. Parallelism is achieved by statically partitioning elements among processors and having each processor evaluate its assigned elements on every time step. The processors
synchronize at the end of every time step to guarantee that all processors are finished evaluating the values on all circuit nets. Since all processors evaluate the same elements during every time step, execution times are fairly predictable and load-balancing is straightforward. The main advantage of this approach is simplicity, a feature that makes it suitable for direct implementation in hardware. A severe disadvantage of the approach is that processors do a great deal of avoidable and unnecessary work, since typically only a small fraction of logic elements change state on any clock tick. Communication requirements for compiled-mode simulations depends on the number of elements that change state during a time step, typically a small number. The synchronization overhead of this method is, in contrast, quite high, as all processors must synchronize after every time step even if no useful work has been done.

In the centralized time event-driven approach, only those circuit elements whose inputs have changed are evaluated on a clock tick. This avoids the redundant work done in the previous algorithm, however the notion of the global clock and the need to synchronize all processors at the end of every time step limits the amount of concurrency [90]. The approach suffers from contention for global event queues. This problem is due to the small "problem size" of each unit in the queue, and the need for atomic queue updates. Processors typically spend as much time accessing the queue as they do updating nodes and elements, making the communication overhead significant. In addition, there are typically only a small number of available events at every time step. Centralized time approaches work efficiently on multiprocessors with on the order of 10 nodes, but for larger machines they fail to effectively take advantage of the available parallelism.

The Chandy-Misra algorithm is the approach that has generated the most interest in recent work [24]. This distributed-time, discrete event algorithm views simulation as a sequence of parallel
computations. This approach allows each processor to maintain a local clock, and processors communicate with each other using time-stamped messages. Processes maintain synchronization solely by this exchange of time information within messages. This approach has the advantage that there are no centralized resources to impede the speedup of a distributed system as more processors are added. This allows for the simulation of very large models that would not fit within the memory constraints of a smaller system or would cause serious performance degradation in a virtual memory system. This algorithm can be readily implemented in a loosely coupled, distributed processor system. The major factor limiting concurrency with this algorithm is the large number of "deadlocks" that occur during execution, effectively forcing global synchronization to be performed. Deadlock occurs when the simulation halts because no elements have pending events on all inputs. Deadlock resolution is achieved by updating the clock values of all elements and restarting the simulation.

The Time-Warp algorithm, proposed by Jefferson, is a distributed-time, event driven simulation technique in which the processor simulation clocks are allowed to advance or roll forward until conflicting time information appears [50, 51]. The arrival of a message with a time-stamp earlier than the current process time indicates that processing has proceeded too far, and that results obtained later than the incoming time stamp are invalid. Therefore, Time-Warp necessarily involves rollback and state storage overhead because processor states must be saved so that simulation can revert to a previous state when event precedence is violated. Time-Warp is relatively difficult to implement compared to simpler, more conservative methods, and may demonstrate excessive memory requirements if the amount of state information is large.
2.3.2. Distributed and Parallel Simulation

A great deal of work has been reported in the area of distributed logic simulation. Some results applicable to distributed test generation are presented here.

Arnold used the Time-Warp algorithm to implement an asynchronous switch-level simulator [6]. His approach statically partitions circuits, giving each processor a separate section of a circuit. The processors then proceed by periodically sending messages of the generated events and receiving messages sent as a result of events on other processors. Processors that receive messages in their "past" rollback the state of their portion of the circuit to that time and continue from that point. Results showed that a speed-up of four over the uniprocessor version was obtained with a six processor machine. The performance is primarily limited by the need to detect and process rollbacks. In addition, the complicated rollback mechanism produces complex interprocessor communication relationships and a significant state storage problem.

Soule and Blank developed and implemented three parallel algorithms for logic simulation on a general purpose shared-memory parallel machine [90]. The algorithms were tested with circuits modeled at a variety of levels, from gate-level to functional-level. They present results for a traditional event-driven simulator, a unit-delay compiled-mode simulator, and an asynchronous simulator using an algorithm similar to the Chandy-Misra algorithm. The synchronous, event-driven approach achieves speed-ups of 6 to 9 with 15 processors. The synchronous, compiled mode algorithm achieves speed-ups of 6 to 13 using 15 processors. The asynchronous simulation technique gave the best absolute performance, executing between 1 to 3 times faster than the conventional event-driven algorithm using a single processor. Depending on the circuit, the asynchronous algorithm achieved 10 to 20 percent better utilization with speedups ranging from
7 to 11 using 15 processors. The results show that a general-purpose shared memory machine can be used efficiently for simulations at the gate-level through the functional-level.

Duba et al. describe a distributed fault simulator implemented on a loosely-coupled network of general purpose computers connected via a local area network [32]. A hierarchical fault simulation approach based on the CHIEFS system described earlier is used to partition faults. The hierarchical partitioning algorithm creates a series of circuit representations or partitions, each of which is a complete representation of the circuit. Each partition contains a unique set of faults for simulation. Circuits are represented by logic gates at the lowest, primitive level, and by functional blocks, if available, at higher levels. Results show very good speedups for the distributed implementation running on 1 to 8 processors, and indicate that distributed fault simulation can be effectively used for fault simulation of very large circuits.

In work that explores the suitability of the Chandy-Misra algorithm for digital logic simulation, Soule and Gupta characterize the parallelism and deadlocks inherent in distributed logic simulation [91, 92]. The types of deadlocks that occur during gate-level logic simulation are characterized, and a method is proposed to reduce the number of deadlocks that occur during simulation, increase the average parallelism available, and improve performance. Results showed that the Chandy-Misra algorithm does extract more parallelism or concurrency than a conventional event-driven approach, where concurrency is defined as the number of logic elements that could be evaluated in parallel if infinite processors were available. By reducing the amount of time spent on deadlock resolution, the Chandy-Misra algorithm gives significantly better parallel performance than a traditional parallel event-driven algorithm. For the results given, the Chandy-Misra
algorithm outperformed the centralized-time, event-driven algorithm by a factor of 2 to 15 using up to 60 processors.

### 2.3.3. Parallel and Distributed Test Generation

The successful application of parallel and distributed computing techniques to increase the performance of logic and fault simulation has prompted researchers to consider similar approaches for accelerating test generation. There is increasing evidence that the use of parallel processing can effectively reduce the execution time of test generation, and work in this area has proceeded in a number of directions.

*Fault* parallelism [77, 40, 69] for test generation is analogous to pattern partitioning for logic simulation, and is perhaps the most obvious and simplest scheme for exploiting parallelism in test generation. In this approach, the fault set for which tests will be generated is distributed across available processors. Each processor then independently generates test vectors for the faults assigned to it. To maximize throughput, the load on each processor should be kept as similar as possible by partitioning and distributing faults carefully. This approach has the disadvantage of requiring enough memory for each processor to contain the entire circuit representation along with the test generation algorithm, a potentially enormous amount of information. An advantage is its low communication requirement between processors. This method has the potential for nearly linear speedup.

Motohara et al. [69] describe an approach combining fault partitioning with *task* partitioning to generate tests for combinational circuits described at the gate-level. The approach uses an algorithm based on the PODEM algorithm, and achieves a speedup in test generation by (1)
distributing independent faults and tasks to separate processors, and by (2) collecting test patterns for the distributed faults simultaneously with test generation. The strategy was implemented on a distributed memory or multicomputer system and results indicate that test generation contains a high degree of parallelism, and that a significant acceleration of test generation is possible using parallel techniques. The use of parallelism also made it possible to generate tests for faults that are extremely difficult to test using a conventional sequential test generation algorithm.

*Heuristic* parallelism [22] is an approach that relies on the fact that faults dropped due to excessive back-tracking by one heuristic method are often detected by the use of a different heuristic. By using a small time limit for each fault, the algorithm can quickly switch to different heuristics as faults are dropped by the previous test generation pass. Faults are passed to each heuristic measure in turn, until all have been tried. Only about five or six heuristics are used with algorithms such as PODEM: (1) Random, (2) Distance, (3) COP, (4) SCOAP, and (5) Camelot are the most common [76]. Heuristic parallelism can be implemented in one of two ways. In the first scheme, composite heuristic parallelism, fault partitioning is used to distribute faults between processors and every processor computes tests for its faults using the full composite strategy. In the second scheme, concurrent heuristic parallelism, each processor uses a different heuristic to guide the search for a single fault. A disadvantage in this case is that parallelism is limited by the number of available heuristics. Superlinear speedups due to search anomalies are possible with this approach, but synchronization and communication requirements are much higher.

*Search* parallelism is a parallel branch and bound technique that speeds up backtracking by exploring different portions of the search space concurrently [69, 78]. By distributing the search space over the available processors and having each processor search its own subspace
simultaneously, the time required for backtracking can be reduced. Search space allocation can be accomplished by simply dividing the search space into equal parts, or by using heuristics to choose portions of the search space where the probability of finding a solution is higher. In the first case some processors may expend effort in a useless search in non-solution areas, but in the second case search spaces may overlap if precautions are not taken to keep them disjoint, leading to redundant searching. Search parallelism is particularly suited for generating tests for hard-to-detect faults, and can produce superlinear speedups due to search anomalies. This approach has been used successfully to implement parallel test generation on a 16-node Intel iPSC/2 hypercube [97], but the use of a centralized scheduler would cause a bottleneck in the performance of the parallel algorithm for larger numbers of processors.

In circuit partitioning, the circuit under test is divided into distinct portions and distributed among processors. Every processor analyzes and generates tests only for faults in its own subset of the circuit, and processors cooperate to generate a test for any particular fault. Circuit partitioning avoids the need for every processor to maintain a complete representation of the circuit under test, and thus offers the possibility of test generation for very large circuits. A disadvantage is that the degree of parallelism decreases as test generation proceeds and processors complete handling of faults within their subset of the circuit. Another disadvantage is the very high communication requirement between processors. Although circuit partitioning techniques have been used in some massively parallel approaches to test generation [57, 20], these approaches are limited in the size of circuits that can be handled. Finally, simulation parallelism is a subset of circuit parallelism, and refers to the use of parallel logic simulation techniques to accelerate the forward implication phase of test generation.
2.4. Summary

The classical approach to test generation, consisting of a line stuck-at fault model coupled with a gate-level circuit description, is inadequate for generating tests with very high fault coverage for CMOS circuits. Such an approach cannot represent the structure of CMOS circuit designs or model the faults that result from physical failure mechanisms. With CMOS, logic functions are realized in a topological fashion by constructing networks of N- and P-channel transistors. Physical failure mechanisms in CMOS are also topological, and can be represented as perturbations in the good circuit topology. The switch-level model employed in this research allows for the direct representation of realistic CMOS faults such as short, open, transistor stuck-on, and transistor stuck-off faults in addition to stuck-at faults.

In addition to the representation of faults and circuits, this research also considers an alternative technique for the detection of faults, made possible by the unique properties of CMOS circuits. In particular, short faults, or bridging faults, require a different strategy for test generation than stuck-at faults. Tests can be generated for static CMOS circuits to detect bridging faults using variations in quiescent supply current, making possible a dual-mode approach to fault detection. This research investigates \( I_{\text{DQ}} \) testing for CMOS circuits, and presents a method that addresses the primary issues involved in test generation, including fault sensitization, propagation of faults to observable outputs, and back-wards implication of logic values to controllable inputs.

The use of a switch-level circuit representation necessarily increases the amount of structural circuit information that must be represented. Previous approaches to this problem have generally modelled circuits with flat, switch-level descriptions and thus require enormous memory capacity.
for large circuits. In addition, the time needed to process a circuit with such an approach increases exponentially in circuit size. In this research the computational overhead imposed by the use of switch-level techniques is counteracted by the formulation of a modular, hierarchical approach to circuit representation.

A circuit partitioning approach to reducing the run-time complexity of test generation via parallel processing is the one method that has received little attention in the literature to date. This research explores the feasibility of using a circuit partitioning approach to deal with large circuit sizes and accelerate the test generation problem. A set of theoretical models that characterize distributed test generation are developed, and the effectiveness of a circuit partitioning approach is evaluated with respect to its expected performance in a distributed processing environment.
Chapter 3. Object-Oriented Test Generation

This chapter presents a case study describing the application of object-oriented design in the development of a hierarchical test generation application implemented using C++. In this work, traditional test generation is extended using an object-oriented approach to support the testing of circuits containing both structural hierarchy and elements modeled using multiple levels of abstraction. In doing this, the primary objectives were to produce a easily maintainable system, provide an extensible framework for test generation supporting the straightforward incorporation of new types of circuit primitives and faults, and retain the same level of computational efficiency that can be achieved using a procedural language such as C.

Along with a presentation of general issues arising during the development process, this chapter discusses how specific design decisions were impacted by the choice of C++ as an implementation language. Characteristics of the object-oriented hierarchical test generation application, such as expandability and run-time efficiency, are compared to those of a standard gate-level test generation program implemented using C and a procedural design approach.

Object-oriented software development is an involved process, involving analysis, design, and implementation activities that are interleaved in an iterative development process. The
development cycle may be repeated many times, dealing with successively lower levels of detail and abstraction. The resulting object-oriented application thus emerges as a product of successive refinement, driven by a spiral development process. However, to simplify the presentation, the object-oriented development of the hierarchical test generation application is described as a linear, orderly phenomenon.

3.1. Object-Oriented Analysis

One of the fundamental principles of the object-oriented paradigm is that a computer program should be written as a model of specific aspects of reality. The closer a program corresponds to a real-world environment, the easier it should be to understand, implement, and maintain. Object-oriented languages support this approach by providing convenient mechanisms for directly representing the important concepts in applications.

In C++, the key mechanism for accomplishing this is the class, a language facility for creating user-defined types that encapsulate important concepts and the operations that are associated with them. A class represents a set of objects, or concrete entities, that have a common structure and behavior. An object is then a single instance of a class. Careful specification of classes allows the programmer to logically partition an application into manageable pieces that correspond to familiar concepts in the problem domain. The problem of how to best begin design of an object-oriented application then becomes one of finding and isolating application concepts that can be practically modeled as classes.
Object-oriented analysis [12] addresses this problem, and is an appropriate and often necessary front-end activity for object-oriented design. Object-oriented analysis begins with examining the problem at hand, and identifying the essential objects and entities that exist in the problem domain, along with the ways that these objects behave and interact. Identifying key concepts focuses attention on the important components of a system that are highly relevant during application design, and prevents consideration of (initially) superfluous implementation details. The goal here is to accurately describe the behavior of a real-world system, considering only enough structure to support that behavior. Object-oriented analysis is highly domain-specific, and not surprisingly, its end product is often a list of key abstractions that exactly matches the vocabulary of the problem domain. This was especially true when analyzing the test generation problem. The following section briefly describes a widely used test generation algorithm, and following an example of its application to a gate-level circuit, presents an object-oriented analysis of hierarchical test generation.

3.1.1. The Test Generation Problem

Test generation is most commonly performed using a gate-level circuit description and the single stuck-at fault model. With this fault model, a single gate input or output is assumed to be permanently set to logic 0 or logic 1 in the presence of a fault. If a line in the circuit is stuck-at 0 (1) when it would really be a 1 (0) in the good circuit, the faulty value on that line is represented by the symbol $D$ ($\overline{D}$), meaning discrepancy. To detect the presence of a stuck-at fault in a circuit, a test generation algorithm must perform three steps:

- Sensitize or expose the fault by driving the faulty line to the opposite logic value from the fault value, generating a $D$ or ($\overline{D}$) on that line.
- Determine how to assign values to the primary inputs of the circuit to ensure that the fault site is sensitized properly.

- Propagate the $D$ or $\overline{D}$ at the fault site to a primary output of the circuit, where it can be observed, indicating the detection of the fault.

Performing the operations needed to accomplish these steps requires some sort of control algorithm. For this research a standard method for performing test generation for combinational circuits, the PODEM (Path Oriented Decision Making) algorithm, was chosen [43]. The basic operations performed by PODEM are illustrated using a simple example; a more detailed description of PODEM is given in [43].

Consider the two-gate circuit shown in Figure 1, with primary inputs A, B, and C, and a single primary output Q. All lines in the circuit start out with a value of $X$, or Don't Care. A stuck-at 0 fault is shown on one of the inputs to the Or gate $G_2$. To sensitize this fault, the output of the And gate $G_1$ must be set to a logic 1. To accomplish this, PODEM employs a backward justification procedure that examines individual gates, and attempts to justify the desired objective value on the output of a gate by specifying an appropriate value on a single input. The term backtracing is used to denote the process of repeatedly justifying objective values backwards through a circuit until a primary input is reached.

For the circuit shown in Figure 1, the backtrace procedure determines that to produce a logic 1 on the output of the And gate $G_1$, a logic 1 should be applied to an input, say line A. Since A is also a primary input of the circuit, the backtrace procedure is done. At this point, PODEM uses a forward propagation procedure to calculate the good and faulty logic values generated on circuit
Figure 1. (a) Example circuit with stuck-at 0 fault, (b) Valid test that detects fault.
lines as a result of assigning a 1 to line A. The process of determining the output of a particular logic gate based on its inputs is referred to as forward implication. In PODEM, forward propagation is performed after every primary input assignment. For this example, forward implication on gate $G_1$ shows that the output remains at $X$, so the backtrace procedure is invoked a second time with the same goal of setting the output of $G_1$ to logic 1.

Backtracing now sets input B to logic 1, and forward implication shows that the output of $G_1$ becomes 1, successfully sensitizing the fault site to $D$. The task now is to propagate the $D$ value on the input of gate $G_2$ to $Q$, the observable output of the circuit. This is accomplished using a fault propagation procedure, that can appropriately set the unassigned input of gate $G_2$ so as to make the fault effect visible on the output. This procedure sets input C to logic 0, and forward implication shows that the set of values $ABC = 110$ is indeed a valid test, producing a $D$ at the output of the circuit.

Test generation for realistic circuits is complicated by the presence of something not present in the above example, reconvergent fan-out. Reconvergent fan-out is present when a signal line branches out to the input of several distinct gates, and then at some point in the circuit the output signals of different gates influenced by the same fan-out source come back together and combine as inputs to a gate. Reconvergent fan-out causes the backtrace procedure to be heuristic in nature, by introducing dependencies in the values that must be assigned to circuit wires and causing conflicts during backtracing. This necessitates the incorporation of a backtracking search procedure into test generation that can resolve conflicts. PODEM performs backtracking at the primary inputs of a circuit, generally by using a last-in first-out stack containing primary input assignments.
Although PODEM has been described here in terms of its operation on a gate-level circuit, it has been successfully applied at a variety of levels of circuit abstraction [21, 25]. Consequently, the PODEM algorithm can be viewed as a general test generation approach rather than a solution for specific types of circuit descriptions. Being able to successfully exploit PODEM in this manner was a primary motivation for moving to an object-oriented approach.

### 3.1.2. Identifying Fundamental Abstractions

In the above description of gate-level test generation using PODEM, a number of key entities and concepts were mentioned repeatedly, immediately suggesting a number of candidate objects for consideration during object-oriented analysis. Many of these objects correspond to real-world, physical objects encountered in test generation:

- **Gate:** A gate implements a logic function such as And, Nand, Or, or Invert, producing an output value as a result of one or more input values.

- **Wire:** Wires, or circuit lines, are used to interconnect gates and transfer the signals that gates operate on.

- **Circuit:** A circuit is made up of a collection of gates and wires, arranged in a logical manner to perform some useful function. Certain wires are distinguished at the circuit level as primary input or primary output wires.

Other objects correspond to concepts or algorithms not possessing a tangible physical representation:

- **Logic Value:** In this work five logic values, 0, 1, X, D, and D, are used by PODEM during test generation. Two of the values, 0 and 1, do correspond to specific conditions or voltages in physical wires, but the other three are strictly conceptual values.
• **Fault**: A fault represents a physical defect or malfunction in a circuit. Faults are included in this section because faults such as the single stuck-at fault often model behavior rather than the actual physical defect mechanisms that occur in circuits. Faults alter the behavior of elements in the good circuit.

• **Control Algorithm**: The test generation control algorithm, PODEM in this case, contains the knowledge needed to generate a test given a particular fault. The complexity of the control algorithm is a good indication that it could probably be decomposed further into cooperating sub-objects, but further analysis is better postponed until later in the development cycle.

This list is not necessarily complete, but does include the major concepts encountered during this first-order analysis. At this point it is important to consider how the introduction of hierarchy into the problem domain affects the set of abstractions. There are two forms of hierarchy that can be taken advantage of during test generation, based on the concepts of *structure* and *abstraction*.

Structural hierarchy is commonly present in circuits, due to the use of hierarchical design methodologies. In a hierarchical design, circuit primitives (gates) are combined to form useful subcircuits, which are further combined to form additional larger sub-circuits, and so on until the highest level of the circuit is reached. VLSI design tools take advantage of structural hierarchy by storing the definition of unique subcircuits only once, and referring to that single copy wherever an instance of that subcircuit is needed. This results in an enormous savings in memory requirements when representing a large circuit, as long as the circuit description never needs to be completely flattened out. This advantage, as well as the ability to directly accept circuit
descriptions produced by hierarchical design tools, is a powerful motivation for utilizing hierarchy in test generation.

Although using structural hierarchy reduces memory requirements, this approach by itself does not provide any opportunity for increasing the speed of test generation. To do this, it is necessary to include additional modeling information for subcircuits in the form of multiple levels of abstraction. For example, a collection of gates implementing a 32-bit parallel multiplier can be represented at a higher behavioral level as a multiply function that accepts two 16-bit arguments. At this higher level, the multiply operation can be computed significantly faster than at the gate-level, given suitable input values. Given more than one level of abstraction for a sub-circuit, use of the highest available level is desirable whenever possible, to avoid unnecessary computational complexity.

Given a circuit description containing structural hierarchy and multiple levels of abstraction, implementation of the test generation algorithm obviously becomes much more complicated. To take full advantage of the redundant circuit information while keeping memory requirements manageable, it becomes necessary to perform dynamic reconfiguration of the circuit representation during test generation. In addition, different subcircuits need to be processed in radically different ways. Despite this increased complexity, it is still desirable to be able to view a circuit simply as a netlist of interconnected circuit primitives that have properties that enable them to be processed during test generation. To support this notion, the concept of Modules is introduced.

- **Module**: A Module is a generic or "black box" representation of a subcircuit that allows that subcircuit to be treated by a general PODEM algorithm in much the same manner that a logic gate is handled during gate-level test generation.
It is not possible to describe the composition of a Module in more detail without getting into design and implementation issues. At this point, object-oriented analysis has produced a list of test generations abstractions that can be used directly at the start of object-oriented design. The entities and relationships identified during this object-oriented analysis are represented in the object diagram shown in Figure 2.

3.2. Object-Oriented Design

The goal of object-oriented design is to produce a clear and understandable overall structure for a program. Properly done, this design creates a logical framework for combining individual pieces of code in a manner that is flexible, extensible, and resilient to change. In C++, classes provide the means of encapsulating aspects of an application that are likely to be changed or extended.

Object-oriented design begins by taking the list of key abstractions produced during object-oriented analysis, and then identifying how classes and objects can be used to clearly represent these abstractions and implement the mechanisms that provide the behavior observed during analysis. Each class is given the responsibility of maintaining all information relating to a single abstraction. The impact of modifying and extending abstractions is then primarily limited to implementing changes in single classes. It is important to note that a key concept identified during object-oriented analysis may turn out to be either a class, an object, or just an attribute of certain objects.
Figure 2. Object diagram for the test generation problem.
The object-oriented design of the hierarchical test generation application is described as a multi-step process as suggested in [12] and [94]. Just as with the overall analysis, design, and implementation process, the four design steps discussed here were actually interleaved in an incremental and iterative fashion as development progressed, but are described as a linear process.

### 3.2.1. Specify Classes

The first step in object-oriented design is to specify the classes and objects that will be used to model both application concepts and the designer's basic understanding of the application behavior. In many cases, this step consumes much less time than the subsequent three steps, especially when design is preceded by object-oriented analysis.

For test pattern generation, we can immediately specify classes to represent some of the straightforward concepts identified during analysis, giving us Gate, Wire, Circuit, Fault, and ControlAlgorithm classes. A LogicValue class is not defined to represent the five-valued logic model; logic values are instead represented as an *enum* (enumeration) construct. This decision was made for reasons of efficiency, since the manipulation of logic values is a very low-level operation that is used extensively in test generation. Using enumeration represents the logic values as named integer constants, a low-overhead data type in C++. The cost of such a decision is some loss in generality, but the tradeoff was acceptable in this situation. In any case, in the initial design of an application it is not necessary to model every minute detail as a class. The goal is to model concepts at an appropriate level of abstraction.

In addition to directly modeling real-world physical entities, classes are often created as artifacts of an application. In this case, classes represent concepts related to the implementation techniques
used by a designer. An example of this need for classes arises when trying to represent the structure of circuits containing hierarchy.

To model the behavior of specific subcircuits, a Module class is used to implement the "black box" approach discussed in the previous section. Each object belonging to the Module class then contains all information needed to completely describe the characteristics of a particular subcircuit. To support hierarchy and avoid replicating this information each time a Module appears in a circuit, an Instance class is introduced to represent the actual usage of Modules and their physical arrangement in a circuit. Thus Modules contain information such as the number and type of inputs and outputs, a netlist of circuit primitives, and higher level descriptions. Instances contain an instance name and pointer to a Module object, controllability and observability data necessary for test generation, and information concerning the connectivity with other instances. Modules can contain Instances of other Modules, allowing the representation of arbitrarily partitioned hierarchical designs.

The need for additional classes will certainly surface during the design process. However, the classes specified up to this point can be expected to carry through the entire design process.

3.2.2. Specify Operations

In the second step of design, each class discovered so far is refined by specifying a set of complete and useful operations that can be applied to that class. Each class should be viewed from the perspective of its external interface to identify the things that can be done with each object of a class, and the things that each object is allowed to do to other objects. Unfortunately, deciding on the protocol of every object can be quite difficult, and this step is typically much
harder than the first. Concentrating on defining a minimally useful set of operations for each class is generally a good strategy.

For the classes that represent physical entities encountered in test generation, the set of useful operations is somewhat predetermined. For example, a Circuit object should provide us with the ability to directly observe and control the primary input Wires, but only observe the output Wires. There should be some means of traversing the netlist of circuit primitives or Instances that make up the circuit. The Wire class should provide a means for setting and observing the logic value of Wire objects, and enumerating the Instances that are fed by Wires that fan-out. Deciding on the set of operations that should be supported by such classes is not difficult.

What is more interesting in the context of hierarchical test generation is determining a means of defining operations on Modules so that the interface between the ControlAlgorithm object and Modules is consistent across different types of Modules. Ideally, any Module could be treated as simply as a logic gate. The previous description of the gate-level PODEM test generation provides insight into the three primary operations that must be supported by a generic Module in order for it to be uniformly processed by a generalized PODEM algorithm:

- **Forward implication:** given a set of values on the inputs of a Module, determine all implied values on Module outputs.

- **Backward justification:** given an objective output value, and possibly a partially specified set of input values, decide how to set an unassigned input so as to generate the desired output objective.
• Fault propagation: given at least one faulty value ($D$ or $\overline{D}$) at an input and at least one unknown value ($X$) at an output, decide how to set an unassigned input so as to propagate the faulty input value to an output.

Given the ability to invoke these operations on a netlist of miscellaneous Modules, a PODEM ControlAlgorithm object can perform test generation in the same manner as it would for a netlist of simple logic gates. However, depending on the type of module under consideration, the complexity of performing these operations varies considerably. For example, forward implication for a logic gate can be performed using a simple table lookup. Efficient implication of input values for a large netlist of circuit primitives, on the other hand, requires the use of a technique such as event-driven simulation.

The dependence of these three principle Module operations on the specific type of Module being processed is a strong indication that these operations should probably be implemented as virtual functions. Virtual functions are a C++ mechanism for providing a set of type-dependent operations for the most general or prototype version of a concept, expressed as a base class. The implementation of these operations can then be refined for particular special cases of the base class, represented in C++ as derived classes. This is discussed in more detail in the next section.

One question that becomes important at this stage of design is: how should the ControlAlgorithm object and the various Modules contained in a circuit interact to ensure that Modules are processed properly? With most gate-level test generation applications, the control algorithm contains embedded knowledge of how to imply or justify a logic gate. Typically, only a very limited set of gate types is allowed, such as And, Nand, Or, Nor, Inverter, and Buffer. This approach breaks
down very quickly when trying to add new, different types of Modules. In any case, it violates a basic object-oriented concept, encapsulation, by making the Control Algorithm dependent on the details and internal state information of Modules. This information should be the sole responsibility of the Modules themselves.

One solution to this problem would be to have a Module describe itself to the Control Algorithm in some pre-arranged manner, telling the Control Algorithm what steps it would need to take to perform the three Module operations. This approach is perhaps the most general, but imposes an unacceptable cost in terms of both implementation complexity and run-time overhead.

A second solution would be to make Modules aware of the test generation environment maintained by the Control Algorithm object, and let Modules process themselves whenever the need arises. This is an improvement in terms of freeing the Control Algorithm from the responsibility of recognizing and operating on all possible types of Modules. However, making Modules aware of details of the test generation environment again violates encapsulation and the idea that a class should be responsible for a single set of well-defined operations.

The best solution for this problem is to simply make Modules completely responsible for performing implication, justification, and fault propagation for themselves. Modules are also responsible for maintaining a consistent internal state for themselves during test generation. This frees the Control Algorithm object from the need to understand the internal makeup of Modules, and the need to make complex decisions concerning changes in the circuit description.
A final issue that should be dealt with during this step of the design process is deciding where the responsibility for initiating dynamic circuit reconfiguration should be placed. During test generation for a fault, only the instances that contain the fault should be expanded to their low-level representation, and all other instances should be processed using their highest-level representation. Although the choice was not obvious in the early stages of design, the Fault class finally emerged as the best candidate class for encapsulating decisions regarding circuit reconfiguration. This works well because Fault objects necessarily possess some means for representing the physical locations of the faults being modeled. This physical location is also described in a hierarchical form, such as a path of Instance names, so the necessary information for determining which Modules must be expanded to lower levels is encoded in the internal state maintained by Fault objects. Thus a gate-level stuck-at fault affecting the gate "/instance_1/sub-instance_A/gate-name" must understand the proper level of abstraction to use for instance_1 and sub-instance_A (a netlist of module Instances), and be able to configure the circuit representation appropriately.

3.2.3. Specify Dependencies

The third step of object-oriented design is an extension of the previous step, and focuses on the continued refinement of classes by specifying the dependencies of each class on other classes. This corresponds to determining the exact relationships existing between the various concepts involved in a problem solution, and the interaction of objects in a system. The task of organizing the relationships between classes in a program is treated as a separate step in design due to its difficulty, as it is often a much more challenging process than the initial discovery of individual classes. The two key forms of class relationship that must be considered during design are uses and inheritance relationships.
The uses relationship arises when one class is a client of, or uses an object of, another class. This kind of dependency is often referred to as a has-a relationship, as opposed to an is-a or inheritance type of relationship. Uses relationships commonly arise with classes used to represent real-world objects. For instance, it is clear that a Wire has-a Fault, or that a Nand gate has-a Wire as an output. However, in cases where classes represent more abstract concepts, the type of relationship is not always so apparent. For example, does an Instance object use a Module, or is it a kind of Module? Both approaches could be made to work, but in this work the concept of a Module is clearly distinguished from that of an Instance representing its inclusion in a circuit. This approach supports hierarchy by only requiring a single Module object for each distinct type of subcircuit. This Module object is then "used," or referred to, by all Instances of that Module.

As is often the case, the uses relationship here is represented by embedding a pointer to another object inside the using object, and relying on a set of programmed conventions rather than specific language features. A revised object diagram reflecting the relationship between Modules and Instances is shown in Figure 3.

Inheritance is a higher level relationship than uses, and unlike the uses relationship, can be represented directly by constructs in the C++ language. Organizing classes using inheritance relationships takes advantage of two powerful tools for managing complexity, classification and hierarchical ordering. The idea here is to organize related classes in a tree structure, or directed acyclic graph, with the most general class at the root of the tree. Through inheritance, a new class that is similar to existing classes can be created by expressing only the differences between it and the classes it inherits from. New classes can re-use the code and member variables that implement similar operations in existing classes, eliminating the need to include redundant code for different class implementations.
Figure 3. Revised object diagram for the test generation problem.
In C++, the most general version of a set of concepts modeled with an inheritance tree is represented by a base class. More specialized concepts that inherit and redefine the behavior of this base class are then represented using derived classes. A base class can have more than one derived class, and each derived class can, in turn, serve as a base class for additional derived classes. Derived classes are said to have an is-a relationship with base classes, that is, a derived object is-a specialized version of a base object. Figure 4 shows the resulting tree-structured arrangement of related classes, or class hierarchy, that is created using inheritance.

To hide the differences between the classes contained in an inheritance tree so that client classes can interact with them in a uniform manner, C++ provides a dynamic binding mechanism based on virtual member functions. A base class can declare certain member functions as virtual, indicating that these functions are a uniform set of operations for manipulating instances of itself or of any its derived classes. Dynamic binding allows each class in a class hierarchy to transparently maintain a different implementation of any particular operation. When a client applies a virtual member function to an object of any one of the classes, it can do so without worrying about what type of object it is operating on. C++ automatically determines the class of the object and correctly invokes that class’s implementation of the virtual function.

The combination of inheritance and dynamic binding provides an ideal framework for simplifying the interface between the ControlAlgorithm class and the generic Module class in hierarchical test generation. The generic Module concept and the three primary Module operations are modeled using a base Module class with three corresponding virtual functions, and then specific types of Modules are represented by deriving classes from the base Module class. During test generation,
Figure 4. A tree structured class hierarchy.
the ControlAlgorithm object is then able to treat all Module objects as if they were objects of the base Module class. When the ControlAlgorithm object applies an operation to a Module, C++ automatically determines which class the Module object belongs to at run time, and transfers control to the correct implementation of the function.

There is a tremendous benefit in restricting the ControlAlgorithm class in test generation to only work with a generic base Module class. Since the ControlAlgorithm object relies on no knowledge of the internals of derived Module classes, new Module classes can be added without changing the ControlAlgorithm class. In fact, since the dynamic binding provided by virtual functions is performed at run time, it is not even necessary to recompile the code implementing the ControlAlgorithm class.

Figure 5 shows a Module class inheritance tree formed by including an example set of different Module types. The classes that are immediately derived from the base Module class provide default operations for behavioral (functional) descriptions, netlists of circuit primitives, logic gates, and pass transistor networks. These classes in turn serve as bases for more-derived Module classes. Certain classes, such as those in the Netlist subtree, are defined as abstract classes, meaning that they serve only as bases for more specialized Modules and cannot be instantiated as Module objects themselves. Others, such as the Adder and Xor Modules, possess a fully defined set of operations and are suitable for instantiation as testable objects. The actual module class tree used in this research to model switch-level standard cell subcircuits is shown in Figure 11 of Chapter 4.
Figure 5. Module class inheritance tree.
To provide an extensible fault model, the hierarchical test generation system also relies on inheritance relationships for classifying Fault types. Operations that depend on the specific fault type, such as circuit reconfiguration, are specified as virtual functions in the base Fault class. Figure 6 shows part of the fault class inheritance tree formed by the faults modeled in the test generation application used in this research. The complete fault class tree used to model bridging faults, based in part on a bridging fault classification scheme presented in [65], is shown in Figure 12 of Chapter 4.

### 3.2.4. Specify Interfaces

The final step of the object-oriented design process involves the definition of the interface or specification of individual classes. This is where the argument types of member functions used to implement operations on classes are considered and defined. Also, the exact details of the interfaces provided by base classes intended for use in class hierarchies are considered here, as they impact the interfaces of all derived classes.

The goal of this step is to complete the proper encapsulation of each class, by restricting all interaction with it by client classes to a publicly accessible set of member functions. This isolates client classes from the details of how a particular class represents the data it contains. This data, along with the algorithms a class uses, are referred to as the class’s implementation. For example, to represent fanout the implementation of the Wire class would include some means of storing a list of Instances that a Wire object feeds. Clients of the Wire class are given access to a Wire member function that allows them to iterate over this list. However, client classes should have no idea of how the list is represented; it could be either a linked list or an indexed array of Instance pointers. Hiding this knowledge allows the Wire class’s representation of the list to be
Figure 6. Fault class inheritance tree.
changed without affecting the operation of client classes.

Done properly, encapsulation helps ensure that changes to the implementation of a class will not cause other classes to become obsolete or malfunction. The ease of program development and maintenance is enhanced, since all code that directly modifies a class’s data is localized to the implementation of that class.

### 3.3. Implementation

The implementation phase of the object-oriented development process is the point where the internals of classes are first considered, taking an inside view of each class to decide how its behavior should be implemented. Ideally, the task of actually implementing the application would simply involve taking the products of analysis and design (classes, semantics, and relationships), and coding the implementation of each class appropriately.

Unfortunately, for all but the most trivial key abstractions and mechanisms, the designer typically discovers that it is necessary to return to the first step of design and repeat the process when considering the inside views of classes. The design process then becomes iterative, with each iteration focusing on increasingly lower levels of abstraction. This was found to be the case during development of the hierarchical test generation application.

As lower levels of abstraction were considered during design and development, the number and variety of classes used to implement the test generation application naturally increased. Of these,
only a few significant applications of classes are described here. The use of classes proved to be quite appropriate for purposes other than classification and the representation of real-world concepts. For example, classes were used in the test generation application to:

- Hide the complexity of involved algorithms; the ControlAlgorithm class is the highest-level example of this. Clients of this class have no idea of how the ControlAlgorithm actually generates a test. They can only supply a fault to be tested, and then wait for a result.

- Provide an encapsulated, object-oriented interface to existing procedural libraries. During development, libraries of routines written in C for a previous test generation program were used in intermediate prototype versions of the object-oriented test generation application. These routines were then gradually replaced as the design of the new application stabilized, with minimal impact on the overall system, thanks to encapsulation.

- Isolate design decisions that involved machine or operating system dependencies. Changes due to differences in operating system calls are localized to the implementation of a few classes. This eases the porting of the application between different environments.

- Combine basic data structures and associated algorithms to form abstract data types that are both highly re-usable and extremely valuable during development. Typical examples are container classes such as LinkedList, Stack, PriorityQueue, and HashTable.

The last use of classes is strongly supported by many language features in C++ that enable the convenient incorporation of user-defined data types in programs. The generic container classes were designed to take advantage of templates, a recently introduced feature of C++ that provides the ability to implement parameterized types. A template is essentially a form of generic class
declaration that accepts a type argument. The template is then used to specify the creation of specialized class types in much the same way that classes specify the creation of individual objects. Template classes are used in the test generation application to provide specialized container classes such as Stack<Wire> and HashTable<Instance>. The use of specialized container classes avoids the computational overhead that would be incurred by using generic container classes that accept homogeneous collections of objects and rely on run-time type checking to distinguish object types. Homogeneous collections of objects are common in many object-oriented applications, but are not necessary, or even particularly useful, in the test generation problem.

The current implementation of the hierarchical ATPG application supports the use of three levels of abstraction for representing Modules, a functional description, a netlist of circuit primitives, and a switch-level or transistor-level description for representing CMOS circuits. To obtain a significant speed up in test generation, it is critical that the functional model included in a Module class can perform the three basic test generation operations faster than the other, lower levels can. Although a functional description can be represented in many ways, such as a parse tree derived from a hardware description language, the test generation implementation used in this research emphasizes efficiency and requires that all functional operations for Modules be specified as compiled C++ routines. The development of ad hoc C++ routines to implement functional behavior for different Modules is quite reasonable, as the number of different Modules used for design is often small. Also, the inheritance relationship used to classify Modules in a class hierarchy provides excellent support for code re-use between Modules.
During the implementation phase of development, it was discovered that the Module class hierarchy shown in Figure 5 does not completely address the issue of how to represent Modules that contain more than one level of abstraction. For example, where does a Multiplexer Module containing functional, gate-level, and switch-level descriptions fit into the inheritance hierarchy? The answer is that it does not; an additional level of inheritance relationships is necessary to describe such a Module.

Figure 7 demonstrates the two forms of inheritance that can represent multiple levels of abstraction in this case. The first, single inheritance, is the simpler and more common form of inheritance used in C++ in which each derived class inherits from a single base class. The second, multiple inheritance, occurs when a derived class inherits from more than one base class simultaneously. The ControlAlgorithm object can operate with Modules modeled with either form of inheritance, since its interface is dependent only on the base Module class. The test generation implementation used in this research uses only the single inheritance approach.

A final important issue that arose during implementation of the ATPG application was that of maintaining the state of the circuit during changes in the circuit representation. The state of each Instance object in a Circuit actually depends on three things: the level of abstraction currently being used to model the Instance, whether or not the Instance is affected by a Fault, and the set of logic values assigned to the Wires of the Instance. This Instance state clearly cannot be stored in the associated Module object, since each Module may be referred to by multiple Instances. Also, state information cannot be stored internal to the Instance, since the Instance may itself be used by a higher level Module to support structural hierarchy. The result is that dynamic state information cannot be stored in the hierarchical circuit representation at all.
Figure 7. Alternatives for representing multi-level behavior via inheritance.
To solve this problem, information that is unique to each instance is stored in a tree data structure that contains all state information for the circuit. This state tree is dynamically reconfigured by Fault objects to reflect a suitable circuit representation for every fault considered during test generation. Figure 8 illustrates an example state tree generated during testing for a switch-level fault.

3.4. Comparison

To evaluate the ease of maintenance, expandability, and run-time efficiency of the object-oriented hierarchical test generation application, it is compared with a standard ATPG program implemented using C and a procedural design approach. The C version is limited to generating tests for gate-level circuit descriptions, so gate-level test generation is used as the basis for comparison.

3.4.1. Maintenance

With the object-oriented ATPG application, the early consideration of flexibility, extensibility, and portability issues in the design process had the direct consequence of producing a highly maintainable implementation. This was especially evident during performance tuning, as different data structures were tried to minimize the computational overhead of various aspects of the implementation. In one case, three Set container classes in the ControlAlgorithm class were implemented using hash tables based on the names of objects. After profiling the program execution revealed that these hash tables were a significant source of overhead, the Set implementations were replaced with a version based on arrays of Boolean flags. The only
Figure 8. Tree structure for representing hierarchical circuit state.
modifications required outside of the Set implementations were three one-line changes in the class declaration of the ControlAlgorithm class, and another three one-line changes in the ControlAlgorithm implementation. These simple changes improved the run-time performance of the program by an average of 34 percent for typical circuits.

In comparison, the ease of maintenance of the procedural ATPG implementation was relatively poor. A change of the type just described would require extensive modifications to several data structures and a significant amount of code. Even the smallest modifications tend to affect seemingly unrelated procedures in the program, requiring the maintainer to scan the bulk of the program to understand the implications of a change.

3.4.2. Extensibility

The procedural ATPG program was originally written without support for Exclusive-Or gates, so to evaluate its expandability the task of adding support for a new Xor gate type was chosen. The program’s use of a common technique in procedural programming made this task challenging. Logic gates are represented using a structure with a type field denoting the kind of gate being described, sometimes called a discriminated record in languages other than C. These gate structures are processed by switch statements distributed in the program wherever operations are performed on gates. Each switch statement contains case statements to handle each possible gate type. Consequently, the code that deals with each logic gate is also scattered around in the program, and to add a new Xor gate type, it is necessary to find and modify each and every switch statement used to process gates. These modifications are not localized, and making them all is both tedious and error prone.
In contrast, the object-oriented C++ ATPG implementation replaces the switch statement technique using class hierarchies and virtual functions. A derived class in the Module class inheritance tree represents each type of logic gate, and virtual functions replace each switch statement. The functionality provided by each switch statement case is provided by a virtual member function in one of the derived Module classes, with the result that all code that handles a particular gate is contained in a single Module class. Adding an Xor gate type is simply a matter of adding an additional derived Module class and defining its implementation. Absolutely no modifications to existing code are required; it is not even necessary to recompile the existing code. Simply linking in the compiled code for the Xor gate is all that is needed. It is clear that using derived class and virtual functions in C++ is more general, maintainable, and flexible than using structures with type fields and switch statements in C.

3.4.3. Performance

Because of the extreme computational complexity of test generation, the efficiency of C++ was perhaps the single most important reason for its choice over other object-oriented languages to implement the hierarchical test generation application. During the process of design and implementation efficiency issues were kept in mind, but efforts were primarily focused on producing a clean and simple design. This strategy produced a system conducive to optimization via analysis and performance measurement.

A number of simple techniques can be applied during implementation to avoid significant amounts of overhead when using C++. For example, passing objects by reference rather than by value, avoiding the excessive use of virtual functions, avoiding superfluous creation and destruction of objects, and utilizing inline functions whenever possible are all important techniques for enhancing
the performance of a C++ application. Using inline functions is especially important in reducing run time, especially when encapsulation is properly used and classes possess many small member functions that exist only to provide controlled access to private member variables.

To compare the efficiency of the object-oriented hierarchical ATPG application with the procedural version, both applications were used to generate tests for the 1985 ISCAS combinational gate-level benchmark circuits [15]. Two versions of the object-oriented application were used, a version with no inline functions and a version that uses inline functions for most simple, single line member functions. Tests were generated for all single stuck-at gate-level faults, with a backtrack limit of 20. No attempt was made to identify equivalent faults or reduce the fault list using fault simulation or test compaction.

Table 1 shows the run-time results obtained for the largest benchmark circuits, shown in terms of the average CPU time in seconds required to generate each test. These figures were found using a Digital Equipment Corporation DECstation 3100 running Ultrix. For three of the circuits, c2670, c5315 and c7552, the inlined object-oriented version actually required less CPU time than the procedural version, which could be considered to be highly optimized. In the worst cases, c1355 and c6288, the object-oriented version required 4.6 and 3.2 times as much CPU time, respectively. By profiling the execution of these cases, the extra overhead was found to be caused by a combination of substantial backtracking during test generation, and inefficiencies in the Stack<Wire> used for backtracking by the ControlAlgorithm and the PriorityQueue<Instance> used during forward implication. Improving the implementations of these collection classes would significantly improve the run-time of these cases. It is instructive to note here that the object-oriented ATPG version compiled without inlining took an average of 78 percent more CPU time

Chapter 3. Object Oriented Test Generation
Table 1. Performance Comparison of Procedural and Object-Oriented Test Generation Applications

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Gates</th>
<th>Procedural (straight C) version</th>
<th>Object-oriented (C++) version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>With Inlining</td>
<td>No Inlining</td>
</tr>
<tr>
<td>c880</td>
<td>383</td>
<td>0.011</td>
<td>0.031</td>
</tr>
<tr>
<td>c1355</td>
<td>546</td>
<td>0.088</td>
<td>0.401</td>
</tr>
<tr>
<td>c1908</td>
<td>880</td>
<td>0.078</td>
<td>0.190</td>
</tr>
<tr>
<td>c2670</td>
<td>1193</td>
<td>0.103</td>
<td>0.099</td>
</tr>
<tr>
<td>c3540</td>
<td>1669</td>
<td>0.160</td>
<td>0.415</td>
</tr>
<tr>
<td>c5315</td>
<td>2307</td>
<td>0.153</td>
<td>0.131</td>
</tr>
<tr>
<td>c6288</td>
<td>2406</td>
<td>0.323</td>
<td>1.025</td>
</tr>
<tr>
<td>c7552</td>
<td>3512</td>
<td>0.514</td>
<td>0.412</td>
</tr>
</tbody>
</table>
than the inlined version.

These results are quite promising, especially since the object-oriented ATPG application was specifically developed to take advantage of hierarchical circuit descriptions, while the benchmark circuits used flat gate-level descriptions. Its full power and potential for providing a speedup in test generation is only realized when it is used with large, multi-level circuit descriptions.

3.5. Summary

This chapter described the use of an object-oriented approach for the analysis and design of a hierarchical test generation algorithm that supports the testing of circuits containing elements modeled with structural hierarchy and multiple levels of abstraction. Implemented in C++, the test generation system is easily maintainable, provides an extensible framework for test generation supporting easy incorporation of new types of circuit primitives and faults, and its efficiency is promising, retaining a similar level of computational performance to what can be achieved using C. The experience gained through development of the test generation implementation confirms claims that object-oriented programming and C++, when properly used, offer opportunities for developing better applications that are easier to write, easier to maintain, and easier to extend than equivalent applications implemented using a procedural design approach and language.
Chapter 4. $I_{DDQ}$ Test Generation for Bridging Faults

This chapter considers the accurate representation of bridging faults (BF's) in CMOS, and the use of $I_{DDQ}$ testing, an alternative technique for the detection of such faults made possible by the normally small supply currents in CMOS circuits [2]. The bridging fault test generation algorithm presented here addresses the primary issues involved in test generation, including fault sensitization, backwards implication of logic values to controllable inputs, and propagation of faults to observable outputs. The emphasis of this work is solely on the generation of tests to achieve very high fault coverage of unrestricted BF's using current observation. Logic level testing and faults that cannot be modeled as a BF are not considered.

The following section presents background and previous approaches related to this research. Section 4.2 describes the circuit models used in this research, and Section 4.3 discusses the representation of bridging faults. Sections 4.4 and 4.5 cover the algorithms used for gate-level and switch-level test generation, respectively. Section 4.6 describes the implementation of a full ATPG application for $I_{DDQ}$ testing of bridging faults, and presents results.

*Portions of this work also appear in [11].
4.1. Background

Much of the previous research on bridging faults has examined test generation at the gate level, with faults modeled as logical operations between node values [64, 37, 55, 1]. This approach to generating tests for BF’s assumes that a fixed Boolean relationship, such as wired And or wired Or, exists at the fault site. Most of this work relies on simplifying assumptions concerning the BF model used, such as restricting the location of BF’s or considering only certain simplified classes of circuits. Also, feedback bridging faults pose a particularly difficult problem since they may induce oscillations that are undetectable using voltage measurement or cause sequential behavior.

A recent approach also assumes a wired logic model for BF’s, and adapts that assumption to CMOS by simulating the effects of both wired Or and wired And logic for every BF [67]. Although it is true that bridging faults in CMOS circuits usually result in voltages far enough from threshold values to be interpreted as a valid logic value, it is shown in [93] that CMOS gate output voltage in the presence of a BF is dependent on the voltage divider network formed by the shorted P and N networks. As the topology of this network varies with different input patterns, the Boolean output function of the faulted gate output may not be a simple And or Or.

A number of approaches for generating tests for MOS circuits have relied on the graph-theoretic model to represent circuit structure and faults [33, 27]. In these models the circuit nodes and edges of a MOS gate are represented by the vertices and edges of a graph. Graph algorithms to determine paths and cutsets are then used to determine tests. These methods have been estimated
to model little over half of the faults commonly occurring in MOS circuits [35]. This is due to
two factors: (1) these methods are unable to model faults that bridge input nodes, and (2) pass
transistor networks are either not modeled at all, or can only be modeled as unidirectional
elements. The circuit model and algorithms formulated in this research are not subject to either
limitation.

A second problem with previous approaches to modeling CMOS circuits is the common practice
of representing large circuits with a single-level monolithic netlist, requiring enormous storage
capacity and exponential processing time. Approaches to switch-level test generation using a
strictly graph-based technique have failed to adequately address the problem of handling this
complexity. Some approaches rely on the generation of a single graph to describe an entire
circuit, leading to extremely complex and unwieldy graph representations. To maintain reasonable
memory requirements and computational complexity for VLSI circuit processing, this work
incorporates a hierarchical circuit model.

A number of researchers have reported the characteristics of bridging faults and the use of $I_{ddq}$
observation to detect both BF’s and parametric failures [2, 23, 45, 61, 65, 80]. Also, various
types of test sets, for example stuck-at and pseudo-random, have been investigated for their
voltage-observation and $I_{ddq}$ coverage of BF’s [67, 93, 36]. However, little work has been
reported on algorithms that automatically generate $I_{ddq}$ tests for bridging faults directly from a BF
model. A single stuck-at fault (SSF) test generation algorithm was modified in [36] to generate
tests for transistor stuck-on faults. Results showed that test generation for $I_{ddq}$ faults produces
smaller test sets and is less computationally complex than for equivalent SSF faults. Although

Chapter 4. $I_{ddq}$ Test Generation for Bridging Faults
the algorithm is able to handle large circuits, it cannot generate tests for BF's other than transistor stuck-ons (TS-ON's).

Evidence that test generation for stuck-at faults is more efficient when algorithms are based on $I_{ddq}$ observation rather than logic testing is presented in [38]. A conventional sequential circuit testing algorithm is modified to generate test vectors suitable for $I_{ddq}$ testing of gate-level stuck-at faults. The $I_{ddq}$ algorithm was superior in terms of reduced CPU time and test set size, increased stuck-at fault coverage, and the ability to detect non-stuck-at faults. This approach is applicable to large circuits, but is unable to generate explicit tests for BF's other than bridges between gate-level nodes and $V_{dd}$ or $V_{ss}$.

In [63], a leakage fault analysis method is used to select a subset of test vectors for $I_{ddq}$ testing from a supplied functional test set. The algorithm is able to select a small percentage of vectors that provide the same leakage fault coverage as the complete set, reducing the time needed for $I_{ddq}$ test generation. The approach differs from the work presented here in its use of exhaustive testing of circuit components to build static leakage fault detection tables. The only BF's considered are leakage faults affecting single transistors, and the method is not able to increase the BF coverage over that achieved by the original functional test set.

The approach most similar to this research is presented in [93, 73]. In this work, $I_{ddq}$ test vectors for BF's are generated using the symbolic simulator COSMOS [28] to derive Boolean expressions for internal circuit nodes, and manipulating the expressions so as to force one of the shorted nodes to $V_{dd}$ and the other node to $V_{ss}$. Due to the complexity of the switch-level model used in COSMOS, the storage requirements for the algorithm limit the size of circuits that can be
considered. For the largest ISCAS benchmark circuits [15] used to report results, c499 and c880, the circuits could not be processed in their entirety, necessitating their partitioning into sections.

4.2. Circuit Models

Accurate test generation for CMOS circuits requires the use of techniques that are different from those commonly employed in gate-level test generation since the effects of physical failure mechanisms and certain structures, such as pass transistors, commonly found in CMOS circuits do not possess a corresponding Boolean gate representation. As a result, CMOS circuits are more appropriately modeled as a set of interconnected transistors. Such a model is commonly called a switch-level representation, since transistors are treated as switches that are either on or off, depending on the state of their controlling gates. Since the switch-level model captures only the digital behavior of a circuit, any voltage drop across an on transistor is ignored. In general, the exact logical values for each node in a switch-level circuit are determined by both the logical levels of connected nodes and the relative strengths of logic values.

4.2.1. Switch-Level Circuit Representation

A variety of techniques have been presented in the literature for representing a circuit at the switch-level. Matrix methods, tree structures, and graph-based techniques have been employed with varying degrees of success. The circuit model used here is a subset of the switch-level model for MOS circuits developed by Bryant [16]. Transistors and nodes are modeled as having uniform strength, and the control values for edges are explicitly enumerated. These simplifications
serve to make the model more suitable for test generation purposes by eliminating the need to store unneeded information.

Of the various methods used to represent a switch-level circuit, the graph-based approach used in this research has the advantage of offering an accurate description that corresponds directly to the circuit topology, and the ability to easily represent structural faults such as general short and open, transistor stuck-on and stuck-off, and line stuck-at by modeling them as graph perturbations. A circuit model is defined by the undirected multigraph \( G = (V, E, C, L) \), where \( V \) is the set of graph vertices, and every \( v \in V \) represents a non-separable electrical node. The set of graph edges \( E \) represents transistors, wire connections between separable nodes, or potential bridging faults. Edges are unordered pairs of distinct vertices, denoted by \((v, w)\). For every edge \( e \in E \), there is an associated Boolean control variable \( c \in C \) that determines whether or not the edge is conducting. N-channel transistors are controlled directly by the gate variable, and p-channel transistors are assigned the complement of their gate variables. Edges representing wires are assigned a \( I \) since they always conduct. The set of labels \( L \) identifies the characteristics of every node in \( V \) and every edge in \( E \).

Every node \( v \) in the graph representation \( G \) of a CMOS circuit element has an associated label \( l_i \in \{input, output, source, sink, internal\} \). The labels identify the characteristics of a node, such as whether the node's logic value can be directly influenced by nodes not contained in \( G \) under fault-free conditions. The presence of N- and P-transistors implicitly declares the presence of the set of distinguished nodes of type \( l_i = input \). Input nodes are used to store the logic values of the controlling gate variables associated with transistor edges. Input nodes are not connected to other
nodes in $G$ by edges, but have an implied relationship with one or more associated transistor edges.

Figure 9 shows an elementary CMOS circuit and one possible graph representation. Each transistor and connection is modeled by a labeled edge in the graph. The edge labeled by a 0 is included to model a potential short fault between $V_{SS}$ and node $n_r$.

Advantages of the graph model defined above include its adaptability and the potential sharing of the data structures by a companion CMOS fault simulator. Such a simulator is used to validate the test sets generated by the switch-level test generation algorithms. The simulator also plays an optional role in the generation of individual tests, as part of a generate/validate procedure that immediately detects invalid tests as soon as they are generated. The graph-theoretic approach is used here to support the modeling of both fully complementary (FCMOS) and non-classical complex gates. Only static CMOS gates, i.e. gates that do not employ dynamic clocking techniques, will be considered. The bidirectional charge transfer capability of MOS transistors is modeled correctly as well, supporting the representation of transmission gates and arbitrary pass transistor logic.

### 4.2.2. Hierarchical Circuit Representation

Partitioning a circuit into a set of interconnected modules reduces the computational complexity of finding paths and supports a hierarchical circuit model. At the lowest level of the hierarchy, individual circuit elements such as transistors and wire connections define the interconnection of nodes and edges. The next level up in the hierarchy is occupied by modules, which are represented internally by switch-level graphs composed of arbitrarily connected nodes and edges.
Figure 9. Example switch-level module representation.
Each module \( M \) must include one or more source nodes, a single sink node, and, optionally, a set of one or more inputs and a single output. These four types of nodes are referred to as external nodes. External nodes provide an interface between interconnected modules. The logic values of externally driven input nodes are forcing, and cannot be altered except in the presence of fault conditions. Source and sink nodes may be treated as input nodes or fixed at \( V_{DD} \) and \( V_{SS} \), respectively. In elementary and complex CMOS gates, logic values flow from the source node \( (V_{DD}) \) to the output node or from the sink node \( (V_{SS}) \) to the output node. Alternatively, the source and sink may be influenced by other nodes, allowing the flow of information from source to sink, source to output, or output to source, for example. Also, a module may contain any number of internal nodes, and these internal nodes may control transistor edges. The characteristics of modules support direct modeling of simple and complex CMOS gates, pass transistor networks, and more complex structures such as Xor's. Complicated circuit blocks may be decomposed into modules by partitioning the circuit appropriately.

At the highest level in the hierarchy, a network module defines the primary inputs and observable outputs of the entire circuit. The network module is defined by connections of internal circuit primitives and modules. Two modules \( M_i \) and \( M_j \) may only interact via their external nodes, i.e. the output nodes of \( M_i \) may influence the input nodes of \( M_j \), or the sink node of \( M_i \) may influence the source node of \( M_j \). References to modules serve to define the connections between adjacent subcircuits, and are known as module instances. During test generation or simulation, a module instance does not allocate storage for any nodes or edges generated by circuit primitives in the instance. Each instantiation requires only enough storage for the instance connectivity and the set of external nodes. This approach conserves memory resources by maintaining a single switch-
level structural description for every distinct module in the circuit, no matter how many times the module is replicated.

The hierarchical approach described above is extended in this work to represent an additional layer of abstraction. The lack of logical information in a switch-level graph is relieved by attaching a gate- or functional-level description to appropriate modules. These high-level descriptions are a prerequisite for efficient implementation of the backward line justification and forward implication routines that are used in the test generation routines presented here. Figure 10 shows an example circuit, C17 from the 1985 ISCAS benchmark circuits [15], modeled in hierarchical form. This version of C17 was produced by the Microelectronics Center of North Carolina's (MCNC) design automation system [56], and differs from the standard gate-level version.

The use of a circuit model incorporating a hierarchy of modules or sub-graphs solves a number of problems that have limited the practicality of previous switch-level approaches. The use of hierarchical definitions provides for structured circuit partitioning that matches design styles such as the use of standard cell libraries. Processing circuits hierarchically eliminates the need for non-linear circuit expansion and reduces both circuit processing time and overall analysis complexity. Complex graph algorithms for path tracing and connectivity are limited to small sub-graphs, and simpler algorithms are used for higher level tasks such as backwards implication.

4.2.3. Switch-Level Module Taxonomy

The switch-level modules used in this research are categorized based on characteristics that affect their processing during test generation, such as internal switch-level configuration and ease of high-level implication and backtracking. Figure 11 shows a partial taxonomy of the switch-level
Figure 10. Example hierarchical circuit representation.
Figure 11. Switch-level module class hierarchy.
standard cell modules used to obtain the results presented in this chapter. These standard cell models are classified into three groups, and are referred to as primitive, complex, and composite modules.

**Primitive Modules.** In primitive modules each transistor edge is controlled directly by an input node, and all edges are contained in a single transistor group. Primitive modules implement the inverting class of basic logic functions such as Nand, Nor, and Invert, and are traditionally directly supported by classical gate-level test generation algorithms. High-level implication can be handled by a single table lookup, and backtracing is fairly simple to perform.

**Complex Modules.** Complex modules are similar to primitive modules in that transistor edges are controlled only by input nodes, but the edges are connected in more complicated series-parallel arrangements to provide complex logic functions such as And-Or-Invert and Or-And-Invert. High-level implication can be performed using larger lookup tables than are needed with primitive modules, or by using a sequence of primitive table lookups. Backtracing is more complicated than with primitive modules, as the influence of input nodes on the module’s output is no longer symmetrical.

**Composite Modules.** Composite modules are distinguished by the fact that transistor edges may be controlled by internal nodes in addition to input nodes. The presence of such input nodes, referred to as gate nodes, can significantly complicate switch-level implication and backtracing. In a standard cell environment, composite modules can often be classified into one of two subgroups, inverted output or dual group modules. Inverted output modules are essentially primitive modules with an inverter added to the output to provide basic logic functions such as...
And and Or. High-level implication and backtracing is straightforward as with primitive modules.

Dual group modules are formed when the output of a primitive or complex module is tied to the
input of another primitive or complex module to form a more complicated logic function. For
example, standard cell versions of Xor and Xnor functions are commonly implemented in this
fashion. For efficiency, both forms of composite modules are treated as indivisible gates during
high-level processing, but require the use of more complex switch-level algorithms to handle the
presence of more than one transistor group.

Switch-level modules not fitting into the module taxonomy shown in Figure 11 can be partitioned
into a set of single group or composite modules. This collection of simpler modules is then
treated as a netlist module using the hierarchical approach described above.

4.3. Bridging Fault Model

Bridging faults, or shorts between normally unconnected signal lines, have been shown to be a
common failure in CMOS circuits. Studies examining realistic faults in digital CMOS circuits
suggest that bridging faults account for from thirty to fifty percent of all faults [35]. This section
discusses the representation of bridging faults, assumptions concerning their sensitization and
detectability in an I_{DDQ} test environment, and a classification of BF's based on test generation
considerations.
4.3.1. Bridging Fault Representation

In a switch level graph, a BF is represented by a unique edge type and associated control variable to distinguish it from transistors and conducting lines. A BF edge is considered non-conducting in a good circuit and always conducting in the faulty circuit. The edge’s controlling variable assumes a value of 0 in the fault-free circuit and 1 in the faulty circuit. A BF edge can model the effects of arbitrary, unrestricted bridging faults. For an intra-gate BF, where the fault connects the nodes of a single CMOS gate, the BF edge affects the switch-level graph of a single module, \( M_i \). For an inter-gate BF, a fault involving the nodes of two distinct CMOS gates, the BF edge connects nodes in two modules, \( M_i \) and \( M_j \).

4.3.2. Sensitization Conditions

To detect a short fault in a CMOS circuit, a single node must be connected to both power and ground to establish an abnormally high steady state supply current. In graph terminology, a path \( p \) of conducting edges that traverses the short edge \((v, w)\) must be established between \( V_{DD} \) and \( V_{SS} \). To test for a bridging fault between two graph nodes \( v \) and \( w \), the circuit is partitioned into sets of connected components such that one set contains \( V_{DD} \) and some other set contains \( V_{SS} \). The bridge is then tested if \( v \) is contained in the \( V_{DD} \) set and \( w \) is contained in the \( V_{SS} \) set, or if \( v \) is contained in the \( V_{SS} \) set and \( w \) is contained in the \( V_{DD} \) set.

Let \( m(i) \) be the \( ith \) possible input minterm for a graph \( G \), and \( I \) be the set of all primary input combinations for \( G \). Also let \( \Pi(x, y) \) be a function that accepts two nodes \( x \) and \( y \) as arguments and returns the set of minterms that produce a connected component containing both \( x \) and \( y \). The following steps generate a test \( T(v, w) \) for a BF between \( v \) and \( w \).

Chapter 4. \textit{I}_{BBQ} Test Generation for Bridging Faults

92
Find \( \Pi(V_{dd}, v), \Pi(V_{ss}, w) \) and \( \Pi(V_{dd}, w), \Pi(V_{ss}, v) \), where
\[
\Pi(x, y) = \{ m(i) | m(i) \in I, m(i) \Rightarrow x, y \text{ connected} \}
\]
Then \( T(v, w) \in \left\{ \left[ \Pi(V_{dd}, v) \cap \Pi(V_{ss}, w) \right] \cup \left[ \Pi(V_{ss}, v) \cap \Pi(V_{dd}, w) \right] \right\} \).

In \( I_{DDQ} \) testing, \( V_{dd} \) and \( V_{ss} \) provide observable access points for parametric failures occurring in the circuit. In effect, faulty supply currents are propagated vertically from the power to the ground connection. Tests in which the BF affects the value on the gate of one or more ON transistors in the path between \( V_{dd} \) and \( V_{ss} \) are considered to be valid. This is because such indeterminate values on transistor gates will generally induce a high-current condition in the faulty circuit [35, 93]. There is no need to check the exact value of indeterminate node voltages or propagate faulty logic values to an observable output as is necessary with standard testing algorithms based on logic observation.

Note that in the presence of inter-gate BF’s feedback may be introduced into the faulty circuit. Using \( I_{DDQ} \) tests, BF’s that introduce oscillation are considered to be detectable since the failure of the circuit to enter a static, quiescent state will be detected by a continuous high switching current [35].

4.3.3. Bridging Fault Taxonomy

The bridging faults considered in this research are categorized based on features that impact their testability, such as whether they involve an internal node of a module. Figure 12 shows the taxonomy used to classify unrestricted BF’s in a netlist of primitive, complex, and composite modules. The most general property used to classify a BF in such a netlist is whether a test can
be generated for it using only gate-level routines, or if switch-level test generation is required.

**Gate-Level Bridging Faults.** Short faults belonging to the gate-level subtree in Figure 12 exist only between the wires forming connections between modules. BF's in this category may bridge a wire to power or ground, or may bridge two distinct wires. A bridge between power and ground is considered to be trivially detected. Test generation for these faults can be performed using only the high-level implication and justification routines associated with module instances.

**Switch-Level Bridging Faults.** Short faults in the switch-level subtree in Figure 12 involve at least one interior node of a module, and therefore require the use of low-level implication and justification routines during test generation. Faults in this category include transistor stuck-ons, bridges between an internal node and power or ground, bridges between a wire and an internal node, or bridges between two distinct internal nodes. Test generation for these BF's is discussed in more detail in Section 4.5.

The next section describes the algorithm used to generate $I_{DQ}$ tests for the gate-level classes of bridging faults shown in Figure 12.

**4.4. Gate-Level Test Generation**

In this research a standard method for performing test generation for combinational circuits, the PODEM (Path Oriented Decision Making) algorithm [43], is modified for use in $I_{DQ}$ test generation. Although PODEM was originally developed to generate tests for single stuck-at faults
in gate-level circuit descriptions, it has been successfully applied at a variety of levels of circuit abstraction [21, 25]. The PODEM algorithm is viewed here as a general test generation approach rather than a solution for specific types of faults and circuit descriptions.

Figure 13 shows the simplified pseudo-code for a gate-level PODEM-style generate_test routine. Of the operations shown, the backtrack, backtrack, and imply_changes routines can be used essentially without modification for \( I_{\text{DDQ}} \) test generation. With appropriate changes in the concept of an objective, and in the choose_objective and check_for_test routines, the control flow shown in Figure 13 can be used as the basis for an \( I_{\text{DDQ}} \) test algorithm. The gate-level \( I_{\text{DDQ}} \) test generation algorithm developed in this work is called BODEM (Bridge Oriented Decision Making), emphasizing its close ties to PODEM.

The BODEM \( I_{\text{DDQ}} \) test algorithm does not require the concept of an error value such as the \( D \) and \( \overline{D} \) logic values used in traditional single stuck-at fault test generation. Fault sensitization is accomplished by appropriately driving 0 and 1 logic values to bridged wires, and the use of power and ground as observable access points eliminates the need for fault propagation. Consequently, a three-valued set of logic values, \( LV = \{ 0, 1, X \} \), is adequate for test generation. An objective in BODEM is then uniformly defined as a wire objective pair, \( \text{WireObj} = (\text{Wire}, 0) \) or \( \text{WireObj} = (\text{Wire}, 1) \). Every \( \text{WireObj} \) associates a single wire with a specified logic value, and represents a target condition that BODEM will attempt to satisfy by making appropriate assignments to the primary inputs of a circuit.

Rather than requiring that an explicit gate-level bridging fault be specified as a test generation target, BODEM instead accepts a list of wire objectives. During test generation, the BODEM
procedure generate_test(fault)

    while backtrack limit is not exceeded
        objective = choose_objective();
        if a valid objective is found
            input_objective = backtrace(objective);
        else
            if decision tree is empty
                return Fault_not_testable;
            endif
            input_objective = backtrack();
        endif
        imply_changes(input_objective);
        if check_for_test() == TRUE
            return Found_a_test;
        endif
    endwhile
    return Test_was_aborted;

end

Figure 13. Control flow for PODEM algorithm.
version of the choose_objective procedure in Figure 13 scans this objective list, and returns the first unsatisfied objective in the list. In the event of a conflict due to a wire in the list being driven to the wrong objective value, choose_objective returns a NULL or invalid wire objective. The BODEM version of the check_for_test simply scans the wire objective list and returns TRUE if all objectives in the list have been met. The wire objective list employed in BODEM provides a uniform approach for specifying the sensitization requirements of different gate-level bridging fault types. As discussed in the next section, it also provides a general mechanism allowing a switch-level \( I_{DQ} \) test algorithm to cooperate with BODEM to generate cooperative tests.

In order for the above approach to function properly, bridging faults that will be tested by BODEM must be converted into appropriate wire objective lists. For a wire bridged to power or ground, the objective list would consist of a single wire objective with a logic value of 0 or 1, respectively. For a bridging fault between wires \( w_1 \) and \( w_2 \), the objective list might contain \( \{ (w_1, 0), (w_2, 1) \} \), or \( \{ (w_1, 1), (w_2, 0) \} \).

4.5. Switch-Level Test Generation

To handle bridging faults requiring treatment at the switch-level, the control algorithm shown in Figure 13 is extended in this work to implement a switch-level \( I_{DQ} \) test generation algorithm referred to as NODEM (Node Oriented Decision Making). The implementation of basic test generation operations in NODEM is significantly more complicated than the corresponding operations at the gate level, involving the manipulation of low-level node and edge module primitives. This section describes some of the methods used to facilitate the generation of \( I_{DQ} \).
tests for switch-level bridging faults, discusses the incorporation of these techniques into the NODEM framework, and provides an example of switch-level test generation. The following discussion focuses on switch-level intra-gate BF's.

4.5.1. Switch-level Path Tracing

Although the $V_{DD}$ and $V_{SS}$ connected components can be determined by an exhaustive path search to generate optimal test sets for BF's, the complexity of such an approach rules out efficient processing of large graphs. Instead, implicit enumeration based on back-tracking is used to explore the search space. The algorithm is briefly described below for the case of two internal nodes $v$ and $w$ in a CMOS logic gate.

Rather than attempting to sensitize a single path from $V_{DD}$ to $V_{SS}$ that traverses the fault site nodes $v$ and $w$, two separate paths $P_1$ and $P_2$ are sensitized starting from the BF location. First, if the BF is a TS-ON fault, the input node of the affected transistor is set so the transistor would be off in the good circuit. Starting from either node $v$ or $w$, the closest source or sink node is chosen as a target, and the path $P_I$ is enumerated using a depth-first search algorithm (DFS). Here $v$ is the initial node, and the gate's source node is the initial target. As each transistor edge is encountered in DFS, its controlling gate node is set based on the transistor type, and the edge is added to the current path. As the values of gate nodes are set, the states of dependent transistor edges are updated to maintain consistency. Transistors that are encountered with preset states cannot be traversed unless the edge is turned on. As DFS progresses either the target node of the path will be reached or the path cannot be established due an unmet satisfiability constraint. If the latter occurs, the algorithm backtracks by removing edges from the path and releasing input
node constraints until a new edge can be tried. This procedure is repeated until a path reaching
the target node is established, or no path is established.

Assuming that path $P_1$ is successfully established between $v$ and the source node, the graph input
minterm generated by $P_1$ is checked for path sensitization between node $w$ and the gate's sink
node. If no such path $P_2$ exists, DFS can again be used to strategically set unspecified gate inputs
and complete path $P_2$ if possible. If $P_2$ cannot be formed with the set of input constraints
established by $P_1$, a new $P_1$ may be generated by backtracking and resuming the original DFS
procedure. New paths may be generated for $P_1$ and $P_2$ until a set of two valid paths are found.
If the BF is not testable using the original external node targets, all possible $P_1'$s and $P_2'$s will be
tried without finding a test. At this point, an alternate target, for example the gate's sink node,
can be chosen for the initial target for path $P_1$ emanating from initial node $v$, and the entire
procedure repeated. If no input combination exists that satisfies the BF sensitization requirement,
the circuit is redundant, and no test can be found to detect the BF.

For intra-gate BF's that involve the external nodes of a module, the type of nodes $v$ and $w$ can
be any value in the set \{input, source, sink, output\}. Such nodes are considered to be target nodes
in DFS if their value can be controlled by other modules or are fixed at $V_{DD}$ or $V_{SS}$. Nodes with
$l_i \in \{\text{input, source, sink}\}$ are always considered to be targets. Nodes with $l_i \in \{\text{output}\}$ are target
nodes only if another module can influence the node's value. This is not unusual in CMOS
circuits, where multiple circuit elements and/or module outputs can feed into a single node in the
circuit.
4.5.2. Topological Ordering

Although the algorithms presented in this section are capable of generating tests for bridging faults given a circuit description in which CMOS circuit blocks are described by arbitrary undirected graphs, the algorithms can be made to run more efficiently by taking note of (1) the direction of signal flow in transistors through which charge transfer takes place in only one direction, and (2) the distance of nodes from a common reference point such as the module’s sink node. This determination can be made once for each module in the good circuit, providing information to support efficient path tracing.

By assigning signal flow directions to CMOS transistors, the undirected multigraph representing a switch-level module is converted into a directed multigraph. Hints as to which transistors operate in a true bidirectional manner are taken advantage of, but in their absence the DFS operation discussed above will still function correctly, though not as efficiently since more backtracking may occur. In addition to direction assignment, the nodes of switch-level graphs are ordered based on their distance from the graph’s source and sink nodes. The current test generation implementation parses switch-level module descriptions in which signal flow direction and distances are precalculated, but algorithms do exist to automatically perform these steps [54, 59].

4.5.3. Backwards Implication

After a test is generated for a switch-level BF in a faulty module, the necessary input conditions must be propagated back to the circuit’s primary inputs. During this backwards implication step, gate-level descriptions stored with modules are used with higher level routines to speed the process. Modules that have no corresponding Boolean gate representation, or for which none is
supplied, are handled by a switch-level routine. Given that the value of a graph $G$'s output node must be $I (0)$, this routine establishes a path between the output node and $V_{do}$ ($V_{ss}$) by properly setting the values of input nodes for edges not yet assigned a value during implication. Since several possibilities may exist in the presence of unspecified inputs, an algorithm similar to that explained above must be used to allow an enumerative search of the solution space for implication.

As module input nodes are assigned values during the backwards implication stage, a forward propagation check can be performed on input signals fed from module outputs that fanout to ensure that logic values assigned to nodes during backwards implication do not conflict with previous node assignments, or with the input values assigned to the switch-level module(s) under test. Logic implication is performed at the gate-level when possible for speed, and otherwise at the switch-level.

### 4.5.4. Switch-Level Control Flow

The test generation control flow shown in Figure 13 is used once again to control the switch-level NODEM test routine. As with BODEM, appropriate changes are required in the objective concept and in the choose_objective and check_for_test routines. In addition, the backtrace and imply_changes routines are replaced with the switch-level operations already described in this section. NODEM also employs a specialized backtracking mechanism allowing it to be used as a test iterator.

NODEM operates with the same three-valued set of logic values used in BODEM, $LV = \{ 0, I, X \}$. An objective in NODEM is uniformly defined as a $(Node, LV)$ pair in fault sensitization,
forward implication, and backward justification. NODEM also accepts a list of node objectives rather than an explicit switch-level bridging fault. During test generation, the NODEM version of the choose_objective procedure in Figure 13 scans this objective list, and returns the first unsatisfied node objective in the list. Choose_objective returns a NULL or invalid node objective in the event of a conflict due to a node in the list being driven to the wrong objective value. The NODEM version of the check_for_test scans the node objective list and returns TRUE if all objectives in the list have been met. Switch-level BF’s that will be tested by NODEM must be converted to node objective lists in the same manner that gate-level BF’s are converted to wire objective lists for BODEM.

NODEM is designed to operate on a single switch-level module instance, generating a local test that satisfies the conditions specified by a supplied node objective list. Once a valid test is generated, NODEM maps any input nodes specified by the test into a set of wire objectives, and then passes the wire objective list to BODEM for high-level justification back to the circuit’s primary inputs. If BODEM succeeds in satisfying all objectives in the wire list passed to it by NODEM, a valid test is generated.

The need for NODEM to function as a test iterator is due to the fact that usually more than one set of input node values exist that satisfy the switch-level node objective list associated with a specific BF. Some combinations of input node values may be easy for BODEM to justify back to primary inputs, while others may represent a set of unsatisfiable constraints. For this reason, NODEM employs a special backtracking procedure that can be restarted even after a valid local test has already been found, and can return a new set of input values each time a test is requested, until the search space of the faulty module is exhausted. NODEM also attempts to return
minimally specified tests first, to reduce the number of wire objectives passed to BODEM and increase the chances of finding a valid test quickly.

Test generation for a switch-level bridging fault in the class [internal node to internal node, different gates] actually requires that a pair of NODEM test generators be used, one for each module affected by the fault. Test generation in this case can be very complicated, depending on whether one module influences inputs of the other. An example of test generation in this situation is discussed next.

4.5.5. Test Generation Example

To illustrate the behavior of the switch-level algorithm, this section presents an informal description of how the NODEM/BODEM combination would generate a test for a BF in the circuit shown in Figure 10. The target fault is the inter-gate BF between the internal nodes of the n-networks of the gates labeled A and B in Figure 10. Although the likelihood of this fault occurring is lower than that of other BF's in the circuit, it serves as a good non-trivial example. In the following discussion n- and p-transistors are identified by the notation \( N_x(I) \) or \( P_x(I) \) respectively, where \( X \) identifies the gate under consideration, A or B, and \( I \) identifies the transistor's gate node.

Figure 14 shows the fault location with gates A and B expanded to their switch-level equivalents. The inverter between A and B is not expanded, since it can be handled at the gate level for this BF. Note that the states of transistors in gates A and B are not independent, since A and B share one input signal, and one input of B is influenced by the output of A. The bridged internal nodes are labeled \( n_a \) and \( n_b \). To sensitize the BF, nodes \( \{ n_a, n_b \} \) must be driven to either \( \{ 0, 1 \} \) or
Figure 14. Fault sensitization with unsatisfiable constraints.
\{I, 0\} logic states. Here we assume that the algorithm chooses to sensitize the \{0, I\} condition first, and that driving node \(n_{A}\) to 0 is chosen as the initial objective.

Examining the connectivity of node \(n_{A}\) reveals that transistor \(N_{A}(I_{2\text{out}})\) must be turned ON, requiring that node \(I_{2\text{out}}\) be set to logic 1. No other constraints must be satisfied in this case, and the other input of gate \(A\) is left unspecified at this point. In gate \(B\), sensitizing node \(n_{B}\) to 1 with the depth-first search algorithm requires that transistors \(N_{B}(I_{10}), P_{B}(I_{13}),\) and \(P_{B}(I_{7\text{out}})\) be turned on, constraining nodes \(I_{10}, I_{13}, I_{7\text{out}}\) to the values \(\{I, 0, 0\}\). This can be determined without backtracking, and no other assignments are possible.

At this point, a gate-level implication of the assigned node values immediately reveals that the assignment \(I_{10} = 1\) for gate \(B\) sets the output of gate \(A, I_{12}\), to 0. The output of the inverter is driven to 1, generating a conflict on node \(I_{13}\) with the previously assigned value of 0. This forces the algorithm to backtrack to the switch-level, where it discovers that no other possibilities exist for sensitizing the fault nodes to the initial target values. The target node values for \(n_{A}\) and \(n_{B}\) are now set to \(\{I, 0\}\), and driving node \(n_{A}\) to 1 is chosen as the initial objective.

In this case the DFS algorithm determines that the only way to sensitize a path from \(n_{A}\) to \(V_{DD}\) is setting transistors \(N_{A}(I_{10})\) and \(P_{A}(I_{2\text{out}})\) to the ON state. This requires that nodes \(I_{10}\) and \(I_{2\text{out}}\) be set to logic 1 and 0, respectively, and also sets the output \(I_{12}\) of gate \(A\) to 1. In gate \(B\), the first path found sensitizing node \(n_{B}\) to 0 requires only that transistor \(N_{B}(I_{7\text{out}})\) be turned on, constraining node \(I_{7\text{out}}\) to the value 1. In this case an alternative path does exist in the event that backtracking returns to this point.
Figure 15. Successful sensitization conditions for detection of bridging fault.
Gate-level implication of the assigned node values determines that the I on node \( I_{12} \) drives the output of the inverter \( I_{13} \) to 0. A switch-level simulation of gate B shows that \( I_{13} = 0 \) sets transistor \( P_{B}(I_{13}) \) ON and \( N_{B}(I_{13}) \) OFF. The previously specified node values \( I_{10} = 1 \) and \( I_{7_{q_{a}}} = 1 \) turn transistors \( N_{B}(I_{10}) \) and \( N_{B}(I_{7_{q_{a}}}) \) ON, and \( P_{B}(I_{10}) \) and \( P_{B}(I_{7_{q_{a}}}) \) OFF, isolating the partition that node \( n_{a} \) is in from \( V_{DD} \) and avoiding any conflicts. With the faulty nodes of gates A and B properly sensitized, backwards implication at the gate-level is used to assign primary inputs so that a value of \( I \) is set on node \( I_{10} \). Note that it is not necessary to propagate the outputs of gates A and B any further. A successful implication assigns 0 to one of the inputs of the Nand gate driving \( I_{10} \), and a valid test \{ \( I_{1_{q_{a}}}, I_{2_{q_{a}}}, I_{3_{q_{a}}}, I_{6_{q_{a}}}, I_{7_{q_{a}}} \) \} = \{ X, 0, X, 0, 1 \} is generated, as shown in Figure 15.

4.5.6. Bridging Fault Simulator

To verify tests generated for short faults, a current-mode bridging fault simulator, based on the same data structures developed for the test generation algorithms, is used. The simulator is similar in some aspects to Bryant's MOSSIM-II [16], but differs in its use of modules, hierarchy, and a simplified switch-level model. The simplifications are acceptable within the framework of test generation for static CMOS combinational circuits. The simulator forms an essential part of the test generation package. During test generation, modified simulator routines are used to perform forward implication at the switch-level, and embedded fault simulation routines in the test generation algorithm allow for immediate validity checking of individual test vectors.

In addition to the switch-level simulator used by NODEM, a full gate-level, current mode fault simulation algorithm is built into BODEM. This allows \( I_{DDQ} \) test generation and fault simulation to be interleaved by following each test generation step with a fault simulation step to identify

Chapter 4. \( I_{DDQ} \) Test Generation for Bridging Faults
additional faults that are detected by a test. Since the complexity of bridging fault simulation is much lower than that of test generation, the fault simulation phase can identify bridging faults detected by the same test vector and avoid generating tests for those faults, saving a significant amount of effort in test generation. This also provides dynamic test set compaction since tests are not generated for the additional faults detected during the fault simulation step. The bridging fault simulator in BODEM uses a demand-driven technique for determining when modules must be simulated at the switch-level. The logic values on the internal nodes of a module are updated if and only if in the course of fault simulation those values are required to determine if a BF is sensitized and detected. As the uncovered fault list shrinks during test generation, fewer and fewer modules have to be simulated at the switch-level, and can instead be handled using efficient high-level implication routines.

4.6. Implementation and Results

This section describes the complete suite of tools used to generate IDQ tests for realistic bridging faults in CMOS circuits and discusses implementation details and results for the two tools developed in this research, CARP and BODEM. The second program actually contains cooperating implementations of both the gate-level BODEM and switch-level NODEM algorithms, but is referred to here as BODEM for simplicity. The relationship between these tools is pictured in Figure 16. Executable programs are distinguished by rectangular boxes, and intermediate data files are shown inside ellipses.
Figure 16. iDOQ test generation tool suite.
The results presented in this section were obtained using CMOS layouts of the 1985 ISCAS benchmark circuits. These layouts were produced by the MCNC design automation system, and are Boolean equivalents of the original gate-level versions [56]. The layouts contain a variety of standard cell types, including most of the primitive, complex, and composite modules shown in Figure 11. During test generation, these standard cell modules are treated as indivisible logic gates rather than being expanded into their primitive logic gate equivalents.

4.6.1. Circuit and Fault Extraction

For sizeable CMOS circuits, explicit testing for all possible BF’s is clearly intractable since there are a total of $|V|(|V| - 1)$ possible BF's. In practice, however, only nodes in close physical proximity are likely to be affected by a BF, reducing the size of the "neighborhood" that must be considered for each node in the circuit [93]. In this work, lists of realistic bridging faults are extracted using the inductive fault analysis tool CARAFE [49].

The process of generating $I_{DDQ}$ tests for a circuit begins with the circuit layout described via MAGIC [75] in a .mag file. CARAFE uses Monte Carlo simulation of physical spot defects to generate defect data and store results in a .pro file containing a list of bridged circuit node pairs ranked according to their probability of occurrence. Because CARAFE flattens hierarchical circuit designs into a flat transistor netlist, the resulting extracted netlist of the circuit is not useful for hierarchical test generation. This is worked around by using the hierarchical circuit description contained in the .ext files generated by MAGIC’s :extract command.

The list of BF’s generated by CARAFE contains both faults occurring only within the interconnect regions of a circuit design and faults affecting the interior of logic gates. Because circuit layouts
commonly contain feed-through metal lines that remain unconnected after routing, CARAFE generates some "untestable" BF’s that involve isolated circuit nodes. Also, duplicate bridging faults can be generated, as long interconnect runs are bridged to power or ground more than once. These anomalies are rectified by the tool described in the following section.

4.6.2. Bridging Fault Classifier

Given a list of bridged node pairs from CARAFE and a detailed hierarchical circuit description from MAGIC, the next step in test generation is to perform bridging fault classification and switch-level module extraction. This is done by CARP, the CARAFE output processor. CARP filters out untestable BF’s due to unused feed-through nodes, combines duplicate bridging faults and adjusts their occurrence counts, and then re-ranks the entire fault list. As this is done, faults are classified according to the BF taxonomy in Figure 12, by determining the relationship of nodes as indicated in the .ext hierarchical netlist file.

In addition to saving a classified and ranked BF list, CARP saves a modular, hierarchical circuit representation in the switch-level description language (SDL) developed for this research. Modules in SDL define formal parameters corresponding to external nodes and internal nodes. At the top level of the hierarchy, a network module defines the relationship of primary inputs, observable outputs, $V_{DD}$, and $V_{SS}$ to the modules making up the network. Labels uniquely identify instances of modules, nodes, and edge primitives so that they can be easily referred to during test generation, by using a path name of the form /instance_name/circuit_element.

CARP can produce detailed statistics showing the percentage of bridging faults belonging to each of the classes in Figure 12. This was of use during development of the fault sensitization
heuristics used for the testing of various types of BF's, by indicating where efforts should be concentrated to produce high quality heuristics. Table 2 shows the CARP classification results for gate-level bridging faults generated from the benchmark circuit layouts. The results show that for these circuits, from 65 to 86 percent of all realistic bridging fault can be treated at the gate-level. Of the five most specialized sub-categories of gate-level BF's, wire to wire BF's affecting different gates are by far the most common, representing from 42 to 73 percent of all bridging faults.

Table 3 and Table 4 show the CARP classification results for switch-level BF's in the benchmark circuit layouts. These results show that switch-level faults comprise from 14 to 35 percent of the total bridging fault count, with most faults being fairly evenly scattered across the internal node stuck-at, transistor stuck-on, and wire to internal node BF categories. Even though switch-level BF's constitute a smaller proportion of the total fault count than gate-level BF's, these results show that it is imperative to take switch-level BF's into consideration if very high fault coverage of unrestricted BF's is to be assured. Even the least likely class of BF's, bridges between internal nodes in different gates, is significant when the fault coverage results presented later in this section are considered.

4.6.3. Test Generator Implementation

The complete I$_{DOQ}$ test generation algorithm, referred to here as BODEM, was implemented on a Digital Equipment Corporation DECstation 3100 running ULTRIX. BODEM is a complete automatic test pattern generation system, as it can perform deterministic test generation, fault simulation, and test set compaction. BODEM is written in C++, and consists of approximately 25,000 lines of code. It is implemented on top of an object-oriented test generation framework
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Total Bridging Faults</th>
<th>Gate-Level Bridging Faults</th>
<th>Wire Stuck-At</th>
<th>Wire to Wire</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vdd-Wire</td>
<td>GND-Wire</td>
</tr>
<tr>
<td>c17</td>
<td>72</td>
<td>55</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16.67%</td>
<td>16.67%</td>
</tr>
<tr>
<td>c432</td>
<td>2,843</td>
<td>2,127</td>
<td>174</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6.12%</td>
<td>6.12%</td>
</tr>
<tr>
<td>c499</td>
<td>5,395</td>
<td>3,522</td>
<td>221</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.10%</td>
<td>4.10%</td>
</tr>
<tr>
<td>c880</td>
<td>5,454</td>
<td>4,181</td>
<td>272</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.99%</td>
<td>4.99%</td>
</tr>
<tr>
<td>c1355</td>
<td>7,105</td>
<td>5,614</td>
<td>407</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.73%</td>
<td>5.73%</td>
</tr>
<tr>
<td>c1908</td>
<td>8,318</td>
<td>5,861</td>
<td>341</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.10%</td>
<td>4.10%</td>
</tr>
<tr>
<td>c2670</td>
<td>18,958</td>
<td>15,878</td>
<td>694</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.66%</td>
<td>3.66%</td>
</tr>
<tr>
<td>c3540</td>
<td>23,275</td>
<td>18,983</td>
<td>772</td>
<td>772</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.32%</td>
<td>3.32%</td>
</tr>
<tr>
<td>c5315</td>
<td>51,707</td>
<td>44,560</td>
<td>1,323</td>
<td>1,323</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.56%</td>
<td>2.56%</td>
</tr>
<tr>
<td>c6288</td>
<td>34,057</td>
<td>27,381</td>
<td>1,880</td>
<td>1,880</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.52%</td>
<td>5.52%</td>
</tr>
<tr>
<td>c7552</td>
<td>69,058</td>
<td>59,007</td>
<td>1,680</td>
<td>1,680</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.43%</td>
<td>2.43%</td>
</tr>
<tr>
<td>Circuit</td>
<td>Total Bridging Faults</td>
<td>Switch-Level Bridging Faults</td>
<td>Internal Node Stuck-At</td>
<td>Wire to Internal Node</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td></td>
<td>Vdd-Node</td>
<td>GND-Node</td>
<td>Input to Node</td>
<td>Output to Node</td>
</tr>
<tr>
<td>c17</td>
<td>72</td>
<td>17</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>23.61%</td>
<td></td>
<td>1.39%</td>
<td>6.94%</td>
</tr>
<tr>
<td>c432</td>
<td>2,843</td>
<td>716</td>
<td>134</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>25.18%</td>
<td></td>
<td>4.71%</td>
<td>3.73%</td>
</tr>
<tr>
<td>c499</td>
<td>5,395</td>
<td>1,873</td>
<td>307</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td>34.72%</td>
<td></td>
<td>5.69%</td>
<td>5.84%</td>
</tr>
<tr>
<td>c880</td>
<td>5,454</td>
<td>1,273</td>
<td>187</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>23.34%</td>
<td></td>
<td>3.43%</td>
<td>3.80%</td>
</tr>
<tr>
<td>c1355</td>
<td>7,105</td>
<td>1,491</td>
<td>259</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>20.99%</td>
<td></td>
<td>3.65%</td>
<td>3.65%</td>
</tr>
<tr>
<td>c1908</td>
<td>8,318</td>
<td>2,457</td>
<td>378</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>29.54%</td>
<td></td>
<td>4.54%</td>
<td>5.36%</td>
</tr>
<tr>
<td>c2670</td>
<td>18,958</td>
<td>3,080</td>
<td>417</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>16.25%</td>
<td></td>
<td>2.20%</td>
<td>3.16%</td>
</tr>
<tr>
<td>c3540</td>
<td>23,275</td>
<td>4,292</td>
<td>725</td>
<td>683</td>
</tr>
<tr>
<td></td>
<td>18.44%</td>
<td></td>
<td>3.11%</td>
<td>2.93%</td>
</tr>
<tr>
<td>c5315</td>
<td>51,707</td>
<td>7,147</td>
<td>986</td>
<td>1,379</td>
</tr>
<tr>
<td></td>
<td>13.82%</td>
<td></td>
<td>1.91%</td>
<td>2.67%</td>
</tr>
<tr>
<td>c6288</td>
<td>34,057</td>
<td>6,676</td>
<td>1,224</td>
<td>1,160</td>
</tr>
<tr>
<td></td>
<td>19.60%</td>
<td></td>
<td>3.59%</td>
<td>3.41%</td>
</tr>
<tr>
<td>c7552</td>
<td>69,058</td>
<td>10,051</td>
<td>1,579</td>
<td>1,695</td>
</tr>
<tr>
<td></td>
<td>14.55%</td>
<td></td>
<td>2.29%</td>
<td>2.45%</td>
</tr>
</tbody>
</table>
Table 4. Switch-Level Bridging Fault Classification Results, Part 2

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Total Bridging Faults</th>
<th>Switch-Level Bridging Faults</th>
<th>Transistor Stuck-On</th>
<th>Internal Node to Internal Node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>N-Type</td>
<td>P-Type</td>
</tr>
<tr>
<td>c17</td>
<td>72</td>
<td>17</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>23.61%</td>
<td>6.94% 1.39%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c432</td>
<td>2,843</td>
<td>716</td>
<td>82</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>25.18%</td>
<td>2.88% 3.52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c499</td>
<td>5,395</td>
<td>1,873</td>
<td>154</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>34.72%</td>
<td>2.85% 2.63%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c880</td>
<td>5,454</td>
<td>1,273</td>
<td>151</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>23.34%</td>
<td>2.77% 2.38%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1355</td>
<td>7,105</td>
<td>1,491</td>
<td>195</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>20.99%</td>
<td>2.74% 2.98%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1908</td>
<td>8,318</td>
<td>2,457</td>
<td>269</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>29.54%</td>
<td>3.23% 2.61%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2670</td>
<td>18,958</td>
<td>3,080</td>
<td>473</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>16.25%</td>
<td>2.49% 1.37%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c3540</td>
<td>23,275</td>
<td>4,292</td>
<td>529</td>
<td>571</td>
</tr>
<tr>
<td></td>
<td>18.44%</td>
<td>2.27% 2.45%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5315</td>
<td>51,707</td>
<td>7,147</td>
<td>1,087</td>
<td>695</td>
</tr>
<tr>
<td></td>
<td>13.82%</td>
<td>2.10% 1.34%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c6288</td>
<td>34,057</td>
<td>6,676</td>
<td>918</td>
<td>1,002</td>
</tr>
<tr>
<td></td>
<td>19.60%</td>
<td>2.70% 2.94%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c7552</td>
<td>69,058</td>
<td>10,051</td>
<td>1,131</td>
<td>984</td>
</tr>
<tr>
<td></td>
<td>14.55%</td>
<td>1.64% 1.42%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
originally written to perform single stuck-at fault test generation, and makes heavy use of the object-oriented language features of C++. For example, the module and bridging fault taxonomies shown in Figure 11 and Figure 12 are translated directly into class inheritance hierarchies. BODEM parses the .sdl switch-level circuit descriptions generated by CARP, and for each BF read from the corresponding .bf fault list file, attempts to generate a test for the fault.

Typically, test generation algorithms are evaluated based on the traditional fault coverage criteria, where the fault coverage is simply the percentage of possible faults considered by the fault model that are tested by the set of generated test vectors [14]. With the bridging fault model, however, there are an exceedingly large number of possible faults, even when only realistic BF’s are considered. Many of these bridging faults are not at all likely to occur. An appropriate way of dealing with this is to base fault coverage analysis on the relative likelihood of fault occurrence. For example, if \( w_i \) is the relative weight of the occurrence of fault \( f_i \), and \( d_i = 1 \) if fault \( f_i \) is tested and 0 otherwise, then the weighted fault coverage considering \( n \) faults is

\[
FC = d_1w_1 + d_2w_2 + \ldots + d_nw_n, \text{ or }
\]

\[
FC = \sum_{i=1}^{n} d_iw_i
\]

If the weight \( w_i \) of a fault is very small, then the effect of not testing for fault \( f_i \) is also very small. In addition to reporting standard fault coverage, BODEM takes advantage of the BF probability data CARAFE makes available with extracted BF’s to report weighted fault coverage results, as a more accurate indication of test set quality.
To evaluate the performance of the BODEM implementation, it was run on the switch-level benchmark circuits in several modes, including test generation only, test generation with fault simulation (full ATPG), and full ATPG with an additional reverse order fault simulation pass to perform test set compaction. For the gate-level algorithm, a backtrack limit of 20 was used. The switch-level algorithm allowed up to 10 backtracks in the switch-level portion of test generation, and for each switch-level backtrack, up to 20 backtracks in the gate-level portion of test generation.

Table 5 shows both standard and weighted fault coverage results obtained by performing test generation only, and test generation with fault simulation. Standard and weighted percentages are also given for faults aborted during test generation when the backtracking limit was exceeded, and faults not covered during fault simulation. Because BODEM is a deterministic, complete test algorithm, it can identify bridging faults that are untestable, as long as the backtracking limit is not exceeded. The figures for untestable faults indicate the identification of BF's for which it is impossible to sensitize the fault site with the required logic values for fault detection.

Examining the figures in the "Tested" column of Table 5 shows that the I_{DDQ} test implementation is successful in achieving very high fault coverage, even when no fault simulation is used. The standard fault coverages in the test generation only case range from 94.9 to 100 percent, show that explicit deterministic test generation is possible for almost all realistic bridging faults in the benchmark circuits. The test set coverage is of course even higher when fault simulation is utilized, ranging from 97.9 to 100 percent. The slightly lower figures for c1355, which in its original primitive gate form is c499 with Xor gates expanded into 4-Nand gate equivalents, were due to excessive backtracking in the gate-level BODEM algorithm attributable to the circuit
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Total Bridging Faults</th>
<th>Test Generation Only (Standard/Weighted Percentages)</th>
<th>With Fault Simulation (full ATPG) (Standard/Weighted Percentages)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tested</td>
<td>UnTestable</td>
</tr>
<tr>
<td>c17</td>
<td>72</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>c432</td>
<td>2,843</td>
<td>98.56</td>
<td>99.16</td>
</tr>
<tr>
<td>c499</td>
<td>5,395</td>
<td>97.89</td>
<td>97.53</td>
</tr>
<tr>
<td>c880</td>
<td>5,454</td>
<td>99.96</td>
<td>99.97</td>
</tr>
<tr>
<td>c1355</td>
<td>7,105</td>
<td>94.86</td>
<td>95.21</td>
</tr>
<tr>
<td>c1908</td>
<td>8,318</td>
<td>99.81</td>
<td>99.80</td>
</tr>
<tr>
<td>c2670</td>
<td>18,958</td>
<td>99.08</td>
<td>98.80</td>
</tr>
<tr>
<td>c3540</td>
<td>23,275</td>
<td>99.42</td>
<td>99.45</td>
</tr>
<tr>
<td>c5315</td>
<td>51,707</td>
<td>99.67</td>
<td>99.62</td>
</tr>
<tr>
<td>c6288</td>
<td>34,057</td>
<td>95.72</td>
<td>96.02</td>
</tr>
<tr>
<td>c7552</td>
<td>69,058</td>
<td>99.73</td>
<td>99.59</td>
</tr>
</tbody>
</table>
topology. The weighted fault coverage figures were in most cases equivalent to or higher than the corresponding standard fault coverage, showing that the test sets do manage to cover the most likely to occur faults. In no case was the weighted fault coverage significantly lower than the standard fault coverage.

Table 6 shows the size of the $I_{def}$ test sets resulting from four scenarios: (1) test generation only, (2) test generation followed by application of a simplistic test set compaction scheme, (3) test generation with interleaved fault simulation, and (4) full ATPG with a final reverse order fault simulation pass. The test sets resulting from test generation only are huge due to the large numbers of extracted BF’s. Application of the simple compaction scheme does produce significant reductions in test set size, but the sizes remain too large for practical use. Table 6 shows that the introduction of interleaved fault simulation into test generation is both a necessary and sufficient step for reducing $I_{def}$ test set size. With this addition, the test sets become small enough to make their use in an $I_{def}$ test environment practical. The addition of reverse order fault simulation produces an additional incremental reduction in test set size for all circuits, with a significant reduction in several cases. The extremely high test set compaction ratios shown in Table 6 are very promising, and appear to indicate that test set compression increases with increasing circuit and fault list size.

Table 7 presents the run-time requirements of the BODEM implementation on a Digital Equipment Corporation DECstation 3100 in three modes, test generation only, and full ATPG with and without reverse order fault simulation. The CPU times for the test generation only mode are fairly substantial, reflecting the effort involved in attempting to generate a deterministic test for each and every bridging fault. Even though the total CPU time figures increase rapidly with large
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Test Generation Only</th>
<th>Simplistic Test Set Compaction</th>
<th>Full ATPG (with Fault Simulation)</th>
<th>Full ATPG with Reverse Order Fault Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>c17</td>
<td>72</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>c432</td>
<td>2,802</td>
<td>188</td>
<td>38</td>
<td>30</td>
</tr>
<tr>
<td>c499</td>
<td>5,281</td>
<td>2,169</td>
<td>63</td>
<td>60</td>
</tr>
<tr>
<td>c880</td>
<td>5,452</td>
<td>213</td>
<td>41</td>
<td>33</td>
</tr>
<tr>
<td>c1355</td>
<td>6,740</td>
<td>316</td>
<td>55</td>
<td>53</td>
</tr>
<tr>
<td>c1908</td>
<td>8,302</td>
<td>880</td>
<td>107</td>
<td>98</td>
</tr>
<tr>
<td>c2670</td>
<td>18,784</td>
<td>587</td>
<td>62</td>
<td>50</td>
</tr>
<tr>
<td>c3540</td>
<td>23,140</td>
<td>963</td>
<td>84</td>
<td>69</td>
</tr>
<tr>
<td>c5315</td>
<td>51,535</td>
<td>1,886</td>
<td>67</td>
<td>60</td>
</tr>
<tr>
<td>c6288</td>
<td>32,599</td>
<td>1,269</td>
<td>34</td>
<td>30</td>
</tr>
<tr>
<td>c7552</td>
<td>68,874</td>
<td>2,581</td>
<td>97</td>
<td>82</td>
</tr>
<tr>
<td>Circuit</td>
<td>Test Generation Only</td>
<td>Full ATPG (TG with Fault Simulation)</td>
<td>Full ATPG with Reverse Order Fault Simulation</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
<td>--------------------------------------</td>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total CPU time</td>
<td>CPU time per fault</td>
<td>Total CPU time</td>
<td>CPU time per fault</td>
</tr>
<tr>
<td>c17</td>
<td>0.5</td>
<td>0.0069</td>
<td>0.3</td>
<td>0.0042</td>
</tr>
<tr>
<td>c432</td>
<td>39.5</td>
<td>0.0139</td>
<td>8.0</td>
<td>0.0028</td>
</tr>
<tr>
<td>c499</td>
<td>151.1</td>
<td>0.0280</td>
<td>15.1</td>
<td>0.0028</td>
</tr>
<tr>
<td>c880</td>
<td>41.8</td>
<td>0.0077</td>
<td>4.6</td>
<td>0.0008</td>
</tr>
<tr>
<td>c1355</td>
<td>443.4</td>
<td>0.0624</td>
<td>47.1</td>
<td>0.0066</td>
</tr>
<tr>
<td>c1908</td>
<td>170.9</td>
<td>0.0205</td>
<td>15.1</td>
<td>0.0018</td>
</tr>
<tr>
<td>c2670</td>
<td>259.3</td>
<td>0.0137</td>
<td>28.2</td>
<td>0.0015</td>
</tr>
<tr>
<td>c3540</td>
<td>868.5</td>
<td>0.0373</td>
<td>62.9</td>
<td>0.0027</td>
</tr>
<tr>
<td>c5315</td>
<td>939.2</td>
<td>0.0182</td>
<td>48.5</td>
<td>0.0009</td>
</tr>
<tr>
<td>c6288</td>
<td>7228.6</td>
<td>0.2123</td>
<td>229.5</td>
<td>0.0067</td>
</tr>
<tr>
<td>c7552</td>
<td>1968.6</td>
<td>0.0285</td>
<td>86.4</td>
<td>0.0013</td>
</tr>
</tbody>
</table>
circuit sizes, examining the "CPU time per fault" column indicates that the increasing CPU requirements are due primarily to the sizeable bridging fault lists extracted for large circuits. Also, the topology of a circuit appears to have more influence on the CPU time required per fault than the size of the circuit does. This was especially evident with c1355 and c6288, for which a large portion of the run-time was due to significant amounts of backtracking during deterministic test pattern generation.

The introduction of fault simulation significantly reduces the CPU time required for test generation, by factors of 20 to 30 for the three largest benchmark circuits. The total CPU time figures are quite promising, especially considering the size of the fault lists being handled and the overhead due to the use of switch-level algorithms. Even with the addition of the reverse order fault simulation pass, the CPU requirements of the BODEM application are very reasonable. These results indicate that, using the techniques developed in this research, \textbf{I}_{\text{dQQ}} test generation for realistic, unrestricted bridging faults is practical for circuit sizes that could previously be treated only at the gate-level.

\section{Summary}

This chapter presented a test generation methodology that uses a modular, hierarchical approach to support explicit \textbf{I}_{\text{dQQ}} test generation for unrestricted bridging faults in large CMOS circuits. The switch-level circuit model used at the lowest level of the hierarchy accurately represents the structure of CMOS design structures and realistic faults. Results indicate that this approach should be able to generate high quality \textbf{I}_{\text{dQQ}} test sets for realistic BF's in very large circuits, with
reasonable computational requirements. Although the number of likely bridging faults in a typical VLSI circuit might make the task of generating a test for each and every fault intractable, the algorithm presented in this research provides at the very least the ability to generate tests for (1) faults that logic-level observation techniques fail to detect, and (2) "hard-to-detect" bridging faults between nodes that rarely change value or fail to assume different values under tests generated by other test generation techniques.
Chapter 5. A Circuit Partitioning Approach to Test Generation

The goal of the research presented in this chapter is to determine the amount of parallelism that can potentially be exploited by a circuit partitioning approach to distributed test generation. The major phases of test generation are examined and the potential parallelism existing in each phase is described. The effectiveness of a circuit partitioning approach is evaluated with respect to its expected performance in a distributed processing environment by deriving performance models for a restricted class of circuits. To determine upper bounds on the concurrency that might be extracted from typical circuits, a conflict-free "perfect" test generation algorithm that operates without backtracking is simulated. Results are presented for gate-level test generation using several benchmark circuits.

The following section briefly presents background and previous approaches related to this research. Section 5.2 describes the potential parallelism existing in the various phases of test generation. Section 5.3 discusses conditions that maximize concurrency in test generation. Performance models for a restricted class of circuits are derived in Section 5.4. Section 5.5

*Portions of this work also appear in [10].
presents the method used to simulate a perfect test generator, and the metrics used to gauge the
effectiveness of the simulated circuit partitioning approach. Results are presented in Section 5.6.

5.1. Background

A promising approach for increasing performance that has been successfully applied to logic and
fault simulation is the incorporation of parallel and distributed computing techniques. There is
increasing evidence that the use of parallel processing can effectively reduce the execution time
of test generation, and work in this area has proceeded in a number of directions. In the most
obvious and simplest scheme for exploiting concurrency, fault parallelism [77, 40, 69], the fault
set for which tests will be generated is distributed across available processors, and each processor
generates test vectors for the faults assigned to it. Search parallelism is a parallel branch and
bound technique that speeds up backtracking by exploring different portions of the search space
concurrently [69, 78]. By distributing the search space over the available processors and having
each processor search its own subspace simultaneously, the time required for backtracking can be
reduced. Search space allocation can be accomplished by simply dividing the search space into
equal parts, or by using heuristics to choose portions of the search space where the probability of
finding a solution is higher. Heuristic parallelism [22] is a form of search parallelism that relies
on the fact that faults dropped due to excessive back-tracking by one heuristic method are often
detected by the use of a different heuristic. Each processor uses a different heuristic to guide the
search for a single fault.
In circuit partitioning, the circuit under test is divided into distinct portions and distributed among processors. Every processor analyzes and generates tests only for faults in its own subset of the circuit, and processors cooperate to generate a test for any particular fault. Circuit partitioning avoids a significant limitation of other approaches by eliminating the need for every processor to maintain a complete representation of the circuit under test, and thus offers the possibility of test generation for very large circuits. A disadvantage is that the degree of parallelism decreases as test generation proceeds and processors complete handling of faults within their subset of the circuit. Another disadvantage is the very high communication requirement between processors. Although circuit partitioning techniques have been used in some massively parallel approaches to test generation [57, 20], these approaches are limited in the size of circuits that can be handled. Finally, simulation parallelism is a subset of circuit parallelism, and refers to the use of parallel logic simulation techniques to accelerate the forward implication phase of test generation.

5.2. Parallelism in Test Generation

Test generation is most commonly performed at the gate-level using the stuck-at fault model, where a line or gate output is assumed to be permanently set to logic 0 or logic 1 in the presence of a fault. Except where noted, a test generation problem with the following characteristics will be used as a basis for discussion: (1) a combinational gate-level circuit representation, (2) a single stuck-at fault model, and (3) a PODEM-type control algorithm. This form of test generation is one of the most common configurations appearing in the literature on parallel and distributed test generation. The reader is assumed to be familiar with the details of the PODEM algorithm; descriptions appear in Chapter 3 and [43].
The basic strategies of test generation consist of a sequence of phases commonly labeled as (1) fault sensitization, (2) backward line justification of logic values to controllable inputs, (3) forward propagation or implication of faulty values to observable outputs, and (4) fault simulation. The last three phases introduce possibilities for taking advantage of parallelism.

5.2.1. Fault Sensitization

After selecting the fault that is to be tested, it is necessary to identify the subcircuit in which the fault occurs and devise a local test that, when applied to the subcircuit's input(s), will produce a faulty value on one or more outputs. Fault sensitization is inherently a serial operation, as a single stuck-at fault may only affect a single gate instance. Thus the fault sensitization phase provides no opportunity for parallelism.

5.2.2. Backward Line Justification

After sensitizing a fault, it is necessary at some point during test generation to propagate the resulting objective values back to the primary inputs of the circuit. This process, referred to as backward line justification, is accomplished by examining individual circuit primitives, and justifying signal values on the outputs of primitives by specifying implied values on one or more inputs. The term backtracing is used to denote the process of repeatedly justifying objective values backwards through a circuit until a primary input is reached.

Due to the presence of reconvergent fan-out in almost all realistic circuits, the backtrace procedure is heuristic in nature. Reconvergent fan-out introduces dependencies in the values that must be assigned to circuit wires, and can cause conflicts during backtracing. This necessitates the incorporation of a backtracking search procedure into test generation to resolve conflicts.
There is inherent parallelism in the backtracing procedure, due to the fact that backwards justification of objective values on the output(s) of a circuit primitive often requires the satisfying of multiple objective values on the inputs. Depending on circuit topology, these multiple objectives values can often be justified in parallel, at least until fan-out points are encountered. Although the single backtrace procedure employed by PODEM fails to take advantage of this property, more advanced test generation algorithms such as FAN employ a multiple backtrace technique using sophisticated heuristics to reduce the computational cost required for backtracing objectives through circuits. The logical extension of this concept is to use more than one processor to execute the multiple backtrace procedure in parallel.

5.2.3. Forward Propagation

During test generation the good and faulty logic values generated on circuit wires as a result of primary input assignments must be propagated to the circuit's primary outputs. The process of determining the outputs of circuit primitives based on their inputs is known as forward implication. Unlike the heuristic multiple backtrace procedure, forward implication is a deterministic procedure, but it is nonetheless a major component of the computational load in test generation.

In PODEM, a forward implication is performed after every primary input assignment, making backtracks fairly expensive. Every forward implication is performed taking \( O(w) \) time in the worst case, where \( w \) is the number of signal lines in the circuit. Because forward implication is essentially logic simulation with more values, parallel logic simulation techniques can be used to speed up this phase of test generation [90, 91, 32]. One such approach in particular, an adaptation of the Chandy-Misra distributed time discrete-event algorithm [91], has shown promise for
extracting useful amounts of parallelism during logic simulation. The circuit partitioning strategy investigated in this work is exactly what is needed to support such an approach.

5.2.4. Fault Simulation

Test generation and fault simulation are typically interleaved in a uniprocessor environment by following each test generation step by a fault simulation step to identify additional faults that are detected by the test and to delete those faults from the fault list. Since the complexity of fault simulation is much lower than that of test generation, the fault simulation phase can identify faults having the same test vector and avoid generating tests for those faults, saving a significant amount of effort in test generation. In addition, the test set is dynamically compacted since tests are not generated for the additional faults detected during the fault simulation step.

Fault simulation is in many ways very similar to logic simulation, although fault simulation exhibits higher complexity. As with forward implication, a combined test generation/fault simulation strategy works well with a circuit partitioning approach, supporting the use of parallel simulation techniques [32]. The remainder of this chapter focuses solely on the parallelism available in test generation, so analysis and discussion are limited to test generation without accompanying fault simulation. Some of the complexities introduced by including fault simulation in distributed test generation are considered in detail in [77, 40].
5.3. Determining Maximum Concurrency in Test Generation

Clearly, specific realizations of the PODEM test generation algorithm will differ in the details of how they implement the test generation operations such as backtracking and logic simulation described in [43]. However, the fundamental operations and control flow of PODEM can be abstracted so that the potential concurrency in test generation can be determined. This section discusses the optimal conditions that are assumed to exist to provide for maximum parallelism during test generation, and the procedure for simulating the behavior of a perfect parallel test generator.

5.3.1. Conditions for Maximum Parallelism

In a circuit partitioning approach to test generation, the backtracking scheme used in PODEM to resolve conflicts during test generation would tend to serialize the algorithm by forcing the sequential processing of primary input assignments and new internal objectives. This sequential overhead is not present during the generation of tests for which no conflicts arise. For such tests, the control flow of PODEM is greatly simplified and the computational overhead of test generation is dominated by the backward justification and forward propagation phases. The pseudocode shown in Figure 17 captures the essential behavior of PODEM during test generation for a non-redundant fault in the absence of conflicts and backtracking. Each iteration of the while loop in Figure 17 is referred to as single pass in test generation.

During each test generation pass, a single objective in the form of a circuit line and associated target logic value is chosen, a single PI is selected and assigned a value by the backtrace
procedure generate_test(fault)
    while test is not found
        objective = choose_objective();
        PI = backtrace(objective);
        simulate_circuit(PI);
    endwhile
end

Figure 17. Simplified control flow for conflict-free PODEM.

procedure, and any changes on circuit lines implied by the new PI value are determined via a circuit simulation. Given a hypothetical multiple backtrace procedure that is capable of always assigning appropriate logic values without causing conflicts, the same control flow applies, but multiple PI's may be assigned during a single pass. Under these conditions, two possibilities exist for enhancing test generation performance with parallel processing: (1) concurrent justification of circuit primitives during backtracing, and (2) parallel simulation of primitives during forward implication. Both parallel operations are considered in this research.

5.3.2. Definition of Concurrency

As a measure of the maximum concurrency that is possible with a circuit partitioning approach, we consider how many operations can proceed in parallel given an unlimited number of processors, no synchronization costs or scheduling overhead, and a mapping that assigns a single circuit element to every processor. Both backtrace and simulation operations are assumed to take exactly one unit of time, referred to as a time step, for all circuit elements. With these assumptions, test generation proceeds as a series of time steps, with one or more circuit elements
being justified or implied during each step. The parameters of interest are the maximum and average number of elements that can be processed during a single time step, providing metrics for the best case and expected parallelism, respectively, that is available during test generation for a single fault.

5.4. Performance Models

This section presents the formulation of a set of theoretical models that capture the behavior of a circuit partitioning approach to test generation for a restricted class of circuits. Closed form solutions representing upper bounds on speedup in terms of runtime and communication overhead are derived. These models are used to evaluate the effectiveness of a circuit partitioning approach with respect to its expected performance in a distributed processing environment.

5.4.1. Assumptions and Terminology

The limiting factor that determines whether or not a circuit partitioning approach to test generation will be beneficial is the amount of concurrency that can be taken advantage of during test generation. This depends on both the amount of concurrent activity exhibited during test generation by the circuit under test, and the ability of the test generation algorithm to execute operations in parallel and utilize this concurrency. As discussed previously in this chapter, the following two test generation operations are assumed to contain potential parallelism:
**Forward Implication:** Values at the inputs of two or more instances (gates) can be implied simultaneously if and only if the circuit contains a fanout point feeding the instances and the instances are assigned to distinct processors.

**Backwards Justification:** Values on the outputs of two or more instances can be justified simultaneously if and only if the circuit contains a fanin point that is fed by the instances and the instances are assigned to distinct processors.

Given a distributed processing environment in which multiple communicating processors are available, communication overhead is a second important factor that determines whether parallelism can be effectively utilized. In the following it is assumed that a bus-style interconnection network is used, so that multiple communications cannot be overlapped. In other words, communication costs are always additive. This would be the case, for example, in a distributed environment composed of a network of workstations. It is also assumed that an unlimited number of processors are available, and that task assignment is one instance or gate per processor. The following terms are used when discussing characteristics of a distributed environment:

- $P$ The number of available processors.
- $R$ The uniform processing cost for a test generation operation on any instance, on any processor. $R$ designates the computation time taken by a processor to perform fault sensitization, forward implication, or backward justification for an instance.
The mean communication cost for any transfer of information between processors, including waiting time due to contention, software overhead, and data transfer time between two processors.

The performance models derived in this section assume that test generation is performed on gate-level circuits using the classical single stuck-at fault model, utilizing an error-free test generation algorithm that produces no conflicts and thus requires no backtracking. The following circuit terminology is used to describe the characteristics of the circuits discussed here:

- \( PI \): Number of primary inputs.
- \( PO \): Number of primary outputs.
- \( L \): Number of logic levels in the circuit.
- \( N \): Number of module instances in the circuit.
- \( W \): Number of internal "wires" or connections, each fanout branch counts as a wire.
- \( T_i \): Total time to generate a test for one fault \( f_i \).

The class of circuits considered here is limited to those that can be described as a collection of gates arranged in a binary tree. An example four level binary tree is shown in Figure 18. This class of circuits is fan-out free and can thus be processed without backtracking by test generation algorithms. The characteristics of a binary tree circuit expressed in terms of the variables defined above are:

\[
PO = 1 \\
PI = 2^L
\]
Figure 18. Fan-out free binary tree circuit.
\[ N = 2^L - 1 \]
\[ L = \log_2(N + 1) \]
\[ W = 2^L - 2 = 2(2^{L-1} - 1) \]
\[ W_i = 2^{L_i} \quad \text{(width at level } i) \]

Although limiting the treatment of circuits to those in the binary tree class may seem to be restrictive, this class of circuits is a good approximation of the fan-out free regions existing inside more general circuits. These regions provide opportunity for significant parallelism in test generation by allowing multiple backtraces.

In addition to the above assumptions, the following two conditions are assumed to always be true in the remainder of this section.

1. Forward implications are always successful, and always generate a backward justification requirement, if backward justification is possible.
2. Backward justifications are always successful, and always require the setting of all inputs of the instance or gate being justified.

These conditions produce the maximum potential for parallelism by maximizing (1) the number of primary inputs that must be set to generate a test, and (2) the number of test generation tasks that can be executed concurrently.

In the remainder of this section, performance models are derived to capture the behavior of a parallel circuit partitioning approach for test generation as it is applied to binary tree circuits under
four different sets of conditions. In the first two cases, communication overhead is ignored, and the performance models indicate the speedup possible when considering only the concurrency available in test generation. In the second two cases, communication overhead is accounted for, giving a more accurate pair of models that provides a measure for speedup in terms of the ratio of runtime to communication.

5.4.2. Case 1: Non-overlapped implication and justification

This case examines the concurrency existing in test generation for a binary tree circuit when multiple backtrace is possible, communication costs are not included, and the implication and justification tasks are performed as distinct, serial phases. The alternating implication and justification phases would result from employing a conservative distributed control algorithm such as the Chandy-Misra algorithm.

Before discussing parallelism, consider serial test generation, in which all instances are assigned to a single processor. Since both forward implication (FI) and backward justification (BJ) are always assumed to be successful, each instance in the circuit only has to be processed just once. The total test generation time for any single fault is the same and is given by:

Serial Case: \[ T_s = R \cdot N = R(2^L - 1) \]

For the special case of a fault at the (single) output of a binary tree, the parallel test generation time is simply proportional to the number of levels in the tree, since every level can be processed in parallel in time \( R \). Test generation time is thus given by:
Parallel Case: \( T_p = R \cdot L \)

The speedup can then be calculated from:

\[
\text{Speedup} = \frac{T_s}{T_p} = \frac{R(2^L - 1)}{RL} = \frac{2^L - 1}{L} = \frac{2^L}{L} \quad \text{for } L \gg 1
\]

To consider a fault at level \( v \) of the tree, some additional terms are introduced. Let \( I \) be the time needed to imply a logic value through a module \( (I = R) \), and \( J_v \) be the total time needed to sensitize and justify a fault in an instance at level \( v \) of the tree. Since multiple backtrace is assumed, it will take \( R \) units of time for BJ for the faulty module at level \( v \), \( R \) for the two modules in level \( v-1 \) that feed \( m \), \( R \) for the four modules in level \( v-2 \), and so on. Therefore, \( J_v = R \cdot v \).

For the assumptions in this case, test generation proceeds as follows. First, sensitize the fault at level \( v \) by using BJ to set a primary input \( (J_v) \). Then propagate the fault value through a module in the next level of logic \( (I) \). Now, the other primary input of this instance must be set via BJ \( (J_{v+1}) \). This process repeats until the primary output is reached with a final implication, followed by a final BJ stage \( (J_{L-1}) \).

\[
T_p(v) \quad = J_v + I + J_v + I + J_{v+1} + \ldots + I + J_{L-1}
\]
\[
T_p(v) = Rv + R(L-v) + \sum_{i=v}^{L-1} R_i = RL - Rv + Rv + \frac{1}{2} R[L(L-1)-v(v-1)] = \frac{2RL + R[L(L-1)-v(v-1)]}{2} = \frac{RL^2 + RL - Rv^2 + Rv}{2} = \frac{R(L^2 + L - v^2 + v)}{2}
\]

Now using \( J_v = R \cdot v, I = R \)

\[
\text{Speedup} = \frac{T_s}{T_p} = \frac{R(2^L - 1)}{R(L^2 + L - v^2 + v)} = \frac{2^{L+1} - 2}{L^2 + L - v^2 + v} = S_v
\]

Note that as \( v \) approaches \( L \), the expression for speedup reduces to the one derived for a fault at the output of the binary tree.

\[
S_v \mid_{v=L} = \frac{2^L - 1}{L}
\]
5.4.3. Case 2: Overlapped implication and justification

The assumptions for this case are essentially the same as for Case 1. The difference here is that forward implication and backwards justification are allowed to take place simultaneously, taking advantage of the existence of a perfect test generation algorithm that never produces any conflicts, or has to deal with redundant faults. The overlapping implication and justification phases would result from employing an optimistic distributed control algorithm such as the Time Warp algorithm.

This case reduces to Case 1 for a fault at the output of the tree, so only faults at level \( 0 \leq v < L \) are considered here.

Test generation time for the serial case is again the same no matter where the fault is located:

\[
T_s = R \cdot N = R(2^L - 1)
\]

For the parallel case, all I and J operations can be overlapped once started. The last justify operation (starting at the \( L-1 \) level of the circuit) will take the most time and will therefore dominate all other J operations. For \( 0 \leq v < L \),

\[
T_p = \left( \frac{L-1}{\sum I} \right) J_{L-1} = R(L-v) + R(L-1) = 2RL - Rv - R
\]

\[
= R(2L - v - 1)
\]

Speedup = \[
\frac{T_s}{T_p} = \frac{R(2^L - 1)}{R(2L - v - 1)} = \frac{2^L - 1}{2L - v - 1} = S
\]
5.4.4. Case 3: Non-overlapped implication and justification with communication overhead

This case examines the concurrency existing in test generation for a binary tree circuit when multiple backtrace is possible, implication and justification tasks are performed as distinct, serial phases, and communication costs are included. In this case and the next, the inclusion of communication overhead produces more accurate models that provide measures of speedup in terms of the ratio of runtime to communication. The communication cost for any transfer of information between processors, whether due to implication or justification, is modeled by a uniform time, C. Since a distributed environment using a bus-style interconnection network is assumed, the costs for multiple simultaneous communications are additive.

For serial test generation, no communication costs exist since all instances are assigned to a single processor. The total test generation time for a single fault is the same no matter where the fault is located, and is still given by:

\[ T_s = R \cdot N = R(2^L - 1) \]

For the special case of a fault at the output of a binary tree, test generation proceeds in a straightforward level-by-level fashion from the output instance to the input instances. Every level of instances can be processed in parallel with runtime \( R \), but the total time needed for communication between levels increases by a factor of two as each level is traversed, due to the topology of the binary tree. For example, the total test generation time for the circuit in Figure 18 would be \( T_p = R + 2C + R + 4C + R + 8C + R \), or \( T_p = 4R + 14C \). A general expression for the test generation time of an output fault is thus given by:
\[ T_p = R + 2^1C + R + 2^2C + \ldots + 2^L C + R \]
\[ = LR + C \sum_{i=1}^{L} 2^i \]
\[ T_p = LR + C(2^{L+1} - 2) \]

The speedup is then calculated as:

\[ \text{Speedup} = \frac{T_s}{T_p} = \frac{R(2^L - 1)}{LR + C(2^{L+1} - 2)} \]
\[ = \frac{2^L}{C \frac{2^{L+1}}{R}} \quad \text{for } L \gg 1 \]
\[ S = \frac{R}{2C} \]

For a fault at level \( v \) of the tree, the flow of test generation is more complicated. Assuming that the fault is at the input of an instance in the interior of the tree, an initial justify operation would be required. Following this, an implication through the faulty instance determines the value that must be implied on the other input to the instance, a justification is performed on that input, and then communication takes place to transfer the faulty instance output value to the connected instance in the next level. This imply, justify, and communicate pattern is repeated until the output of the tree circuit is reached. The parallel test generation time is then given by

\[ T_p = J_v + (I + J_i + C) + (I + J_{i+1} + C) + \ldots + (I + J_{L-1} + C) \]
\[ = J_v + \sum_{i}^{L-1} (I + J_i + C) = J_v + (I + C)(L - v) + \sum_{i}^{L-1} J_i \]
For implication, \( I = R \) as always. To derive \( J_v \), note that the modeling assumptions require that all instances in the subtree rooted at the fault site at level \( v \) must be evaluated during the BJ phase. The expression for the total justification time required is

\[
J_v = R v + \sum_{i=0}^{r-1} C 2^i = R v + C(2^r - 1)
\]

Substituting this expression,

\[
T_p = R v + C(2^r - 1) + (R + C)(L - v) + \sum_{i=0}^{L-1} [R_i + C(2^i - 1)]
\]

\[
= R v + C(2^r - 1) + R(L - v) + C(L - v) + \frac{1}{2} R(L(L - 1) - v(v - 1)) + C \sum_{i=0}^{L-1} 2^i - C(L - v)
\]

\[
= R v + R L - R v + \frac{1}{2} R L^2 - \frac{1}{2} R L - \frac{1}{2} R v + \frac{1}{2} R v + C 2^r - C + C L - C v + C(2^{L-1} - 2^r) - C L + C v
\]

\[
T_p = \frac{1}{2} R(L^2 + L - v^2 + v) + C(2^L - 1)
\]

The speedup is then given by

\[
\text{Speedup} = \frac{T_s}{T_p} = \frac{R(2^L - 1)}{\frac{1}{2} R(L^2 + L - v^2 + v) + C(2^L - 1)} = S_-
\]

These expressions are only valid for \( 0 \leq v < L \). Note that as \( C \) approaches 0, the expression for speedup reduces to the one derived for Case 1. For very large \( L \), the speedup becomes

\[
S_- = \frac{2^L - 1}{\frac{1}{2} (L^2 + L - v^2 + v) + \frac{C}{R}(2^L - 1)} \approx \frac{2^L}{\frac{C}{R} 2^L}
\]

\[
S_- = \frac{R}{C}
\]
5.4.5. Case 4: Overlapped implication and justification with communication overhead

The assumptions for this case are the same as for Case 3, except that implication and backwards justification are allowed to take place simultaneously. This case reduces to Case 3 for a fault at the output of the tree, so only faults at level $v$, $0 \leq v < L$ are considered here.

Just as in Case 2, justification at $J_{L-1}$ dominates all other justification times, but the communication costs incurred by justification are additive and must be accounted for. A new term, $C_v = C(2^v - 1)$, is introduced to model the communication overhead required for justification at level $v$. Using this term and the expression $J_v = Rv + C(2^v - 1)$ from Case 3, the parallel test generation time is given by:

$$T_p = \sum_{i=v}^{L-1} (I+C_i) + J_{L-1}$$

$$= I(L-v) + \sum_{i=v}^{L-1} C(2^i-1) + R(L-1) + C(2^{L-1}-1)$$

$$= R(L-v) + C(2^L-2^v) - C(L-v) + R(L-1) + C(2^{L-1}-1)$$

$$= RL - Rv + R - C2^L - C2^v - CL + C - C2^{L-1}$$

$$T_p = R(2L-v-1) + C(2^L + 2^{L-1} - 2^v + v - 1)$$

The speedup in this case is then

$$S_p = \frac{T_s}{T_p} = \frac{R(2^L-1)}{R(2L-v-1) + C(2^L + 2^{L-1} - 2^v + v - L - 1)}$$

For very large $L$, the speedup is approximately
\[ S_{\leq L} = \frac{2^L \cdot C \cdot 2^L}{R} \]
\[ = \frac{R}{C} \]

These equations are only valid for \( 0 \leq v < L \). Note that as \( C \) approaches 0, then the expression for speedup reduces to the one derived in Case 2.

### 5.4.6. Summary of Performance Models

The formulae derived in Case 1 and Case 2 indicate that there is certainly usable concurrency in test generation, at least given the assumptions and the restricted class of circuits considered in the analysis. Furthermore, the expressions for speedup indicate that for a binary tree circuit configuration, the use of parallel processing could result in significant speedups in test generation if communication overhead is negligible and can be ignored.

Unfortunately, the communication requirements introduced by using a circuit partitioning approach to test generation are substantial. Even with the extremely optimistic assumptions used to formulate the above performance models, the expressions derived for speedup with communication costs considered in Case 3 and Case 4 show that the influence of communication overhead on achievable speedup is considerable, making any performance improvement in a realistic distributed memory processing environment unlikely. For example, consider an optimistic scenario in which the time \( R \) to perform forward implication and backward justification is equal to the uniform communication time \( C \), resulting in a runtime to communication ratio of 1. This could be achieved by taking a large circuit and using a clustering task assignment strategy that places multiple connected gates on a single processor to increase the task granularity of test generation. Substituting the ratio \( R/C = 1 \) into the speedup expressions derived in Case 3 and Case 4 shows...
that the resulting speedups are less than 1. This indicates that the communication overhead introduced by the circuit partitioning approach would cause parallel test generation to actually run slower than a standard serial test generation algorithm.

5.5. Simulating a "Perfect" Test Generation Algorithm

A realistic multiple backtrace procedure is unable to operate by simply beginning at an initial objective and working towards primary inputs until a set of primary inputs are reached. Due to the presence of reconvergent fan-out in most circuits, performing a conflict-free backtrace generally requires special consideration and additional processing at fan-out points. For example, when a fan-out stem is reached during multiple backtrace in the FAN algorithm, all paths between the fan-out stem and its reconvergence point are checked to ensure that a consistent value is assigned for the fan-out stem.

To both eliminate the sequential behavior introduced by such a conflict-resolution scheme and avoid having to model the significant complexity of a parallel implementation of the technique, we simulate the behavior of an ideal, though unrealizable, multiple backtrace procedure. This procedure has the ability to correctly determine consistent objective values at fan-out stems without any special processing. Since it generates no conflicts during test generation, this multiple backtrace procedure is deterministic in the same sense as forward implication is. An initial objective implies a valid set of one or more values on the PI's of the circuit under test, and the backtrace procedure can never fail. Note that the definition of this procedure prevents consideration of test generation for redundant faults for which no valid test exists.
To supply the multiple backtrace procedure with the information necessary for implementing the above ideal behavior, a standard sequential PODEM algorithm is used to preprocess faults before they are submitted to the idealized parallel test generator. Redundant faults and faults for which no test is found due to excessive backtracking are discarded from further consideration. Thus the parallel test generation algorithm is only given faults for which a test is known to exist, and it has the ability to request what values need to be chosen for internal circuit lines when any doubt exists as to what value should be assigned. This capability is not needed during forward implication. During backwards justification it is only used so that the multiple backtrace procedure can always make consistent, valid assignments when fan-out stems are reached. During backward justification through fan-out free regions, the procedure operates just as a realistic multiple backtrace procedure would, with the same constraints.

To determine the amount of useful parallelism in a circuit partitioning approach that can actually contribute to a speedup over a reasonably efficient sequential implementation, the parallel operations are ordered so that circuit primitives are only processed once during each phase of test generation. During forward implication, for example, concurrent implications are evaluated in rank order, where the rank or level of an instance is the distance of its output from the circuit’s PI’s. This ensures that a circuit element is not evaluated until all changes on all of its inputs have been evaluated. Multiple backtrace operations are similarly ordered, based on distance from primary outputs. Within these constraints, the parallel test generator simulation evaluates as many circuit elements as possible in parallel during every time step, for both phases of test generation.
5.6. Results

To determine the amount of parallelism available during test generation with realistic circuits, the perfect test generator simulation was run on the 1985 ISCAS benchmark circuits [15]. Tests were generated for all single stuck-at gate-level faults identified as testable by a sequential PODEM algorithm with a backtrack limit of 10. No attempt was made to identify equivalent faults or reduce the fault list with fault collapsing or test compaction. Two configurations of a perfect parallel test generator were simulated, to determine the contribution to parallelism attributable to the multiple backtrace procedure. In the first, forward propagation was performed in parallel, but a single backtrace procedure similar to that used in the standard PODEM algorithm was used for backward justification so that no concurrency existed during backtracing. In the second, both the parallel multiple backtrace procedure and parallel simulation technique were employed.

Table 8 shows the measurements obtained for the configuration with a serial backtrace procedure. The "No. of Passes Per Test" column shows the average and maximum number of test generation passes required to generate tests for each circuit, corresponding to typical and worst case test generation effort. Figures for the average and maximum number of operations performed during both phases of test generation are shown for comparison with the multiple backtrace test generation configuration. The time step concurrency during backward justification is always equal to 1 for this case, since only a single circuit element is justified during each time step. The measurements of particular interest are the average and maximum figures for time step concurrency during forward propagation. The maximum figures, representing the best case parallelism attained during a single time step, are promising for the larger circuit sizes. However, the average time step concurrences, representing expected parallelism for a typical fault, are quite
Table 8. Single Backtrace Test Generation Results

<table>
<thead>
<tr>
<th>Circuit</th>
<th>No. of Passes Per Test</th>
<th>Justify Ops. Per Pass</th>
<th>Evaluate Ops. Per Pass</th>
<th>Time Step Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Max</td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td>c432</td>
<td>12.0</td>
<td>24</td>
<td>4.1</td>
<td>12</td>
</tr>
<tr>
<td>c499</td>
<td>36.9</td>
<td>41</td>
<td>6.4</td>
<td>11</td>
</tr>
<tr>
<td>c880</td>
<td>9.9</td>
<td>23</td>
<td>4.2</td>
<td>14</td>
</tr>
<tr>
<td>c1355</td>
<td>36.4</td>
<td>41</td>
<td>9.6</td>
<td>19</td>
</tr>
<tr>
<td>c1908</td>
<td>17.2</td>
<td>32</td>
<td>9.8</td>
<td>28</td>
</tr>
<tr>
<td>c2670</td>
<td>17.0</td>
<td>65</td>
<td>7.2</td>
<td>21</td>
</tr>
<tr>
<td>c3540</td>
<td>11.3</td>
<td>29</td>
<td>7.9</td>
<td>30</td>
</tr>
<tr>
<td>c5315</td>
<td>12.8</td>
<td>61</td>
<td>6.9</td>
<td>35</td>
</tr>
<tr>
<td>c6288</td>
<td>24.6</td>
<td>32</td>
<td>13.9</td>
<td>102</td>
</tr>
<tr>
<td>c7552</td>
<td>21.8</td>
<td>92</td>
<td>9.6</td>
<td>32</td>
</tr>
<tr>
<td>Circuit</td>
<td>No. of Passes Per Test</td>
<td>Avg</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>c432</td>
<td>4.0</td>
<td>12</td>
<td>12</td>
<td>1.9</td>
</tr>
<tr>
<td>c499</td>
<td>4.3</td>
<td>10</td>
<td>15.8</td>
<td>3.1</td>
</tr>
<tr>
<td>c880</td>
<td>4.1</td>
<td>11</td>
<td>4.2</td>
<td>13</td>
</tr>
<tr>
<td>c1355</td>
<td>5.2</td>
<td>13</td>
<td>28.5</td>
<td>3.5</td>
</tr>
<tr>
<td>c1908</td>
<td>5.2</td>
<td>14</td>
<td>20.2</td>
<td>2.3</td>
</tr>
<tr>
<td>c2670</td>
<td>4.3</td>
<td>13</td>
<td>16.1</td>
<td>2.4</td>
</tr>
<tr>
<td>c3540</td>
<td>4.2</td>
<td>13</td>
<td>11.5</td>
<td>2.7</td>
</tr>
<tr>
<td>c3315</td>
<td>4.6</td>
<td>12</td>
<td>11.4</td>
<td>2.7</td>
</tr>
<tr>
<td>c6288</td>
<td>11.5</td>
<td>31</td>
<td>26.2</td>
<td>7.4</td>
</tr>
<tr>
<td>c7552</td>
<td>4.7</td>
<td>31</td>
<td>26.2</td>
<td>744</td>
</tr>
</tbody>
</table>

Table 9. Multiple Backtrace Test Generation Results
low.

Table 9 shows the results obtained when the ideal parallel multiple backtrace procedure is used in addition to parallel logic implication. In this case, concurrency is taken advantage of during both phases of test generation, and multiple primary input assignments are possible during backtracing. The "No. of Passes Per Test" column shows that fewer test generation passes are needed both on the average and for the worst case faults, indicating that the multiple backtrace procedure increases the amount of useful work accomplished during each pass, contributing to an increase in efficiency. This observation is further demonstrated by noting the increased number of operations per pass for both justification and implication. The "PI's Assign. Per Pass" column in Table 9 shows the multiple backtrace procedure's ability to assign more than one primary input during each pass, generating more simulation activity and consequently higher parallelism during the forward propagation phase. Comparing the time step concurrency figures for the two phases of test generation shows that more parallelism exists in forward implication than in backtracing for all circuits except c499 and c1355 (c1355 is just c499 with Xor gates expanded to equivalent Nand configurations). Once again, however, the promising best case figures for parallelism are offset by poor figures for expected parallelism. For the circuit exhibiting the highest parallelism, c7552, expected concurrency for backward justification and forward propagation were only 2.3 and 6.4, respectively. These figures are surprisingly low considering the optimistic conditions assumed during measurement of the parallelism inherent in the perfect parallel test generator. It is interesting to note that parallelism does not always increase with increasing circuit size, suggesting that circuit topology is just as important a factor as circuit size.
5.7. Summary

This chapter presented an investigation of a circuit partitioning based, parallel processing approach to improving the performance of test generation. Measurements of the concurrency available in a number of circuits during both forward implication and backward justification were empirically determined by simulating the behavior of a perfect conflict-free test generator. The conditions under which concurrency was determined were chosen so that measurements would represent upper bounds.

Even with the optimistic assumptions used in this research, results indicate that the average available parallelism in a circuit partitioning approach to test generation is unexpectedly low. The average concurrency during the forward implication phase was generally found to be higher than that available during backward justification, but was at the most 6.4 for the largest circuit considered. Although the degree of parallelism is not monotonically increasing with circuit size, parallelism does tend to be highest for the largest circuits considered here. However, the rate of increasing parallelism as a function of circuit size is not promising.

The performance models derived in Section 5.4 show that the communication requirements introduced by using a circuit partitioning approach to test generation are substantial. Even with the optimistic assumptions used to formulate the performance models, the expressions for speedup derived with communication costs considered indicate that the influence of communication overhead on achievable speedup is considerable, making any performance improvement in a realistic distributed memory processing environment unlikely. The communication overhead
introduced by the circuit partitioning approach would likely cause parallel test generation to actually run slower than a standard serial test generation algorithm.

These results imply that the amount of parallelism in test generation that can potentially be exploited by a straight circuit partitioning approach is probably not high enough to justify the effort required for implementing such an approach, if achieving a large improvement in test generation performance is the primary goal. However, applying the technique using a small number of processors with a suitable cluster partitioning strategy that assigns multiple circuit primitives to each processor may be profitable. Even if such an approach fails to result in significant speedup, it still has one advantage over other parallel methods for test generation due to its handling of partitioned circuits and ability to process larger circuit sizes.
Chapter 6. Conclusions

6.1. Summary of Work

The work presented here complements and extends prior research in test generation by examining the use of an object-oriented approach for the design and implementation of a hierarchical test generation algorithm that supports the testing of circuits containing elements modeled with both structural hierarchy and multiple levels of abstraction. The implementation was developed using C++, with the objectives being to (1) produce a easily maintainable system, (2) provide an extensible framework for test generation supporting the straightforward incorporation of new types of circuit primitives and faults, and (3) retain the same level of computational efficiency that can be achieved using a procedural language such as C.

The object-oriented implementation of the modular, hierarchical test generation algorithm is extended to consider the accurate representation of bridging faults (BF's) in CMOS, and the use of $I_{DDQ}$ testing, an alternative technique for the detection of such faults made possible by the normally small supply currents in CMOS circuits. To allow for the direct representation of CMOS BF's, a switch-level circuit model is employed. By modeling circuits at the transistor- or switch-
level, the physical fault mechanisms present in CMOS circuits are represented by a general fault model incorporating both shorts and opens in addition to stuck-at faults. The necessary topological information is described using a graph-based circuit-model that corresponds directly to the circuit topology. The computational overhead imposed by the switch-level graph approach is counteracted by the use of hierarchy and multi-level modeling techniques. The bridging fault test generation algorithm presented here is a full ATPG system for IBDQ faults in CMOS circuits, and is able to perform deterministic test pattern generation, fault simulation, and test pattern compaction.

This research explores the feasibility of using a circuit partitioning approach to deal with large circuits and reduce the run-time complexity of test generation via parallel processing. Characterization of the major phases of test generation shows how the inherent parallelism existing in test generation can be exploited during forward implication and backward justification. The effectiveness of a circuit partitioning approach is evaluated with respect to its expected performance in a distributed processing environment. Performance models are derived for a limited class of circuits, and upper bounds on the concurrency available in specific circuits are empirically determined by simulating the behavior of an ideal parallel test generation algorithm that operates without backtracking. Results indicate that the average available parallelism is fairly low in typical circuits, limiting the potential speedup of a circuit partitioning approach to test generation.
6.2. Contributions of this Work

In the research involving object-oriented test generation and $I_{DDQ}$ testing of CMOS circuits, the following contributions are significant:

- The ability to accurately represent realistic combinational CMOS circuits described at the switch-level, and containing structures such as complex CMOS gates in addition to primitive logic gates.

- The incorporation of switch-level, gate-level, and higher-level models in a single modular, hierarchical test generation implementation to support both accurate and high-speed test algorithms for backwards justification, fault propagation, and forwards implication.

- The ability to generate $I_{DDQ}$ tests for realistic CMOS circuit bridging faults extracted directly from layout-level information.

- Deterministic testing is supported for unrestricted bridging faults in CMOS, including bridges affecting internal nodes of switch-level modules. Since an algorithmic and complete method is used for test generation, identification of untestable bridging faults is possible.

- A complete ATPG system for $I_{DDQ}$ testing of bridging faults, including a fast bridging fault simulator used for fault simulation and test compaction.

In the work dealing with circuit partitioning, this research has contributed the following original results:

- Performance models with closed-form solutions predicting the best-case speedups possible for test generation for a limited class of circuits in a distributed environment. These speedup results are expressed in terms of runtime and communication ratios.
• Upper bounds on the potential concurrency existing during test generation for a set of commonly used benchmark circuits. These figures were empirically determined.

6.3. Directions for Future Research

The flexibility and adaptability of the object-oriented test generation approach used in this research make it a rich topic for further exploration. Some of the following suggested topics for further research are natural extensions of this work, while others stem from recognized limitations of the current test generation implementation.

• Although the current test generation implementation assumes that modules can in general have multiple outputs, several critical routines currently rely on a single output assumption in the interest of efficiency. Removing this limitation would not be difficult, but would essentially preclude fair comparisons with existing gate-level test generation algorithms that use the single output assumption to great advantage.

• The support for derivation of new fault types provides a convenient means of adding new faults types to the test algorithm. Module-level stuck-at faults and bridging faults are currently supported; a general open fault model would perhaps be the next logical addition. It would most likely be necessary to increase the amount of state information maintained by modules and instances to support test generation for general open faults.

• Consideration of new fault types could also require the expansion of the five-valued logic system currently used. This could have a fairly dramatic impact on the complexity of individual module implementations, but the overall design of the test generation application should provide good support for such a change.
Due to limitations and "bugs" in the compilers used during this research [13, 31, 95], the use of multiple inheritance in forming class inheritance relationships was not possible. If the current implementation is to be enhanced significantly, it is likely that the use of multiple inheritance would become critical in preserving a clean interface between classes and maintaining the general extendibility of the program.

The true power of any hierarchical test generation design is only recognized with its application to circuits containing significant amounts of hierarchy. The lack of any commonly available set of benchmark circuits containing high-level hierarchy in a form usable in this research posed a problem in evaluating the performance of the implementation. The development of such a set of benchmark circuits would allow researchers in multi-level test generation to compare results in the same way that gate-level results have been compared using the ISCAS 85 benchmark circuits.

The current test generation implementation relies on compiled-in circuit descriptions of switch-level modules. A useful addition would be a parser that accepts text descriptions of switch-level modules and performs necessary pre-processing steps such as the assignment of signal flow directions in transistors.
Bibliography


Bibliography

164


Vita

Stuart Wayne Bollinger

April 1992

Personal

Born on May 30, 1966 in Richmond, Virginia.

Raised in Chesterfield, Virginia, graduated from Manchester High School in June, 1983.

Hobbies: reading, racquetball, piano and guitar, working on cars.

Education


Sep 1983 - Dec 1986 Virginia Polytechnic Institute & State University, Blacksburg, VA. B.S., Electrical Engineering. Summa Cum Laude.

Academic Honors and Awards

Presidential Graduate Fellowship, 1988.
Honor societies: Eta Kappa Nu, Phi Kappa Phi, Phi Eta Sigma.
Dean’s list every quarter.

Publications


S. Wayne Bollinger