APPROACHES TO TEST SET GENERATION

USING BINARY DECISION DIAGRAMS

A Thesis

by

JAMES WINGFIELD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2003

Major Subject: Computer Engineering

APPROACHES TO TEST SET GENERATION

USING BINARY DECISION DIAGRAMS

A Thesis

by

JAMES WINGFIELD

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

M. Ray Mercer (Chair of Committee)	A. L. Narasimha Reddy (Member)
Michael Grimaila (Member)	Chanan Singh (Head of Department)

December 2003

Major Subject: Computer Engineering

ABSTRACT

Approaches to Test Set Generation
Using Binary Decision Diagrams. (December 2003)
James Wingfield, B.S., Texas A&M University
Chair of Advisory Committee: Dr. M. Ray Mercer

This research pursues the use of powerful BDD-based functional circuit analysis to evaluate some approaches to test set generation. Functional representations of the circuit allow the measurement of information about faults that is not directly available through circuit simulation methods, such as probability of random detection and test-space overlap between faults. I have created a software tool that performs experiments to make such measurements and augments existing test generation strategies with this new information. Using this tool, I explored the relationship of fault model difficulty to test set length through fortuitous detection, and I experimented with the application of function-based methods to help reconcile the traditionally opposed goals of making test sets that are both smaller and more effective.

DEDICATION

To Frank and Shirley Wingfield, my first teachers.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

INTRODUCTION

Manufacture Testing

Since the invention of the integrated circuit in 1958, the microchip manufacturing industry has expanded to become a leading part of the world's economy. As in any successful business, manufacturers of integrated circuits must strive to satisfy their customers by delivering products that perform as specified. This policy is founded in the motivation of businesses to retain or increase their customer base in an effort to increase profit.

The process of manufacturing integrated circuits is very sensitive to disturbances from the manufacturing environment and variations in the materials used in the circuit. Such variations are not avoidable in practice, and they produce variations among different integrated circuits that are manufactured to achieve the same product. To deliver products that operate within the performance specifications promised to the customer, manufacturers must test the integrated circuits that they produce to ensure that variations do not cause the circuits to perform in an unacceptable fashion. Unacceptable products (also called 'parts') are labeled as defective, and the variations that cause the unacceptable performance are called defects.

For devices as complicated as integrated circuits, there could be many thousands of ways for a product to fail due to a defect in the circuit. Each defect that could occur in a digital integrated circuit has its own set of tests that can detect the defect. Manufacturers can detect all possible defects in a combinational circuit by applying all possible tests to the digital circuit; however, the test space size is exponentially related to the number of circuit inputs. For example, a circuit with as few as 20 inputs has more than a million

---

This thesis follows the style and format of *IEEE Transactions on Automatic Control*.

possible binary input combinations (2^{20}= 1,048,576). This means that the number of tests required to exhaust the test space does not scale well for circuits with more than a few inputs.

Limitations in time and equipment make exhaustive testing infeasible for manufacturers. Therefore, manufacturers must choose a subset of all possible tests to detect as many defects as they can with the finite testing resources they have. In an effort to develop algorithms to accomplish this task, much research has been done to explore the relationship of the space of all possible defects in a circuit and the space of all possible tests to detect the defects. The goal of this research area is to discover more information about the theoretical nature of defects and test set coverage.

Computational Problem

Choosing a set of tests that detect a number of given defects while minimizing the size of the test set requires knowledge of which tests detect which defects. Since integrated circuits are manufactured in a non-discreet domain (the real world), each location in a manufactured circuit can have an infinite number of possible variations from the intended specifications. Even if we define the term 'defect' to refer only to such variations that exceed an allowable threshold (change the digital value of a node, for example), there are many possible ways a circuit can be defective at each point in the physical circuit layout. This fact, combined with the complexity of physical layout parameters, makes the problem of determining which tests will detect each defect very difficult. To simplify the problem, some researchers use logical models of the circuit instead of physical layout information, along with logical models of defects called faults.

The use of such models simplifies circuit analysis by allowing calculations to be performed in the domain of logic functions. However, the mapping of physical defects to fault models is not one-to-one; there may be many fault models of various complexities to describe a particular defect. Nonetheless, simplifications are made to

reduce the computational requirement, and simple fault models are often used to create sets of test patterns.

Even if we work in the realm of logical circuit models and faults, the problem of producing minimized test set sizes is essentially a covering problem in two dimensions. If a matrix of test patterns vs. faults is created, with all possible test patterns listed as rows of the matrix, and all desired faults listed in columns, then a single bit could represent whether a given pattern detects a given fault at the intersection of the corresponding matrix row and column. This would yield complete information of the problem, but would require $O(m*2^n)$ space, with n = # inputs, and m = # faults. Even for small circuits this quickly grows beyond reasonable size; yet this is only the starting information for the real problem of choosing an optimal subset of patterns to detect all of the desired faults, which is NP-Hard [1].

TEST GENERATION

Regardless of the computational complexity, test patterns must be generated, so the problem has been approached in many ways. Test pattern sets are often generated according to the algorithm in Figure 1.

```
Create a Fault List
Begin Loop
        Choose a Target Fault from the Fault List
        Generate a Test Pattern for the Target Fault
        Simulate the Test Pattern to determine what faults are detected
Repeat Loop until all faults have been detected
```

Figure 1: Test Set Generation Algorithm

This algorithm is a framework of the test generation process. It does not realize an optimum solution to the test generation problem, but each part of this algorithm can be optimized in various ways to decrease the size of the test set that is produced.

Creating a Fault List

To create a fault list, the fault models of interest must be selected. As mentioned earlier, various fault types may be used to model physical defects; thus the choice of fault models to use in test generation is influenced by the type of defects that the user desires to detect. In addition, for a given circuit structure and fault type, some faults in the circuit will subsume other faults. This means that the fault list can be collapsed by subsumption, and some test generation applications will collapse fault lists to speed up the algorithm [2].

Fault Targeting Methods

Choosing a target fault in the test generation algorithm can also be optimized for better performance. The target fault may be chosen randomly from the faults that have not yet been detected, or the choice of target may be based on other information, such as whether the fault has been previously identified as a hard to detect fault. Another fault targeting strategy is to attempt to target a fault that is compatible with the test pattern chosen for the previously targeted fault. This is made possible by the fact that, for most circuits, many faults do not require fully-specified test patterns to detect the fault. This means that there will be some parts of the generated test pattern that don't have to be set to a particular value ("don't care" bits). These "don't care" bits might be assigned particular values such that the same test pattern is designed to target multiple compatible faults. This method is called dynamic compaction [3].

There is also a method to compact the test set after it has been produced. This can be done by examining which faults are detected by each pattern, and eliminating patterns that detect faults which are already caught by other patterns. This method is referred to as static compaction [4].

Due to the difference between physical defects and fault models, generating test sets to achieve multiple detections of each fault can yield test pattern sets that detect more defects [5]. This idea is practiced in most test generation software, and has come to be known as multi-detect testing.

Other approaches to generating test patterns have also been explored and published, including approaches that choose a test pattern first rather than targeting a specific fault, such as in [6].

Generating a Test for a Given Fault

Once a target fault is chosen, generating a test pattern to detect the fault can be done in many ways. The basic constraints of this sub-problem are that logic values can only be

assigned to the circuit inputs, and the assignment made must cause the circuit to enter a state that would be altered in an observable way if the fault were present in the circuit. Thus the generated test pattern must meet two conditions: fault excitation and fault observation. Excitation is the set of conditions that are required to cause the fault to produce an error in the circuit. Observation is the set of conditions that are required to allow that error to propagate to the circuit outputs so that the error may be observed.

For example, consider the stuck-at fault model, which is commonly used in test set generation for circuits. This model explains one effect of a defect that causes a node in the circuit to retain the same logic value, regardless of its stimulus. A stuck-at one fault for a particular circuit node would model the effect of that node exhibiting a value of logic one for any input combination. Of course, this fault would not cause any error in the circuit for input combinations that are supposed to set the node to a one, but it would cause an error for input combinations that are expected to produce a logic zero at the node. Thus, to excite the stuck-at one fault, the input values must be chosen to produce the erroneous state, in which a non-faulty circuit would expect a logic zero at the node. To observe the fault, the faulty node value must be propagated through the circuit to one of the circuit outputs. This will cause the outputs of a non-faulty circuit to differ from a faulty circuit, thus allowing the observation of a fault at that node. Only by meeting both the excitation requirement and the observation requirement can a fault be detected.

Many test generation tools use simulation-based methods to generate tests that meet the requirements for detecting a given fault. Such tools work with a gate structure of the circuit by assuming that a point in the gate structure must be set to a given value, then iterating backwards through the circuit toward the inputs, making assignments to nodes along the way to cause the assumed condition to be valid. For example, if the output of an AND gate is assumed to have a logic one value, then the gate inputs must all be set to logic one. However, if the output of an OR gate is assumed to have a logic one value, the only constraint is that at least one of the gate inputs must be a logic one. As the algorithm works toward the gate inputs, nodes are encountered that connect the inputs of

multiple gates together (these are called fan-outs). Such nodes may show that the algorithm has attempted to set the same node to different logic values, which is not possible and indicates a contradiction in some of the assignments made by the algorithm. Most circuits have fan-outs, thus the circuit-walking algorithms must allow backtracking to exercise alternate assignment options.

Test generation can also be performed using function-based analysis of the circuit. By calculating and storing a representation of the Boolean function at each node of the network, a test generation tool can produce the information necessary to evaluate the detection requirements of a fault model. The excitation requirement for a stuck-at one fault, for example, can be represented by the Boolean function that will yield a logic zero at the fault location. The Boolean functions can be thought of as a way to describe the subset of possible input assignment combinations that will satisfy some constraint. Thus, a Boolean function can also be formed to represent the set of input combinations that will satisfy observation requirements, and the excitation and observation functions can be combined with a Boolean AND operation (intersection operation of sets) to form a detection function. Function-based analysis methods have the advantage of evaluating total information about the faults, since the Boolean functions specify the entire set of input combinations meeting their respective criteria. Likewise, they have the disadvantages that come with working on such detailed information, including large data structure sizes and high computational requirements.

In commercial applications, the demanding requirements of function-based analyses make such methods impractical. This is why most test generation tools are simulation-based. However, using function-based analyses can provide insight into the nature of fault models and their relationship to test generation methods. It is for this reason that my research focuses on the application of function-based analysis to fault modeling and test generation.

BINARY DECISION DIAGRAMS

Previous research has developed a compact way to store and manipulate Boolean logic functions using directed acyclic graphs known as Binary Decision Diagrams (BDDs) [7] [8]. These diagrams have nodes connected by paths to represent the dependency of the logic function on various switching variables. For example, Figure 2 shows a simple BDD that represents the function F= (A*B) + C. The topmost node of the tree is called the root node. From the root node, a path can be followed to the bottom of the tree by choosing one of the two branches at each node along the path. The choice of which branch to follow is decided by the value of the variable that labels the node. For example, if we assume A=1, B=0, C=1, then we would follow the path indicated by the arrows in Figure 2, yielding a result of logic 1 for the function.

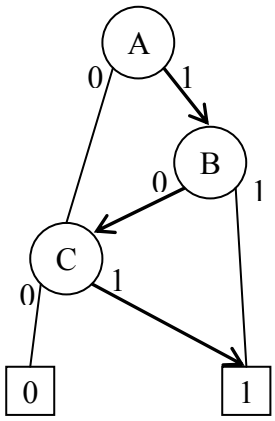

Figure 2: Binary Decision Diagram of (A*B) + C, Showing A=1, B=0, C=1

Space Complexity of BDDs

We can examine the space complexity of a BDD by dividing it into levels, as depicted in Figure 3. In the worst case, each level of nodes in a binary tree could be twice the size

of the level above it since each node has two branches. BDDs can have a level of nodes for each variable, so this leads to the initial estimate that a binary tree representation of a logic function might require $O(2^n)$ nodes where n is the number of variables. However, for BDDs that represent binary logic functions, there can be only 2 terminal nodes at the bottom of the tree (logic zero and logic one), thus the size of each level must reduce as the levels approach the terminal level, and the worst-case size will not actually reach the worst-case size of a full binary tree, though the order of space complexity may remain the same. Since the data structure size can grow exponentially as the number of variables increases, this method appears to be impractical for use on circuits where the number of variables is determined by the number of circuit inputs, and this number can grow beyond 40 for commercial circuits.

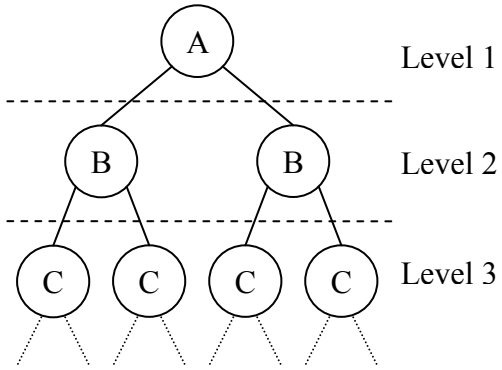

Figure 3: Binary Tree Depicting Levels

In practice, however, the sizes of BDDs used to represent functions at nodes in a circuit rarely approach the worst-case size. This is due to many reasons, including the fact that most functions do not depend on every variable on every path through the BDD. Consider again Figure 2. In this BDD, the path for A=0 does not contain a node for B, since it doesn't depend on the value of B to make a difference in the result of the function. We can describe this situation by saying that this path of the BDD is vacuous in variable B, and this happens whenever both branches from a node can point to the

same child node. For every vacuous (missing) variable, there is only one child directly below the vacuous variable rather than 2 separate children, so the size of the BDD is reduced by 2^k, where k is the number of levels below the vacuous variable. Fortunately this occurs often in practical applications.

Another way that the BDD structure lends itself to reducing its space requirement is by the reuse of sub-trees. If there is some branch that leads to a child node that has the same descendant structure as another node in the same BDD, there is no need to have both copies of the same sub-tree because all branches that point to the duplicated structure can point to the same node at the root of that structure. To demonstrate this idea, consider the BDD structure for an XOR function of 3 variables (A XOR B XOR C), shown in Figure 4. In this structure, the C nodes are reused twice each, reducing the space requirement by 2 nodes. The lattice pattern of this XOR structure continues for an arbitrary number of variables; such that the size of the XOR BDD is linearly related to the number of variables (1+2*n) rather than exponentially related as in the worst case space complexity.

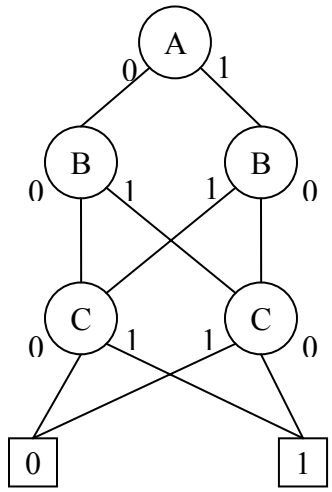

Figure 4: BDD for XOR Function

Algorithmic Benefits of BDDs

In addition, there are other benefits to the BDD structure that can reduce its space requirements in practice. An inherent advantage of the BDD structure is that the inverse of a function can easily be attained by simply swapping the logic zero and logic one terminal nodes. Thus, finding the inverse of an arbitrary BDD can be done in constant time.

Another useful task when working with functions is to calculate the number of variable assignment combinations (also called minterms) that lead to logic zero or logic one. This calculation can be performed by a simple algorithm that operates on the BDD structure. Each branch coming from a BDD node represents the assignment of the variable that labels that node. When an assignment to a variable is made, it reduces the number of possible remaining combinations by a factor of 2. Graphically, this can be illustrated by considering that half of the minterms at a given node will follow one of the node's branches, and the other half will follow the other branch. Thus, a simple recursive algorithm can be used to calculate the minterms that go to a logic one terminal node by starting at the root node with a count of the total number of possible minterms (2^n), and divide the count by 2 to determine the number of minterms at each of the child nodes. Since BDDs allow multiple branches to point to the same node (sub-tree reuse as described earlier), the algorithm must allow summing the minterm counts that come to a node from multiple paths. When the algorithm has finished working on all the nodes in the BDD, the minterm count at the terminal nodes can be recalled to yield the number of minterms for the function. By performing the counting operation with a breadth-first, non-repeating search of the tree, the minterm counts can be calculated in linear time (O(n) where n is the number of nodes). In the environment of function-based circuit analysis, minterm counting can be used to count the number of circuit input combinations that will satisfy the detection requirements for a fault.

To be useful in circuit analysis, there must be a way to perform logical operations between BDD structures, such as AND, OR, and XOR. These can be done with

reasonable efficiency according to the algorithms described in [8], as long as the BDD structure is in a canonical form. Reduced BDDs are canonical if they follow a constant ordering of variables in the structure levels. Such BDDs are called Ordered Binary Decision Diagrams (OBDDs). An additional benefit of canonical forms is that they allow easy comparisons between OBDDs to determine equivalence of functions.

FAULT MODEL DIFFICULTY AND TEST SET SIZE

Fortuitous Detection

Traditionally, test pattern sets are generated to detect nearly all stuck-at type faults in a circuit. For commercial circuits, the number of faults can grow beyond 100,000. However, the test pattern set that detects all of these faults is much smaller than the number of faults. In the test generation process of Figure 1, several faults may be chosen as target faults before the algorithm is complete, but since each test pattern that is generated detects more than just the targeted faults, it is not necessary to target every fault. These extra detections can be called fortuitous detections, since they happen consequentially. It is because of fortuitous detection that test pattern sets of reasonable size can be generated in the traditional simulation-based manner to detect large numbers of faults.

If the type of fault model is changed, but the same test generation procedure is used, the number of fortuitous detections may change and thus the length of the generated test set will change. Intuitively, if a fault model is more difficult to detect given a random test pattern, it is less likely that the faults will be fortuitously detected, and more test patterns will be required to detect all the faults. My first major experiment was to test this theory and investigate the importance of fortuitous detection. To accomplish this task, I examined test set sizes and the probability of detection using different fault models. The difficulty of detecting a fault given a random test pattern can be measured using a function-based circuit analysis tool, and the same tool can be used to generate test sets according to the single-target test generation process.

Detection Probability

Using a function-based circuit analysis tool, I computed the Boolean functions that represent the requirements for detection of stuck-at and transition type faults for several

benchmark circuits. As described earlier, these functions can be used to count the number of input assignment combinations (possible test patterns) that will detect the faults. By dividing this count by the total number of possible input combinations (2^n where n is the number of inputs), the probability of randomly detecting the fault can be computed. Since the set of test patterns that detect a fault is calculated as the intersection of the set of patterns that excite the fault and the set that will observe the fault, I also examined the probability of randomly exciting and randomly observing the faults.

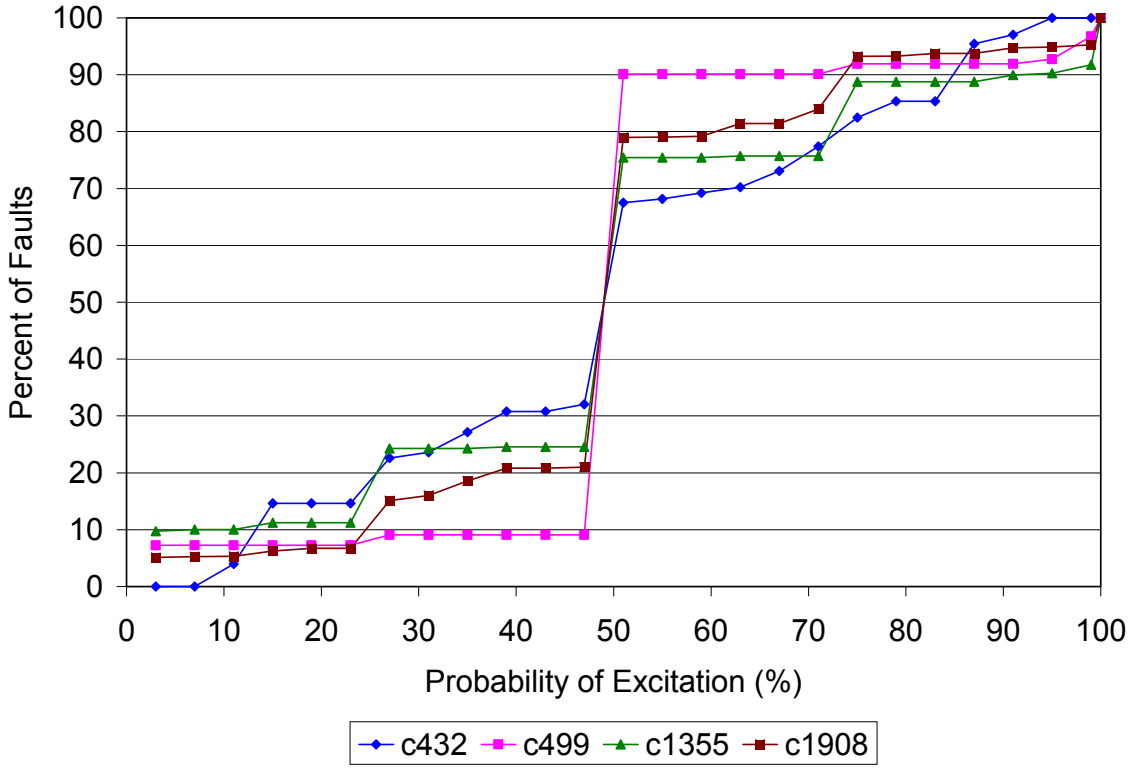

Figure 5: Cumulative Excitation Probability for Stuck-At Faults

The results of my computations for random excitation probability using the stuck-at fault model are shown in Figure 5 for four benchmark circuits: c432, c499, c1355, and c1908. This graph shows the cumulative percentage of faults with a given excitation probability.

To understand the meaning of this graph, consider the point where the probability of excitation is 19% and the percent of faults for c432 is about 15%. This means that 15% of the faults in c432 have a probability of random excitation that is less than or equal to 19%. Steep changes in the graph indicate a large number of faults that have nearly the same probability of excitation at that point.

One particular feature of this graph that is interesting is that there is a large change at the 50% probability mark, indicating that most of the stuck-at faults have a 50% probability of excitation. Since the excitation requirement for a stuck-at fault is the same as the inverted function at that fault's location in the circuit, this result indicates that, for these circuits, the statistical probability of a logic one or logic zero occurring at most nodes of the circuits is 50%.

Also notable is the fact that the graph is symmetric about the point (50%, 50%). This is expected, since if the excitation for a stuck-at one fault at a particular location is very probable, then the excitation for the stuck-at zero fault at the same location must be very improbable, forming complimentary pairs of data on which the graph is based.

Figure 6 shows the cumulative distribution of observation probability for stuck-at faults. Note that this graph is not symmetric and that most of the faults have a relatively low probability of observation. By comparing this graph with the excitation probabilities in Figure 5, it can be seen that, for most faults, observing the fault is less probable than exciting it. In other words, it is more difficult to observe stuck-at faults in these circuits than to excite them. By showing that there are fewer ways to observe a stuck-at fault than to excite it, this data lends definitive support to the work done in [9] in which the observation requirement is met first, and the easier excitation requirement is satisfied in multiple ways to produce more varied states in the circuit (which can detect more defects).

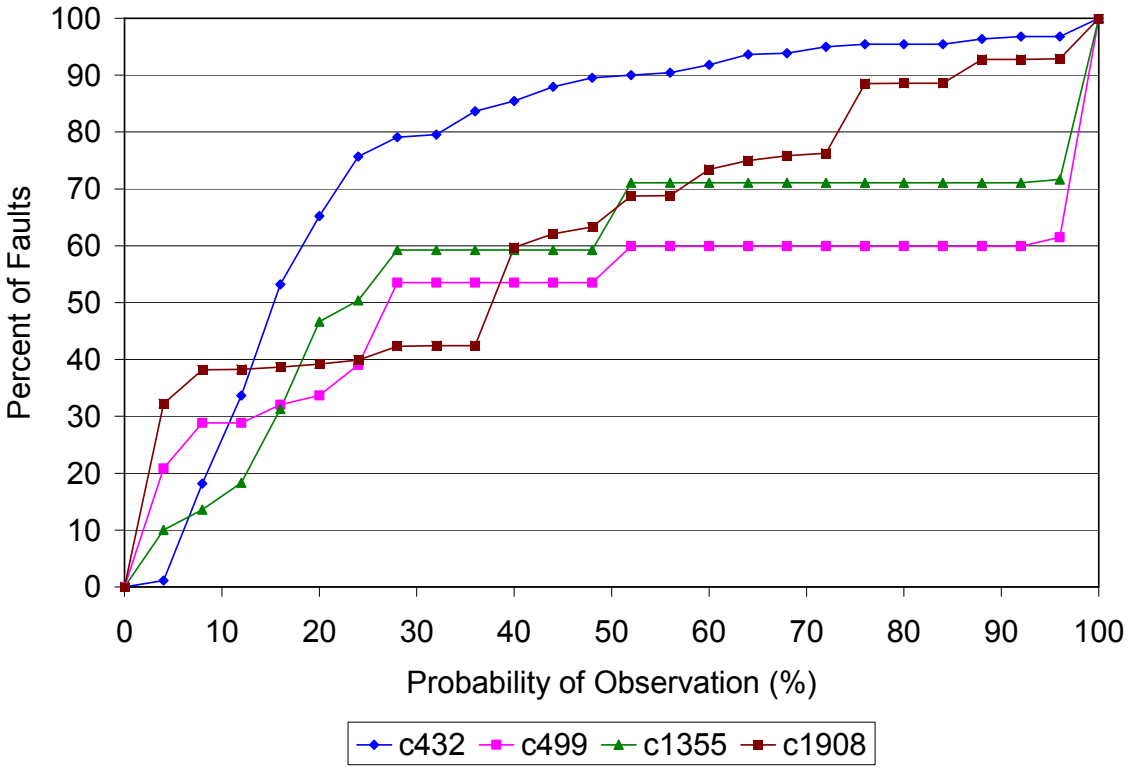

Figure 6: Cumulative Observation Probability for Stuck-At Faults

The combination of excitation and observation criteria yields the requirements for detecting faults. Figure 7 shows the cumulative distribution of random detection probabilities for stuck-at faults. As expected, the probability of detection is lower than excitation or observation alone for most faults, since both criteria have to be met at the same time. The graph reveals that nearly all of the faults for these circuits have at most a 50% probability of detection.

Similar data was measured for transition faults in the same circuits. Figure 8 shows the excitation probability distribution for transition faults, and the results show that this fault model is inherently harder to excite than the stuck-at model. The increase in excitation

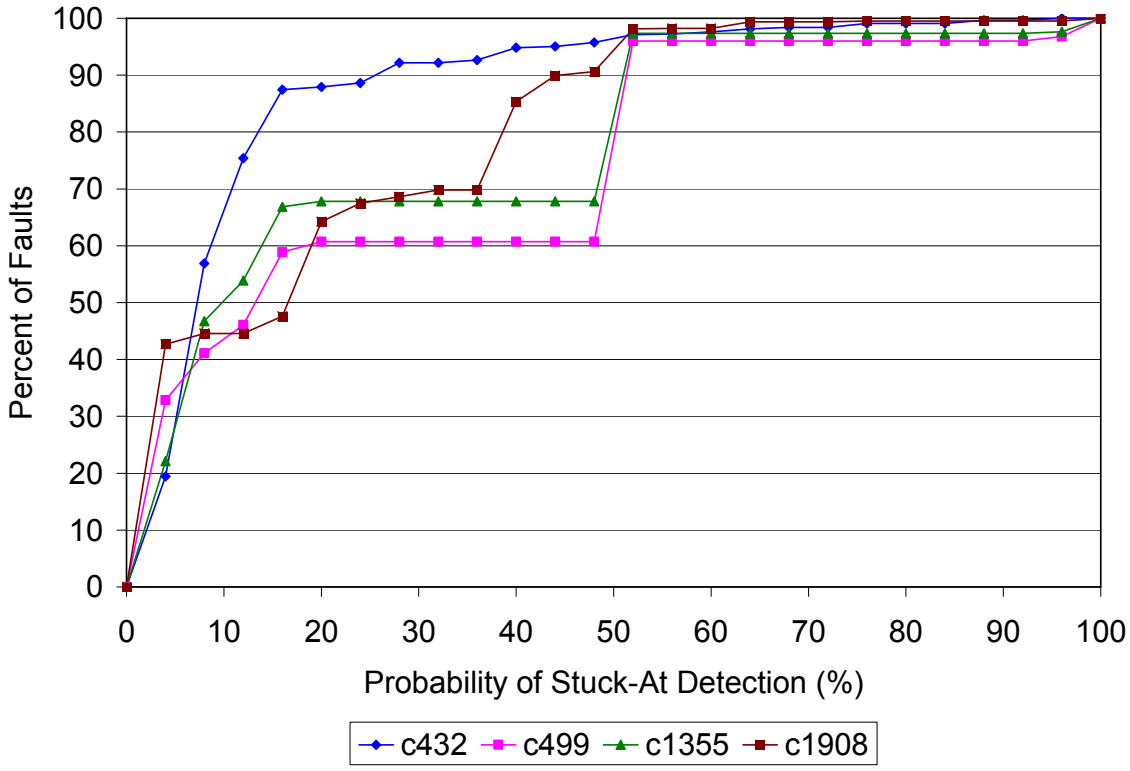

Figure 7: Cumulative Detection Probability for Stuck-At Faults

difficulty is due to the fact that exciting a transition fault requires setting the value of the faulty node to two opposite values in two consecutive test patterns. The pool of possible input combinations is therefore 2^{2n} since two patterns, each with n values, must be assigned. Because of this increase in possible combinations, there are no transition faults with a probability of excitation greater than 25%, which is half the excitation probability of most of the stuck-at faults. In Figure 9 we see that the difficulty of detecting transition faults is revealed, and most of the transition faults have a less than 10% chance of detection, given a randomly generated pattern. Since all of the transition faults have a probability of detection less than 25%, and most of the stuck-at faults have a detection probability less than 50%, we can estimate that transition faults are twice as hard to detect as stuck-at faults.

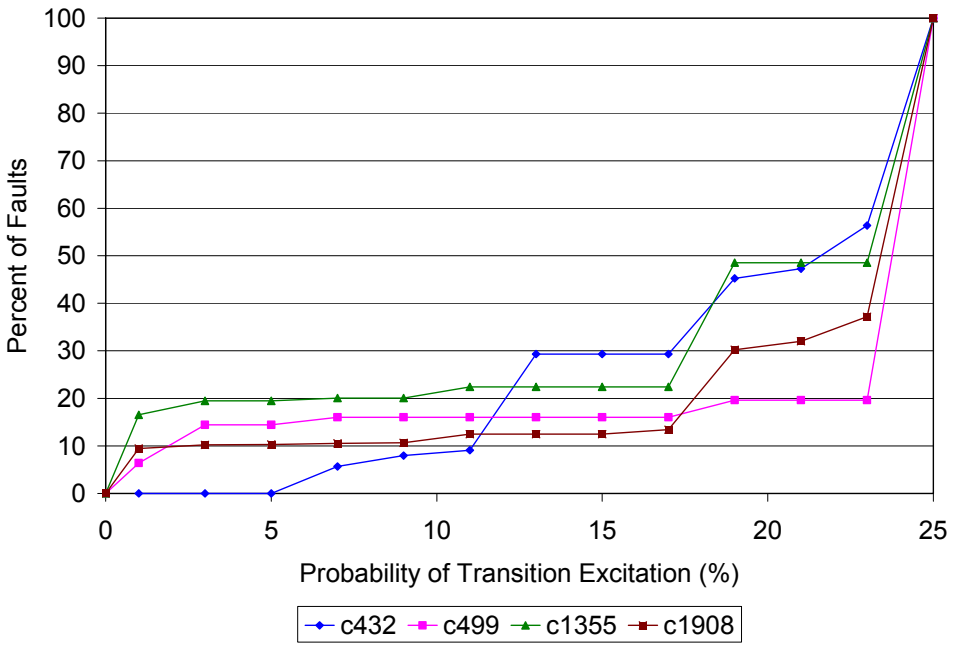

Figure 8: Cumulative Excitation Probability for Transition Faults

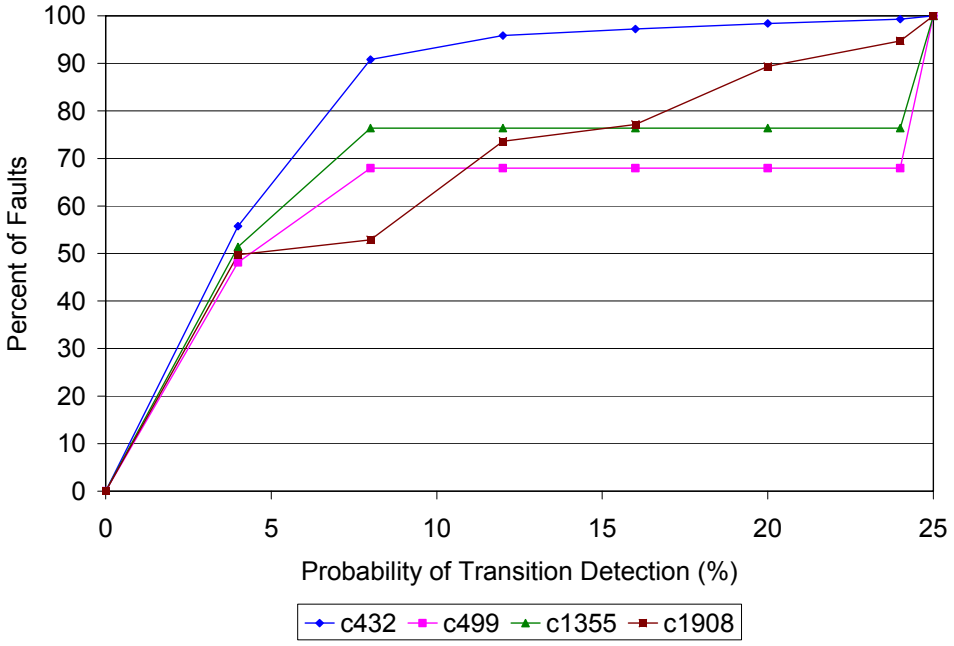

Figure 9: Cumulative Detection Probability for Transition Faults

Test Set Sizes

Since transition faults are more difficult to detect, we would expect that they would experience less fortuitous detection when generating test sets in the traditional way. Since traditional test set generation depends on fortuitous detection, this should lead to larger test sets for transition faults. To experiment with test set sizes, I augmented the function-based circuit analysis tool to generate tests according to the single-target method of Figure 1. In my experiment, I configured the tool to choose target faults at random from the remaining faults that have been detected least. Once a target fault is chosen, a test pattern for that fault is generated randomly by choosing from the pool of all patterns that will detect the target fault. Furthermore, I explored using this tool to generate test sets that detect each of the faults at least once, and test sets that detect the faults a given multiple of times. For the multi-detect test sets, I chose to use the multiple 15 based on the work of [10].

Table 1: Test Set Lengths for Stuck-At and Transition Fault Coverage

	1D SA	1D T	Ratio	15D SA	15D T	Ratio
c432	65	101	1.6	629	911	1.5
c499	80	143	1.8	982	1660	1.7
c1355	119	295	2.5	1601	3335	2.1
c1908	147	304	2.1	1748	3191	1.8

Table 1 shows the average number of test patterns in test sets generated using the single-target test generation method for both stuck-at and transition fault models in the four benchmark circuits. It should be noted that the size of transition test sets is given as the number of test pattern pairs in the set, since each transition test uses two test patterns. Stuck-at tests only require one pattern per test, so the size of stuck-at test sets is the same as the number of test patterns in the set. It is clear from this table that the test sets for

transition faults (1D T and 15D T in the table) are much larger than the test sets for stuck-at faults (1D SA and 15D SA in the table). Furthermore, if we consider the ratio of the size of transition test sets to stuck-at test sets, we find that transition sets are about twice as long, which corresponds to our estimate that transition faults are twice as hard to detect.

Interpretations

The results of this experiment show that transitions fault models are twice as hard, and require test pattern sets that are twice as long when the sets are generated in the traditional way. This result is due to the fact that the traditional method relies heavily on fortuitous detection, and harder fault models yield less fortuitous detection. The limited resources of manufacture testing prompt the goal of reducing test set sizes, yet harder fault models might model more types of defects than easier fault models. Therefore, relying on fortuitous detection is not a good strategy when using harder fault models to produce better test sets.

FUNCTION-BASED DYNAMIC COMPACTION

To reduce the dependency of test generation on fortuitous detection, I explored augmenting the information provided by function-based analysis into the dynamic compaction test generation method. As described earlier, traditional dynamic compaction affects the fault targeting step of the test generation process. Figure 10 illustrates the process of dynamic compaction.

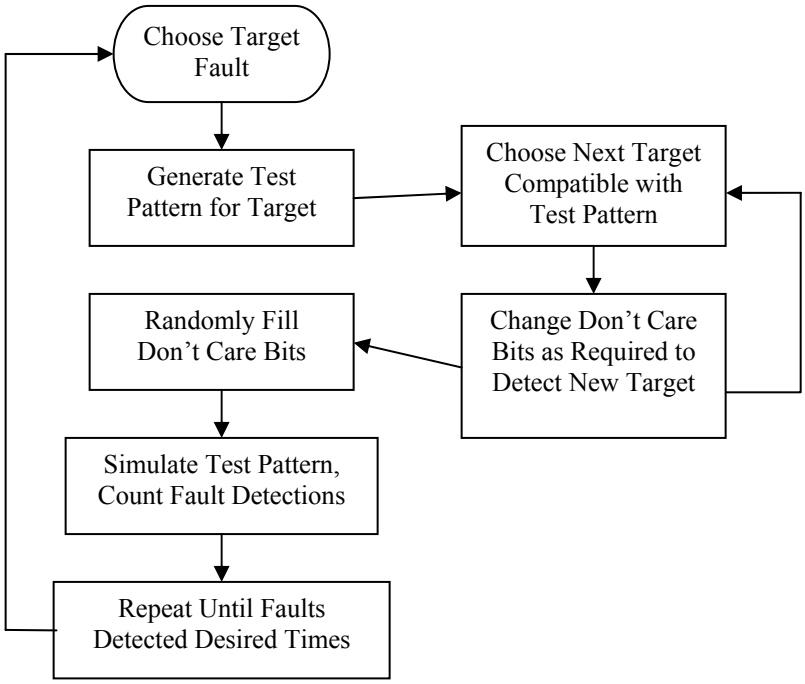

Figure 10: Pattern-Based Dynamic Compaction Algorithm

As shown in Figure 10, this process is usually done by choosing an initial target fault, and then generating a test pattern to detect that fault. During the test generation, however, input assignments that are not required to detect the fault are marked as "don't cares". If the input assignments that were made in the test generation are found to contribute to a test that will detect another fault in the fault list, then that second fault is

also targeted and any "don't care" assignments are filled in as necessary to detect that fault. This process is repeated until there are no "don't cares" remaining, or no further faults are compatible. This method makes good use of the "don't care" assignments of the fault that was targeted first, rather than assigning the "don't cares" randomly.

Once the dynamic compaction process is complete, any remaining "don't care" assignments are decided randomly to generate a complete test pattern. The test pattern is simulated through the circuit to count which other faults it detects (these are fortuitous detections), and the process will repeat to generate a set of test patterns that detect the desired faults a given number of times. I will refer to this method as pattern-based dynamic compaction, since the compaction is based on the compatibility between the chosen test pattern and potential target faults.

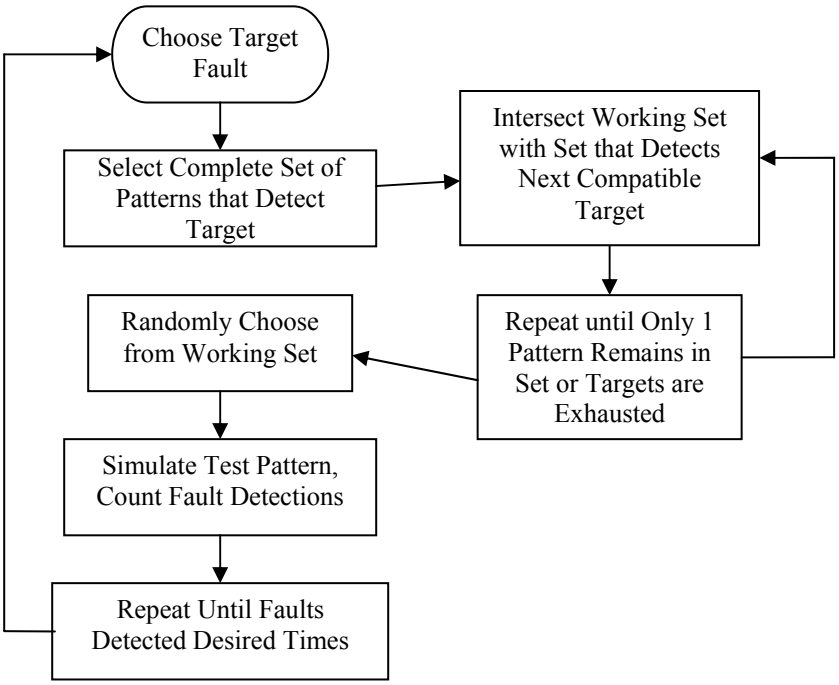

Figure 11: Function-Based Dynamic Compaction Algorithm

To improve this method of test generation, I introduced the use of functions so that the compaction can be based on all possible patterns that detect the target fault, rather than on the single pattern generated in pattern-based dynamic compaction. For my proposed method, I consider the BDD representation of the function that describes the detection requirements for the initial target fault as a set of all the test patterns that will detect that fault. I will call this set of test patterns the working set. Another fault is chosen, and the set of test patterns that will detect the new fault is intersected with the working set (using the logical operation AND on the functions) to produce a set of input combinations that detect both faults. If the resulting set is empty, this means that the faults are incompatible (no test pattern will detect both) and another fault is chosen to test for compatibility with the working set. If the intersection operation results in a set that is not empty, the resulting set becomes the new working set, and the second fault is added to the list of targets. The process is repeated until there is only one input combination remaining in the working set, or there are no faults left to test for compatibility. Finally, a test pattern is chosen randomly from the working set to add to the set of generated test patterns. As with other test generation methods, the chosen pattern is evaluated against the faults in the fault list to record which faults are detected by it, and the whole method is repeated to generate a set of tests that detect the faults a desired number of times. This method is illustrated in Figure 11. I will refer to the method as function-based dynamic compaction since the compaction relies on the compatibility of detection functions between faults.

Experimental Setup

To evaluate the performance of function-based dynamic compaction, I used a BDD-based test pattern generation tool. This tool takes a circuit description as input, calculates the logic functions for the nodes of the circuit, and uses these functions to compute the logic functions that specify the requirements for detecting particular types of faults. Using the detection functions, the tool can generate sets of test patterns using a single-target random pattern generation, pattern-based dynamic compaction, or function-

based dynamic compaction to achieve a desired minimum number of detections for each of the faults in the circuit.

I used two fault models for the experiment: stuck-at faults and transition faults. As described earlier, stuck-at faults are nodes in the circuit that retain a constant logic value regardless of the input pattern (stuck-at one or stuck-at zero). Stuck-at faults can be detected by a single test pattern that meets both the excitation and observation requirements for the fault, as mentioned earlier in this work. Transition faults are designed to model timing defects that might cause the logic value of a node in the circuit to change too slowly (slow to fall from 1 to 0, or slow to rise from 0 to 1). Detecting transition faults requires two test patterns. The first test pattern must excite the faulty node to its initial state, and the following pattern must detect whether the node remained stuck at that initial state.

Since increasing the number of detections of the least detected faults has been shown in a commercial experiment to produce more effective test sets [9], I decided to measure the performance of the random single-target method, pattern-based dynamic compaction, and function-based dynamic compaction when generating test sets that detect each fault both once and multiple times. For generation of multiple-detect test sets, I chose to produce sets that detect each fault 15 times. This number is derived from the work of [10] in which no escapes occurred at rated speed when a minimum of 15 detections per fault was used. Though no ideal minimum number of detections has been discovered for targeting transition test sets, I use 15 when targeting both stuck-at and transition faults for comparison.

Therefore, for each test generation method, I had two variables to set: the type of fault and the number of detections. As a result, for each method, I ran a total of four sets of experiments such that both of the two fault types we chose were coupled with both single- and multi-detection strategies. I selected four of the commonly-used ISCAS 85 benchmark circuits on which to perform these experiments: c432, c499, c1355, and c1908.

Results

Stuck-At Test Sets

Table 2 shows the test set lengths that result from running my experiment using each of the three test generation methods discussed earlier. Since single-target generation and pattern-based dynamic compaction depend on random selection of a test pattern, I executed the experiment 50 times for each of these methods to attain an average test set length. Minimum and maximum test set lengths are also given in Table 2 for these methods. Though function-based dynamic compaction uses random filling of "don't care" values, the selection of targeted faults follows the given order of our fault list, which is in gate order according to the circuit description. Because the faults are always in the same initial order when the experiment is run, the function-based dynamic compaction always produces test sets of the same size, so it is unnecessary to execute the function-based method multiple times.

It can be seen from Table 2 that pattern-based dynamic compaction improves on the average test set length of single-target generation as expected, and that function-based dynamic compaction produces by far the smallest test set sizes for both single detection and multiple detection test sets.

It is also notable to observe the ratio of average test set sizes between single- and multi-detect sets. This ratio is given as the last column in Table 2. A ratio of less than 15 indicates the method exhibits more compaction in the multi-detect sets than in the single-detect sets. For all of the circuits I tested, the functional-based method has the highest ratio, and the pattern-based method has the lowest for all except c1908. Since the single-detect average test set length and the ratio for the pattern-based method is lower than the corresponding values for the single-target method, pattern-based dynamic compaction must be doing a better job of compacting the multi-detect sets than single-target generation.

The function-based method does not compact the multi-detect set much more than the single-detect set, as evidenced by the higher ratios. However, the test set lengths achieved are near the theoretical minimums for single-detect test sets, which means it is not possible to compact the set much more than what is produced by the function-based method. The theoretical minimums are based on the results of [11] in which the maximum number of independent faults is computed and given as the minimum test set size.

Table 2: Test Set Lengths for Stuck-At Fault Targeting

		Stuck-At 1-Detect			Stuck-At 15-Detect			Ratio of Avg.
		Avg.	Min.	Max.	Avg.	Min.	Max.	
c432	Single-Target	65	55	72	614	595	643	9.4
	Pattern-Based	56	51	64	487	477	496	8.7
	Function-Based	32	32	32	409	409	409	12.8
	Theoretical Min	27						
c499	Single-Target	78	67	93	1003	981	1029	12.9
	Pattern-Based	71	62	78	825	814	840	11.6
	Function-Based	53	53	53	781	781	781	14.7
	Theoretical Min	52						
c1355	Single-Target	133	126	142	1701	1678	1723	12.8
	Pattern-Based	128	118	143	1366	1352	1382	10.7
	Function-Based	85	85	85	1261	1261	1261	14.8
	Theoretical Min	84						
c1908	Single-Target	155	142	167	1818	1793	1839	11.7
	Pattern-Based	131	125	138	1700	1687	1714	13.0
	Function-Based	110	110	110	1594	1594	1594	14.5
	Theoretical Min	106						

Transition Test Sets

The results of my experiments targeting transition faults are shown in Table 3. As with stuck-at fault targeting, I executed the test generation 50 times for the single-target and pattern-based methods and calculated average test lengths. The functional-based method produces identical test lengths for the same reason as mentioned earlier.

Table 3 shows that the pattern-based method yields much smaller test sets than the single-target generation, and the function-based method surpasses both. The improvement in performance between single-target and pattern-based compaction is much greater than what was shown for stuck-at fault targeting. This is evidence of the impact of fortuitous detection on test set sizes, as discussed in the previous section of this thesis. Transition faults are harder to randomly detect than stuck-at faults, which leads to fewer fortuitous detections of non-targeted faults in the single-target method. Since the pattern-based and function-based methods do not rely solely on fortuitous detection to produce the desired number of detections per fault, these two methods perform much better than single-target generation when a harder fault model is used.

These results also show that function-based dynamic compaction is able to detect all of the transition faults using a test set that is only 7 patterns longer than the stuck-at test set in the case of c432, and only 3 patterns longer in the case of c499. This is notable, since transition faults are, on average, twice as hard to randomly detect as stuck-at faults (demonstrated in previous section). It is important to keep in mind that each transition test pattern is actually a pair of tests: an initialization (or setup) pattern followed by a detection pattern. In spite of this, such a small increase in the test set length for a fault model that is twice as difficult to detect lends support to the hope that more complex fault models might be targeted without drastic increases in test set size.

Table 3: Test Set Lengths for Transition Fault Targeting

		Transition 1-Detect			Transition 15-Detect			Ratio of Avg.
		Avg.	Min.	Max.	Avg.	Min.	Max.	
c432	Single-Target	102	94	113	882	851	906	8.6
	Pattern-Based	66	58	74	571	552	597	8.7
	Function-Based	39	39	39	413	413	413	10.6
c499	Single-Target	148	137	169	1637	1600	1668	11.1
	Pattern-Based	76	69	83	847	836	866	11.1
	Function-Based	56	56	56	785	785	785	14.0
c1355	Single-Target	333	317	354	3450	3413	3503	10.4
	Pattern-Based	214	200	230	2278	2242	2334	10.6
	Function-Based	120	120	120	1744	1744	1744	14.5
c1908	Single-Target	311	296	330	3182	3130	3234	10.2
	Pattern-Based	166	160	175	1916	1894	1937	11.5
	Function-Based	130	130	130	1735	1735	1735	13.3

For this fault model the pattern-based approach does not show a better (lower) compaction ratio as compared to the single-target method, which means that these two methods are about equally as good at compacting multiple-detect sets over single-detect sets. For the easier stuck-at model, the pattern-based method had a lower ratio. As with the stuck-at targeting, the function-based method has the highest ratios, which is (as before) due to having such short single-detect test pattern sets.

Interpretations

The Problem of Computational Effort

The dynamic compaction methods that I discussed involve greater computational effort than the single-target method. The pattern-based method is not prohibitively difficult to implement practically, and it is already included in standard industry tools. Function-based dynamic compaction requires a test generation tool to operate in the functional domain of the circuit, and thus requires much more computational work to implement

than pattern-based methods. I was able to use my function-based tool on comparatively small combinational benchmark circuits, but the additional amount of effort required is prohibitive for use on commercial-size sequential circuits. There is an effort to investigate ways to reduce the computational complexity and time required to execute operations involving BDD representations of functions. This may lead to the feasibility of performing function-based analysis, including the function-based dynamic compaction that we propose, on commercial circuits in the future. For the present, my results are useful only for further discovery of the nature of test pattern generation and fault models.

The Problem of Test Set Sizes

Considering the problem of test set sizes, my experiment shows that function-based dynamic compaction performed much better than single-target and pattern-based methods in every case that was tested. Based on these results, I predict that function-based dynamic compaction would also show outstanding results if applied to commercial circuits. The experiment also demonstrates that attempts to reduce test set size when harder fault models are used may have greater success if the methods rely less on fortuitous detection and more on deterministic targeting. If the barrier of computational effort is relieved, function-based dynamic compaction could allow the use of more complex fault models that have more difficult detection criteria without significantly increasing test set length.

sByDDer

To perform the experiments mentioned earlier, I have developed a software application to manipulate BDDs for the purposes of test generation and fault analysis. The application is called sByDDer, as a partial acronym for Binary Decision Diagram. Due to the complete information provided by function-based circuit analysis, sByDDer has become a platform for running experiments to test the ideas of many people in my research group. I have served as primary software architect for the sByDDer project, and have managed its distribution and use by the research group.

History of Development

The sByDDer application is based on a simple BDD tool originally developed by Li-C. Wang. The original tool was used circa 1990-1995, when the limits of computational power made it difficult to calculate just the detection functions for the small benchmark circuits (c432) in less than a couple days of time. Now that computational power has expanded by many factors of magnitude, the small circuits are usable and additional functions or ideas can be applied to them, such as the ideas and results listed earlier in this work.

The original BDD tool was written in the C programming language and included the capability to calculate the Boolean function at each node of a circuit, as well as the observation function at each node. These functions could be combined to form detection functions, and to produce pseudo-random test patterns. I began with this base tool that had been developed several years ago, and then I corrected operational bugs that were discovered and expanded the tools capabilities to allow new types of experiments. One of the improvements made to the tool was the addition of a minterm counting algorithm to count the number of input combinations that would excite, observe, or detect a fault. I also expanded the tool to enable multi-detect test generation, and I incorporated both stuck-at and transition fault models into the application. Further expansion included the

capability to incorporate pattern-based and function-based dynamic compaction as described in an earlier section. The resulting sByDDer tool could produce multi-detect test sets (randomly or by dynamic compaction) for stuck-at or transition fault models on a given circuit, along with fault dictionaries for the test patterns produced and a matrix of pair-wise fault compatibilities.

One improvement I made to the BDD processing was the introduction of a compacted BDD structure. Every BDD that is stored in the application is compared to an existing stored BDD structure to find any common substructures. Instead of storing each new BDD separately, the common substructures are combined, such that common structures are never duplicated in memory. This reduces the amount of memory required to hold the BDDs. It also greatly reduces the amount of time required to count the minterms of the BDDs (used for fault difficulty and random pattern generation). Once the minterms of a common substructure have been calculated, the minterm counts are stored with the common structure and they never need to be recalculated. This means that instead of calculating minterm counts over all nodes of every BDD, the application calculates only over uniquely-structured nodes of the BDDs.

Latest Version

The many augmentations made to sByDDer over the years resulted in a very complex set of programming code, with only a small portion of the original tool's code remaining intact. Since the application works on problems with great time and space requirements, the code was designed to be very efficient for the computer. The C programming language was originally chosen for the project because of the detailed control that it grants to a programmer to make the program more efficient. Code that is efficient for a computer is often very difficult for a human to understand, edit, and debug, thus augmentations to the project took an increasing amount of time to complete as the application became more complex. In addition, the complexity of the tool made it difficult for anyone except the designers to edit it for experimenting with new ideas.

What we had in the end was a tool that performed its intended functions well, but was difficult to improve and play with for testing new ideas. In a research environment, where rapid prototyping is vital to measuring the value of a new idea, such applications do not usually survive long.

After taking classes on the subjects of software design and smarter tree-based algorithms, I thought of new ways to organize the program and its data structures by utilizing object-oriented programming practices and the data-hiding concepts that such practices are based on [12]. To fit sByDDer into the new organization would require a complete rebuild of the program from scratch; but the benefit of such reorganization would be much time saved when new ideas are implemented with the tool. In my judgment, the benefits outweighed the cost of redesigning such a complex tool from scratch, even if the new tool was less efficient. I chose to rewrite sByDDer using C++, and coded in a way to make it cross-platform compatible, rather than optimized for a particular computer architecture.

The resulting application is called sByDDer version 5.0, and has been tested on small circuits. It is modularly designed to be extensible without much further effort, allowing new fault models or new test generation algorithms to be integrated without requiring knowledge of the complete operation of the program. The application is divided into three major components: circuit objects, BDD objects, and fault objects.

The circuit objects provide data structures to store information about the gates in a logic circuit and how they are connected. I have written functions within the circuit objects that allow reading circuit descriptions written in the former sByDDer input file format, and it would be a simple process to add functions to read from other useful formats, such as Verilog or Bench. The circuit elements (gates) are created in a way that allows easy identification of locations in the circuit by user-definable names for nodes, and the capability to differentiate between branches of a fan-out network.

The BDD objects include all of the storage structure and algorithms for manipulating the BDD representation of logic functions. The implementation details are hidden from

other parts of the program to allow simple manipulation of functions using easy-to-understand operators such as +, *, !, and ^ of the operations OR, AND, NOT, and XOR, respectively. The BDD object is configured to use two terminal nodes (logic one and logic zero) in BDD trees, but it is flexible enough to easily allow the addition of other types of terminal nodes, such as X or X*, which may be useful in future experimentation.

Fault objects are coded to include a random test generation algorithm, and allow the modular addition of other test generation methods. New fault models can also be easily added by creating a new fault object based on the fault object template that is built-in to the program.

The newest sByDDer application is now a fully-modular object-oriented system that can be easily expanded or included in future applications. It will meet the rapid prototyping needs of other researchers in our area, and will be easy to learn by future participants in the computer engineering group.

SUMMARY AND CONCLUSIONS

There are two primary goals in generating tests for manufacture testing of integrated circuits: to generate tests that detect as many physical defects as possible, and to generate compact test sets that fit within tester resources. Since there are many ways that a physical defect can occur in a circuit, certain effects of defects are modeled by fault models. More complicated fault models may describe physical defects with greater accuracy, but they also have a smaller probability of random detection. My experiments show that such fault models are less likely to be detected fortuitously, thus targeting them with traditional test generation methods produces larger test sets, and pits the two goals of test generation against each other. Based on this knowledge, test generation methods that rely on fortuitous detection are not a good strategy to meet both goals.

By using function-based circuit analysis, complete information about the faults may be augmented into the traditional test generation process, such as with the function-based dynamic compaction method that I have proposed. This method has produced test sets that are near the theoretical minimum size for some of the ISCAS85 benchmark circuits. However, the computational effort required to perform function-based analysis makes this method infeasible when applied to commercial-size circuits using current computational power. Future increases in the efficiency of function-based analysis or increases in computational resources may enable the use of such methods in commercial environments. For now, this approach remains useful to researchers studying the concepts of fault modeling, test generation methodology, and designing circuits for testability.

The BDD-based functional circuit analysis tool that I have developed to perform these experiments will be useful for further research into new theories of the nature of fault models and test generation for digital integrated circuits. The newest version of this tool is designed with modularity and ease of use in mind, so that it can be used as a rapid prototyping tool to quickly evaluate new ideas. The tool can also be easily expanded to

incorporate additional functionality, or it can be integrated into other tools to put the information of function-based analysis to use in other ways.

REFERENCES

[1] B. Krishnamurthy and S. B. Akers, “On the complexity of estimating the size of a test set,” *IEEE Transactions on Computing*, vol. C-33, pp. 750-753, 1984.

[2] A. Prasad, V. D. Agrawal, and M. V. Atre, “A new algorithm for global fault collapsing into equivalence and dominance sets,” in *Proceedings IEEE International Test Conference*, 2002, pp. 391-397.

[3] P. Goel and B. C. Rosales, “Test generation and dynamic compaction of tests,” in *Digest of Papers Test Conference*, 1979, pp. 189-192.

[4] I. Pomeranz, L. N. Reddy, and S. M. Reddy, “COMPACTEST: a method to generate compact test sets for combinational circuits,” in *Proceedings International Test Conference*, 1991, pp. 194-203.

[5] J. Dworak, J. Wicker, S. Lee, M. R. Grimaila, K. M. Butler, B. Steward, L-C. Wang, and M. R. Mercer, “Defect oriented testing and defective part level prediction,” *IEEE Design and Test of Computers*, vol. 18, no.1, pp. 31-41, 2001.

[6] S. Lee, B. Cobb, J. Dworak, M. R. Grimaila, and M. R. Mercer, "A new ATPG algorithm to limit test set size and achieve multiple detections of all faults," in *Proceedings Design Automation and Test In Europe*, 2002, pp. 94 – 99.

[7] C. Y. Lee, “Representation of switching circuits by binary decision programs,” *Bell System Technology Journal*, vol. 38, no. 4, pp 985-999, 1959.

[8] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers*, vol. C-15, no. 8, pp 677-691, 1986.

[9] M. R. Grimaila, S. Lee, J. Dworak, K. M. Butler, B. Stewart, H. Balachandran, B. Houchins, V. Mathur, J. Park, L. C. Wang, and M. R. Mercer, “REDO – random

excitation and deterministic observation – first commercial experiment," in *Proceedings VLSI Test Symposium*, 1999, pp. 268-274.

[10] S. Ma, P. France, and E. J. McCluskey, "An experimental chip to evaluate test techniques: experimental results," in *Proceedings International Test Conference*, 1995, pp. 663-672.

[11] I. Hamzaoglu, and J. H. Patel, "Test set compaction algorithms for combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 8, pp. 957-963, 2000.

[12] B. Stroustrup, "What is object-oriented programming?" *IEEE Software*, vol. 5, no. 3, pp. 10-20, 1988.

VITA

Contact Information:

James Wingfield, B.S. Electrical Engineering
TAMU Computer Engineering Group, MS 3259
College Station, TX 77843-3259
jwingfield@ece.tamu.edu

Professional Interests:

- Computer Design and Testability
- Design Verification
- Simulation
- Automatic Test Pattern Generation
- E-Commerce Application Development
- Internet Application Security

Education:

- Texas A&M University, M.S., December 2003
- Texas A&M University, B.S., December 2001

Professional Experience:

- Co-Owner, Web Application Developer, PureStudio Productions, 2000-Present
- System Administrator, Research Assistant, Computer Engineering Group, Texas A&M University, 2001-Present
- Network Administrator, Lead Web Application Developer, Conference Management Services, Inc., 2000-2002

Consulting:

- PureStudio Productions, 2000-Present
- Mercer & Associates, Streets and Steele, 2003

Professional Societies:

Member, Institute of Electrical and Electronics Engineers (IEEE)

Professional Society Presentations:

- "ATPG Depends on Fortuitous Detection" (with J. Dworak, B. Cobb, S. Lee, Li-C Wang, and M. R. Mercer) International Symposium on Defect and Fault Tolerance in VLSI Systems, Vancouver, Canada, November 6-8, 2002
- "Introduction and Analysis of a Novel Test Generation Approach: Function-Based Dynamic Compaction" (with J. Dworak and M. R. Mercer) International Symposium on Defect and Fault Tolerance in VLSI Systems, Cambridge, Massachusetts, November 3-5, 2003