2023 Girl Hackathon Ideathon Round: Solution Submission

Project Name: Test Pattern Generation for Fault detection in circuits

Participant Name: Vedica Mishra

Participant Email ID: vedica21101@iiitnr.edu.in

ReadMe File Links (Eg Github) https://github.com/vediicaa/ATPG-GirlHackathon/tree/main
https://github.com/vediicaa/ATPG-GirlHackathon/tree/main#readme

Brief summary

The problem aims to design an algorithm for **fault detection in digital circuits** to identify faults at specific nodes. The algorithm should be efficient, able to exhaustively search the input space, and accurately activate and detect faults.

The provided solution implements an Automatic Test Pattern Generation (ATPG) algorithm **influenced by the D algorithm approach**. It generates test vectors, evaluates the circuit's logic, and **compares the observed and expected outputs** to detect faults. The code utilizes a queue-based approach, fault activation, and output comparison techniques to systematically explore the input space and identify faults in the digital circuit.

Problem Statement

The solution addresses the need for efficient testing of manufactured chips in the semiconductor industry. During the chip manufacturing process, faults or defects may occur at certain nodes or locations within the circuitry. These faults can negatively impact the performance, reliability, and functionality of the chips if not detected and rectified before delivery to end users.

The goal is to design an algorithm and corresponding code to identify these faults by generating appropriate test vectors. The provided code implements an algorithm to accomplish this task. The algorithm proposed is mainly influenced by the D algorithm which is known for its effectiveness in generating test vectors to identify single stuck-at faults, which are a common type of structural faults in digital circuits.

By employing the ATPG algorithm, the code helps semiconductor manufacturers ensure the quality and reliability of their chips. It enables them to systematically test and detect faults, allowing for necessary rectifications before delivering the chips to end users. This helps prevent potential malfunctions, failures, or safety hazards that may arise due to undetected faults in the circuitry.

The approach used to generate the algorithm.

The algorithm used is mainly influenced by the D algorithm. The proposed solution explores the circuits's logic by propagating the fault effects through Boolean equations and identifies the patterns that activate the faults. By

analyzing the circuit's structure and logic gates, the algorithm effectively targets specific nodes and evaluates their behavior under different test vectors, allowing for efficient fault detection and diagnosis.

The primary focus of the algorithm is to check out all possible combinations of input one by one and trigger the faulty node, if the faulty node is not triggered (produces the output opposite of its fault), we move onto the next combination vector without checking the value of "Z" thus saving time. Once the faulty node has been triggered we move to its outputs with the same fault to reach the final output "Z". If the final output "Z" produced doesn't match with the output of the non faulty circuit, our fault is detected and we save the test vector generated and break from the loops. Otherwise we continue searching.

The algorithm uses the idea of backtracking in the D algorithm to find the sensitive input and their paths and if the fault has been sensitized move forward in the path to test the final output.

The following functionalities have been used by the algorithm:

1. ***fillqueue():*** It uses a queue Data Structure and is filled with all possible combinations of test vectors that should be tested.
2. ***combinations():*** This function is used to initialize the primary input [A, B, C, D] with different boolean values. Here a map data structure is used to store the values of the nodes.
3. ***isOperator() and getOperator():*** These functions are used while parsing the input files. isOperator returns if the received token is an operator or not. getOperator returns the operator received in a char data type.
4. ***evaluatingCircuit():*** This is the most important function of the program as it evaluates each node of the circuit from the primary inputs and updates its value in the map. The fault nodes are also taken care of in this function. If the fault is not detected we move back to the main function to check another test vector.
5. ***extractingLine():*** This function is used for parsing the circuit file and extracting each token. It further calls *evaluatingCircuits()* to simulate the circuits.
6. ***main():*** This is the main function which runs primarily it reads the input file, calls the above mentioned functionalities and stores the output in the output.txt file.

Proof of Correctness.

1. The algorithm prompts the user to specify the fault location ('Fault_at") and fault type ('Fault_type'). It modifies the values of the corresponding nodes (stored in a map Data Structure "mp" and "mp1") to simulate fault accurately. The map "mp" is used to modify the node's value according to the fault and "mp1" is used to retrieve the original values in case the fault is not detected. Hence **the precise fault activation** allows the algorithm to detect the fault's effects on the output.

2. The algorithm explores all possible 16 test vectors until the right vector is found and exhaustively searches for the faults in the digital circuit. This is supported by the "fillqueue()" function which supplies all the test vectors and pops it one by one. Hence **leaving no untested possibilities** and **maximizing the chances of detecting faults.**

3. The algorithm **terminates** after a finite number of iterations. It either reports the corresponding test vector or determines that no test vector can identify the specified fault. This ensures that the algorithm **does not run indefinitely** and provides timely results.

Complexity Analysis

1. **Time Complexity:** The solution uses approximately **O(16*L*N)** in the worst time where L is the total number of lines in the input file and N is the total number of tokens. Hence we can observe that no extra time is being taken by the algorithm.This is because the algorithm manipulates the circuits while parsing it.

2. **Space Complexity:** The solution used **O(2*n)** for storing values of all the nodes, **O(4*16)** for storing all possible test vectors and an extra **O(N)** for storing all the nodes and operators in the string vector.

Alternatives considered

There are various advanced algorithms like PODEM and FAN that are widely used in the industries.

- **The PODEM algorithm** is the path oriented algorithm which uses advanced techniques like backtracking, finding an objective function and updating d frontier continuously. It would require the use of advanced data structures like graphs or BDTs(Binary decision trees) to implement the same. The algorithm is efficient and good for complex circuits with multiple sensitizable paths. However as mentioned in the problem statement, the circuits for testing are not complex and consist of 4 inputs and 1 output. Hence the use of complex data structures is unnecessary and would require more space and time. Therefore the use of maps is the best as it accesses the values in O(1) complexity and requires a space of O(N) where N is the total number of nodes in the circuit.

- The second alternative to the problem is the **FAN algorithm**. It is the forward-activation backtrack algorithm which again requires complex implementation which also increases the chances of errors and bugs. This algorithm explores multiple paths and hence will need to store additional information, increasing memory usage. This can be a disadvantage for resource-constrained environments and since the circuits are not complex, lesser space usage also provides perfect results. However for complicated circuits FAN would have been the best algorithm.

References and appendices

[1] T. Kirkland and M. R. Mercer, "Algorithms for automatic test-pattern generation," in IEEE Design & Test of Computers, vol. 5, no. 3, pp. 43-55, June 1988, doi: 10.1109/54.7962.

[2] Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," in IEEE Transactions on Computers, vol. C-30, no. 3, pp. 215-222, March 1981, doi: 10.1109/TC.1981.1675757.

[3] SM.Thamarai , Dr K.Kuppusamy, Dr T.Meyyappan, Fault Detection and Test Minimization Methods for Combinational Circuits – ASurvey.ISSN:2231-2803.