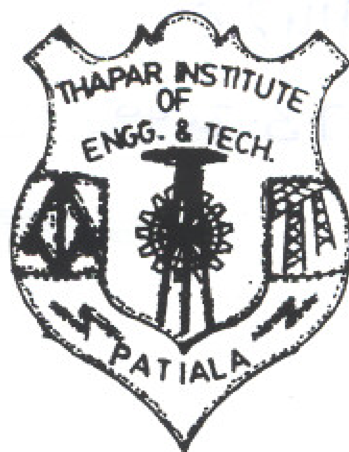
VLSI TEST GENERATION USING OBJECT ORIENTED APPROACH

A THESIS

Submitted in partial fulfilment of the requirement for the Award of the Degree of

MASTER OF ENGINEERING
(COMPUTER SCIENCE)



Submitted

By

KAILASH CHANDER BHARDWAJ

Department of Computer Science & Engineering
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(Deemed University)
Patiala-147 001

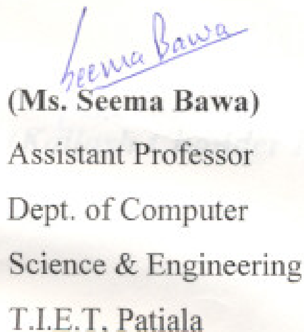
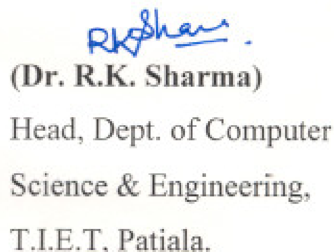
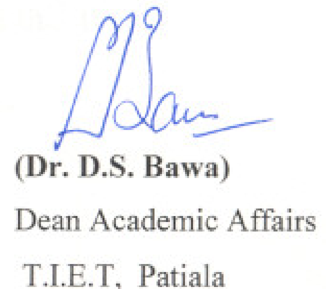
CANDIDATE'S DECLARATION & CERTIFICATE

I hereby certify that work, which is presented in this thesis entitled " VLSI TEST GENERATION USING OBJECT ORIENTED APPROACH" in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science, and being submitted in the Department of Computer Science & Engineering. Thapar Institute of Engineering & Technology (Deemed University), Patiala is an authentic record of my own work carried out under the ***supervision of Ms. Seema Bawa.***

The matter presented in this Thesis has not been submitted in part or full for the award of any other degree of this or any other university.

(Kailash Chander Bhardwaj)

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

(Ms. Seema Bawa)	**(Dr. R.K. Sharma)**	**(Dr. D.S. Bawa)**
Assistant Professor	Head, Dept. of Computer	Dean Academic Affairs
Dept. of Computer	Science & Engineering,	T.I.E.T, Patiala
Science & Engineering	T.I.E.T, Patiala.	
T.I.E.T, Patiala		

The Viva-Voce Examination was held on ______________________________

(External Examination)

ACKNOWLEDGMENTS

It gives me pleasure to express my sincere thanks and gratitude to my guide **Ms. Seema Bawa**, Assistant Professor, Department of Computer Science & Engineering, Thapar Institute of Engineering & Technology Patiala, for her valuable guidance, painstaking efforts and help in every respect for completion of this thesis. Her encouragement and interest in discussions have been highly beneficial.

I am grateful to **Dr. P.S. Bimbhra**, Professor & Head, Electrical & Electronics Engineering Department, Thapar Institute of Engineering & Technology, Patiala for providing me necessary help and facilities.

I am thankful to all my friends and colleagues for providing useful suggestions.

Last, but not the least, I express my profound gratitude to my family members for their constant encouragement to carry out this thesis.

Kailash chander

(Kailash Chander Bhardwaj)

TABLE OF CONTENTS

	ABSTRACT	
1.	Introduction	1
2.	Test Generation	4
3.	Algorithms	
	3.1 D-Algorithm	11
	3.2 PODEM Algorithm	18
	3.3 FAN Algorithm	24
4.	Parallel Processing Techniques	44
	4.1 Fault Partitioning	45
	4.2 Heuristic Parallelization	47
	4.3 Search-Space Partitioning	49
	4.4 Functional Partitioning	50
	4.5 Topological Partitioning	51
5.	Object Modeling Technique (OMT)	53
6.	Object Model of ATPG	67
7.	Conclusion & Future Scope of Work	75
8.	References	76

ABSTRACT

The report presents a CAD Tool for Automatic Test Pattern Generation (ATPG). The problem of Test generation for VLSI circuits can be characterized as the search of N-dimensional 0-1 state space, where N is the number of primary inputs in the circuit and is known to be NP-complete. The basic concepts common to most test generation algorithms, such as implication, sensitization, justification, implicit enumeration, and backtracking are explained in detail. Various automatic test generation algorithms –D-algorithm, PODEM, FAN are discussed and compared.

ATPG algorithm uses various heuristics, which play a major role to speed up the ATPG process as they prune the search space effectively.

To further accelerate the Test pattern generation process, various parallel processing algorithms are suggested.

Object oriented approach has been used to analyze and design the ATPG process, leading the inherent benefits of approach like reusability of design and code, easier maintenance, portability etc.

The core of the testing tool, the FAN algorithm is implemented using the object oriented approach. Object oriented analysis and design of FAN is presented in detail.

1. INTRODUCTION

With the advancement in the IC technology the complexity of the circuits increased. Very Large–Scale Integration (VLSI) is the fabrication of thousands of components and interconnection at once by a common set of manufacturing steps. Its advantages are reduced system cost, better performance, and greater reliability. These advantages would be lost unless VLSI devices can be tested economically. Due to increased complexity, testing of VLSI circuits has become difficult as it drastically reduced the controllability and observability of the logic on the chip.

Manufacturing of a product consists of fabrication and testing. Design and test development precede manufacture. The *correctness* of fabricated device is determined through testing. Testing is done to discover the defects in the digital system. The defect can be caused either during manufacture time or because of wear-out in the field. During production the testing is done at various stages: t*he dies are tested during fabrication, the packaged chips before insertion into boards, the boards after assembly, and the entire system is tested when complete*. The choice of test patterns for each of these tests is determined by many factors such as time available for test, the degree of access to internal circuitry, the percentage of failures, which are required to be tested.

The cost of testing a system has become a major component in the cost of designing, manufacturing, and maintaining a system. The cost of testing reflects many factors such as TG (Test Generation) cost, testing time, ATE (Automatic Test Equipment) cost etc. It is somehow ironic that a $10 μP may need a tester thousand times more expensive.

A test for a fault is an input (a vector or a sequence of vectors) that will produce different outputs in presence and absence of fault; they make the fault observable at the primary output. In combinational circuits, a single input vector can test a specified stuck fault. Testing a fault in the sequential circuit is more complex.

The goal of test vector generation algorithms is to generate test patterns at an affordable cost. Test vector generation can be automatic. A pre-requisite for automatic test-generation is existence of an algorithm that can be programmed. ATPG methods have a very high fault coverage for single stuck faults. Designers often use methods that are not fully automated for generation of functional test vectors. Functional tests are used for verifying the correctness of the design. They are developed manually by the designer based on the understanding of the functions implemented in the circuit.

The class of test generation algorithms for combinational circuits are partitioned into two groups. One group consists of algebraic algorithms in which test generation for a given stuck at fault is done by exploiting some algebraic representation of the fault-free and faulty circuit. The other group, called structural algorithms, generates test vectors by exploiting the gate level representation. First group is not very practical because heuristics required to tolerate NP-completeness of the test generation problems are not available.

Among the structural algorithms, D-algorithm is the oldest. But this is highly inefficient for ECAT (error correction and translation) circuits with X-OR gates.

PODEM algorithm treats test vector generation as a branch and bound problem. It is an improvement over D-algorithm. FAN (fan–out oriented test generation) uses sophisticated strategies of unique sensitization and multiple backtrace to further accelerate the test pattern generation process. SOCRATES test pattern generation system enhances the FAN algorithm by more efficient implication and sensitization procedures.

I have designed the FAN algorithm using the Object Oriented Analysis and design techniques and divided the test generation problem into independent modules (objects). With a slight modification in the design these modules can implement any of the test generation algorithms.

Object orientation means thinking in terms of objects as compared to thinking in terms of functionality. It is a new way of thinking about problems using models organized around real world concepts.

The notation of objects provides a solid base for specifying, designing and implementing a software.

The essence of object-oriented development is the identification and organization of application domain concepts rather than their final representation in a programming language.

There are many benefits of object-oriented development:

- Helping specifiers, developers and customers express their abstract concepts clearly and communicate them to each other.
- More clearly understanding problems
- Communicating with application experts, modeling enterprises
- Data and its behavior is tightly coupled
- Increases reusability of design and code
- Leads to smaller systems
- Less implementation time
- Leads to easier maintenance

2. Test Generation

The basic testing problem is to determine an optimal testing procedure for a circuit C and a set F of faults which models most of the failures likely to occur in C. The testing procedure consists of three steps: test generation, test application, and test verification. Testing procedure optimality is in terms of the time and effort required to carry out all three steps.

Given C and F, the test generation step generates randomly and/or algorithmically a set T(F) of input vectors called test patterns or test vectors. Each test pattern when applied to C will cause it to produce incorrect logic values (errors) at one or more output lines of C in the presence of certain specific faults belonging to F. For T(F) to be a complete test set for F, it is essential that for every fault f ε F, there must exist at least one test vector in T(F) which will detect f. Once a T(F) is generated, It is applied to circuit C through test equipment, which converts each test vector into corresponding physical signals, and applies it at the appropriate pins of c with the necessary timing constraints. The test equipment is also used to collect the response generated by the test vector application. In the final step, the test responses are analyzed to determine whether the circuit C is faulty or not. This is referred to as fault detection.

2.1 Approaches to Test Generation

Since VLSI circuit are characterized by their enormous complexity, low-cost methods such as exhaustive testing and random test generation are promising candidates for this application. Other methods, commonly known as algorithmic, have high complexity but are effective for combinational circuits.

Exhaustive Test Generation

When the number of primary inputs are small, application of all possible input vectors ensures 100 percent fault coverage. Such vectors are easily generated by hardware or software, and may be applied quickly at electronic speed.

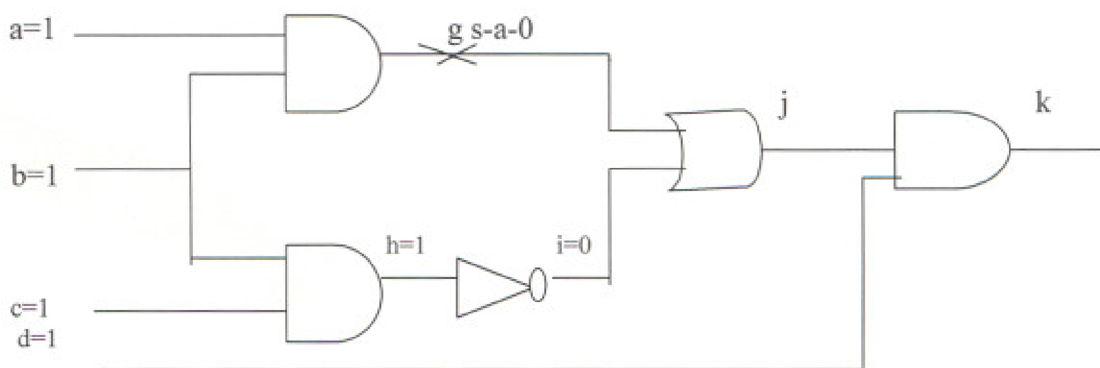
Exhaustive techniques can be applied to more complex circuits by portioning of the logic into smaller sub-circuits. Unfortunately, the logic partitioning problem is intractable, and hence, cannot be fully automated. In exhaustive generation, the response stream is compressed into a compact signature before comparison with the reference signature.

Random Test Generation

Random test generation, like the exhaustive method, depends only on the number of circuit inputs but is otherwise quit independent of the circuit topology. In the random method, tests are generated through a random process. Once the tests have been generated, there is nothing random about them. The method, if used properly, can provide good results at a very low-cost. A fault simulator is generally useful for effective test generation by this method

2.2 Path sensitization algorithms

The central idea of a whole class of test generation algorithms is the concept of path sensitization. Path sensitization is a process of generating a path along which the fault effect is propagated i.e. creating a sensitized path. For a fault to be detected, the test vector must sensitize one or more paths from fault site to the circuit output. While algorithmic test generation methods based on path sensitization vary substantially in details, they all share some fundamental operations as given below:



(Fig. 2.1)

Fault Excitation: This involves establishing a signal value at the fault site which is opposite to that produced by the fault.

Fault effect propagation: The objective of this operation is to move the fault effect closer to a primary output. In the fig. 2.1 shown above to propagate the effect of the fault g s-a-0 to the line j it is necessary to set line i to 0.

Line value justification: This is done to specify the internal line values that must be produced by defining one or more primary inputs, such that they are consistent with already defined values in the circuit. In the fig. 2.1 for making i=0, h is to be made 1 which in turn requires c to be made 1 and b is already made 1.

Line value implication: All the preceding steps are carried out in incremental steps and involve specifying one or more line values. The effect of such specification may ripple through in the forward direction by implication.

The D-frontier:

The D-frontier consists of all goals whose output value is currently x but have one or more error signal (i.e. D's or $\bar{D}$'s) on their inputs. *Error propagation* consists of selecting one gate from the D-frontier and assigning values to the unspecified gate inputs so that the gate output becomes D or $\bar{D}$. This procedure is also referred to as the D-drive operation . If D-frontier becomes empty during the execution of the algorithm, Then no error can be propagated to a PO. This on empty D-frontier shows that backtracking should occur.

The J-frontier:

To keep track of the currently unsolved line justification problems, we use a set called the J-frontier, which consists of all gates whose output value is known but is not implied by its input values.

Unique Sensitization Process:

When the D frontier consists of a single gate, the unique sensitization procedure is applied. In such a situation, often specific paths exists such that every path from the site of the D-frontier to the PO goes through those paths. In the figure (2.3), every path from the gate G2 to the primary output passed through the paths F-H and K-M. In order, to propagate the value D or D-bar to the PO path, F-H and K-M should be sensitized. This partial sensitization which is uniquely determined is called unique sensitization.

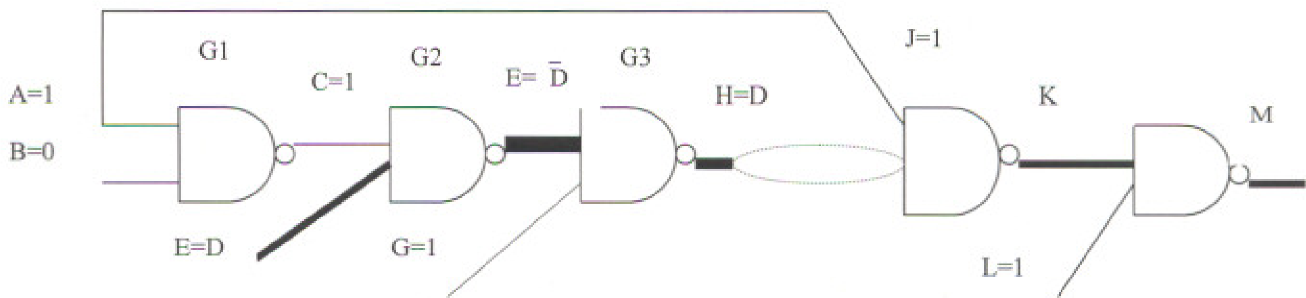


Fig. (2.3)

(Unique sensitization and Implication)

After the implication of this assignment, we get A=1, B=0, F= $\bar{D}$ and H=D without backtracks. Thus, by avoiding no. of backtracks unique sensitization procedure improves the efficiency of the algorithm.

2.2 Fault modeling

Faults occur in VLSI circuits throughout their life. The different causes of faults in VLSI circuits are:

- Design error
- Material defects
- Extremes in operational environment
- Deterioration due to length of operation
- Aging

The large number and complex nature of physical failures dictates that a practical approach to testing should avoid working directly with the physical failures. These physical failures should de described at a higher level (logic, register transfer, functional. Etc.). These description of faults is called a fault model. If the fault model accurately describes all the physical failures of interest, then one only needs to derive tests to detect all the faults in the fault model.

Fault models serve two purposes. First, they help in generating tests, and, second, they help evaluate test quality defined in terms of coverage of faults. A higher level fault can include many physical failures, thus reducing the number of primitive entities to be considered in deriving a test. Even if the details of the physical failure mechanisms are not known, a fault model can be hypothesized to cover most of the possible failures.

2.2.1 Gate-level Faults

The classical gate-level fault model is the stuck-at fault model, where the effects of the physical failures are supposed to be described by the inputs and outputs of the logic gates permanently stuck at logic 0 or 1. This model has the advantage of being independent of the technology being used. Analysis of TTL gates showed that some failures can be modeled by single stuck faults on the lines of the gate; others can be detected even though they cannot be modeled as stuck faults. In most of the cases, stuck-at fault model represents the physical defects satisfactorily.

2.2.2 Functional- level Faults

At the functional level, several types of faults can be simulated. They include stuck-at type faults at the inputs and outputs of the element and the faults that effect the behavior of the functional element.

The faults that effect the behavior of the functional element are called functional faults. They include state –variable stuck faults and control faults. A control fault is defined as a fault that causes the control variable or expression to have an incorrect value and affects conditional transfers in CHDL descriptions. By testing for control faults in addition to stuck faults, true fault coverage is likely to improve. The general function faults comprises of faults which causes changes in the expression used in evaluation of functional element but their usefulness is limited by the need for describing each such fault individually.

2.3 N-P Completeness of Test Generation

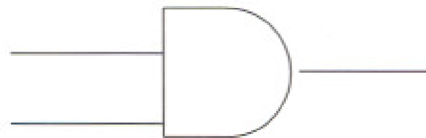
This concept is introduced to prove that the amount of time, that is, the number of steps required to solve some specific problems is beyond a certain practical limit. A problem that can be solved in a polynomial time on a non -deterministic machine is referred to as an N-P problem. Those N-P problems that are not known to have ant deterministic polynomial solution, a special sub class is referred to as the N-P complete class.

Theoretical study by Ibarra and Sahni has shown that test generation for combinational circuits is a N-P complete problems, strongly suggesting that no test generation algorithm with a polynomial time complexity is likely to exist. This property of test generation necessitates that various heuristics be developed to create practical solutions to the problem.

3.1 D-algorithm:

The first algorithm for ATPG that was proved complete is the D–algorithm introduced by Roth in 1966. The D-algorithm includes a notation and a calculus with which a single stuck-at fault can be detected at a node in the circuit and propagated to a primary output of the circuit.

This algorithm uses a five–valued logic, which consists of the logic values 0 and 1, an unknown value X, and two additional values D and $\bar{D}$. A D value signifies a logic value of 1 in the good circuit and 0 in the faulty circuit (0/1). A $\bar{D}$ value signifies a value of 0 in the good circuit and 1 in the faulty circuit (1/0). Each gate in the circuit has two D-cubes associated with it, the *primitive D-cube of a fault (pdcf)* and a *propagation D-cube (pdc).* A pdcf is the set of inputs that produces an error signal on the output of that gate if it contains a fault. A pdc specifies the input values necessary to propagate an error signal on an input of a gate to the output. Figure below shows the pdcf's and pdc's of a two input AND gate.



pdcf	pdc
1 1 D	1 D D D D D
1 0 $\bar{D}$	D 1 D 1 $\bar{D}$· $\bar{D}$
1 0 $\bar{D}$	D 1 $\bar{D}$

(fig. 3.1.1)

The D-algorithm's basic operation is the repeated intersection of the D-cubes necessary to perform the tasks required to test for a specific fault. These tasks consists of three operations: *fault sensitization, fault propagation, fault justification.*

Fault sensitization is the process by which a circuit node is made to produce an erroneous value as a result of the fault. Sensitization is accomplished by specifying an input

combination for the circuit element containing the fault, using the pdcf 's that cause the output to take on the appropriate D value. Fault propagation is the process of propagating the D values to the primary outputs so that they can be observed.

The list of circuit elements closest to the primary outputs that have a D or a $\overline{D}$ on the output is called the D frontier. The objective of fault propagation is to advance the D frontier to the primary outputs. This process sensitizes all possible paths from the fault site to the primary outputs. This multiple path sensitization is necessary for the D – algorithm to guarantee completeness.

During fault sensitization and fault propagation, certain circuit nodes are required to take on specific values. Establishing this value or goal by placing values on the primary inputs is called justification. The primary inputs that can be used to justify a goal are usually determined by backtracking through the circuit topology from the node in question to the primary inputs. A value is chosen for one of these inputs, and a forward simulation-like process, called *forward implication,* is performed to see if this assignment is consistent with satisfying the goal. If not, a different goal is chosen and the process is repeated. A test is finally generated when the fault is sensitized, a path for the fault to be observed at the primary outputs is sensitized, and all of the goals are justified.

D-algorithm example:

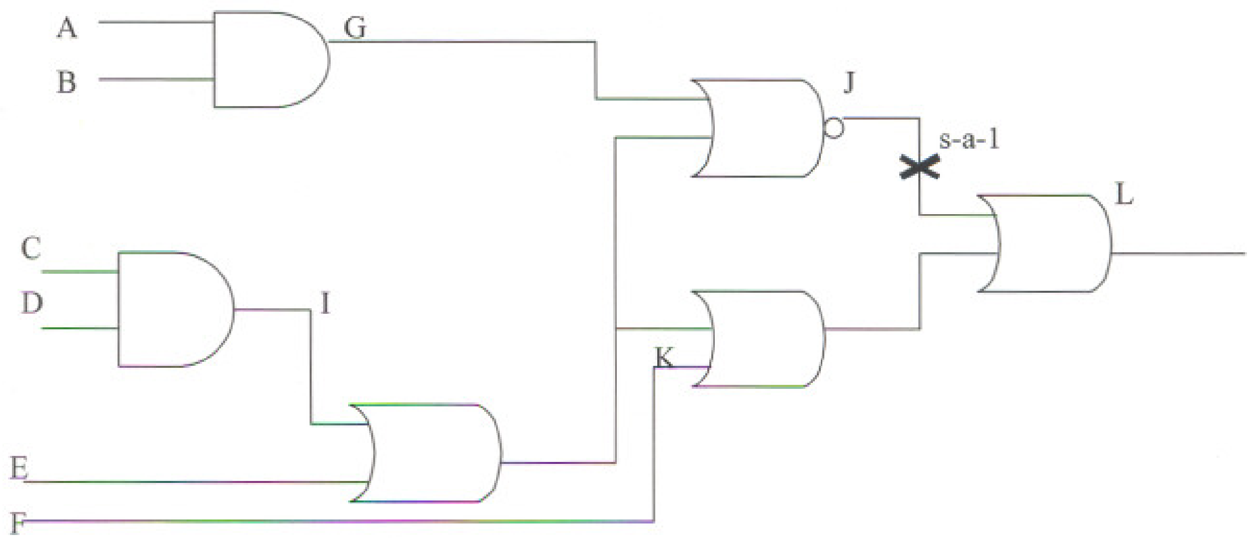


Fig. 3.1.2

Circuit under test

The ECAT Problem

The first ECAT type circuit encountered was an implementation of a single error correcting/double error detecting scheme over a set of 64 data bits and 8 parity bits. Error detection schemes involve the use of EXCLUSIVE –OR (XOR) trees to determine parties over various combinations of data/parity bits. The resulting parity signals are further combined to generate signals that indicate error or help correct error bits. Basically, the ECAT circuit could be viewed as being made up of some number of XOR trees with reconvergence of the generated parity signals. A circuit with the basic ECAT characteristics is referred to as an " ECAT circuit". The circuit shown below (fig.3.1.3) is an ECAT circuit.

DALG is typically used with the primitive gate (AND, OR, NAND, NOR) representation of a logic circuit. However, for ease of explanation DALG's techniques are extended to the XOR and equivalent gates. In generating a test, DALG creates a decision structure in which there is more than one choice available at each decision node. Through an implicit enumeration process, all alternatives at each decision node are capable of being examined. For the fault f in the circuit of fig. (3.1.3), DALG may go through the following steps:

1) The test for f requires logic 1 on M for the good machine. Setting E and F each to 1 result in a D at M.
2) Generating a sensitized path from the net M to the primary output Z, using recursive intersection of D-cubes may result in the ordered assignments K=1 and L=1 represented in the fig. (3.1.4). Alternate assignments K=0 and L=0 are still available for consideration, should the present assignments prove futile.
3) DALG justifies each internal net assignment on a levelized basis (see fig. 3.1.3). Since the functions P and $\overline{P}$ realized at nets K and L, respectively, are complementary, no justification is possible for the concurrent assignments K=1, L=1. However, in establishing the absence of the justification, DALG must enumerate 2^3) primary input values (2^n if 2n were the number of external inputs to the XOR tree) before it can correct the bad decision made on L, that is, change the assignment on L from 1 to 0. Looked at differently, DALG made two conflicting assignments K=1 and

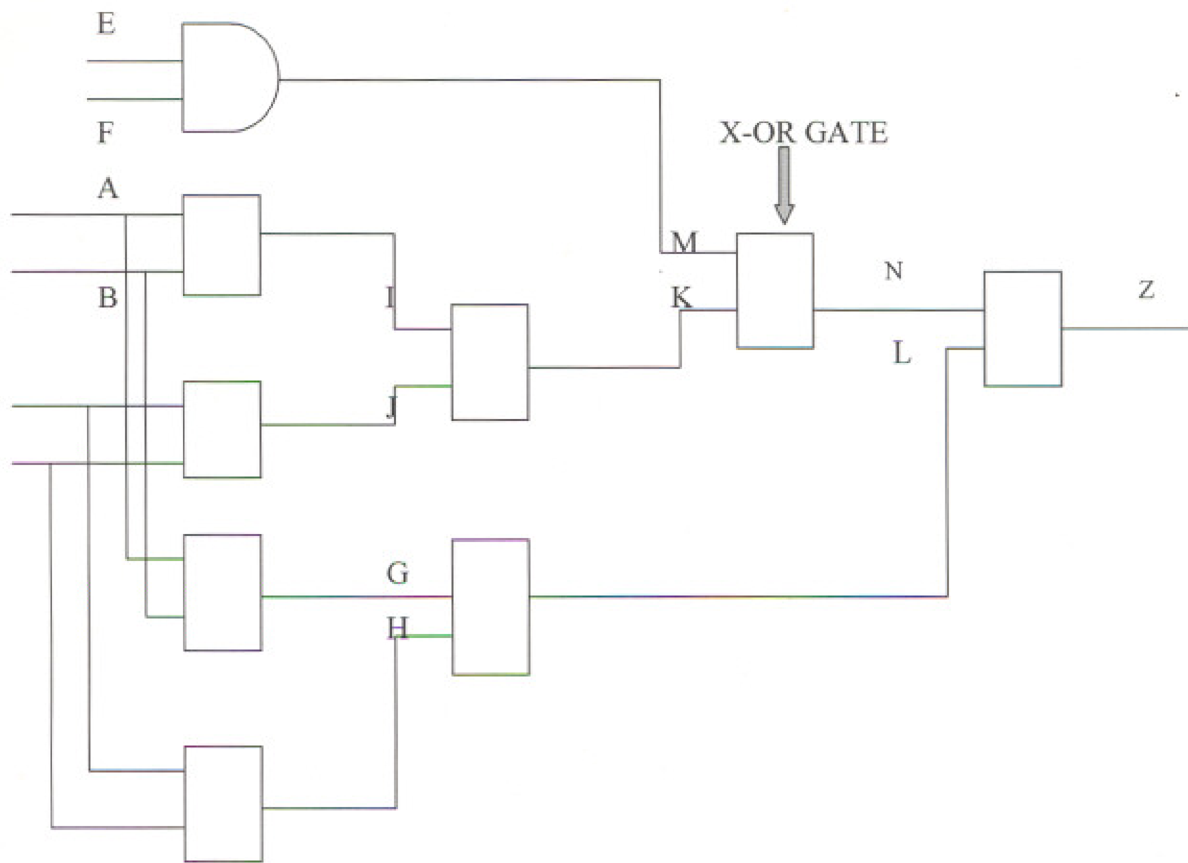


Fig. (3.1.3)

(ECAT type circuit exhibiting fan out and reconvergence involving XOR trees. Depicted fault illustrates DALG's inefficiency for such circuits.)

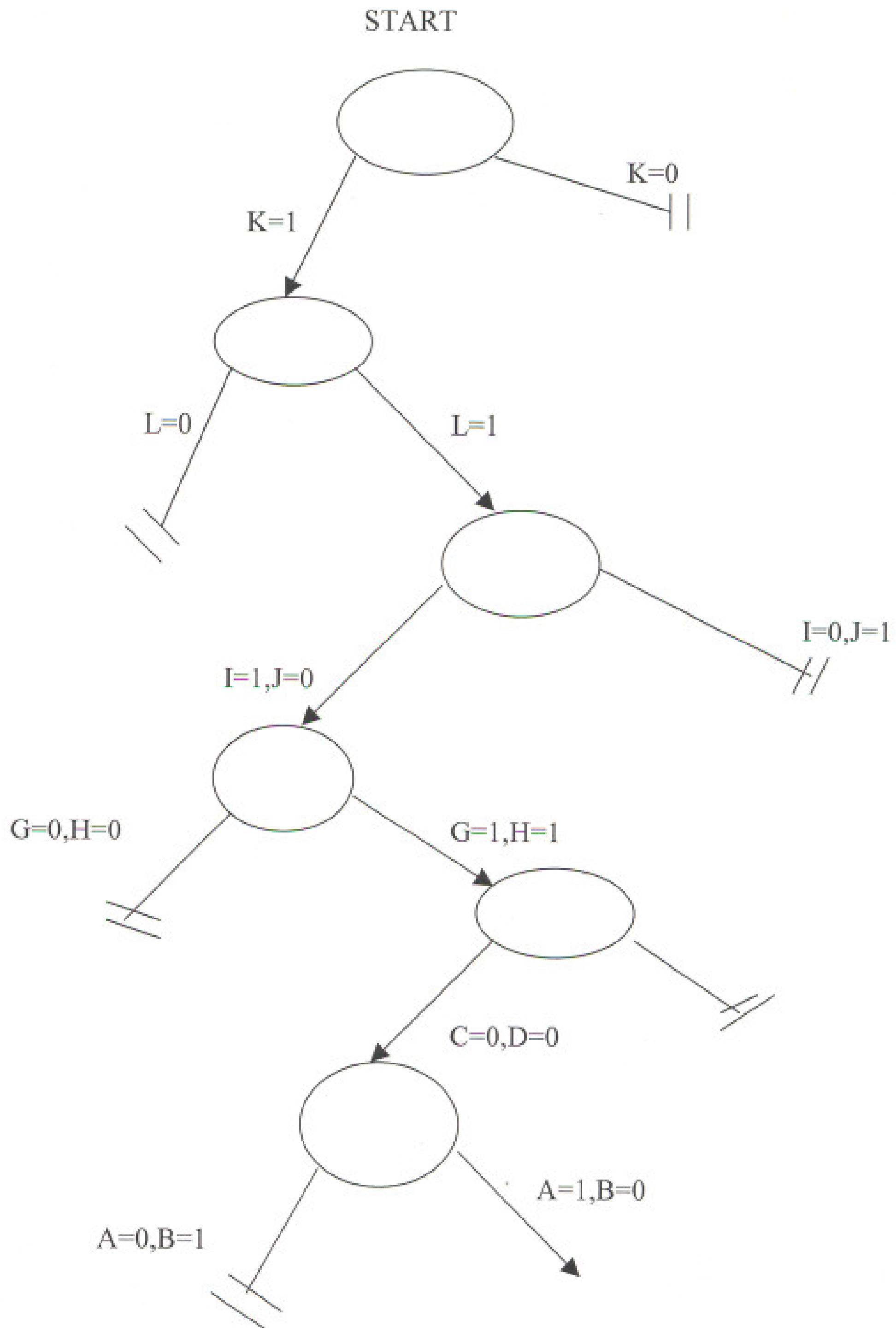


Fig. (3.1.4) DALG decision tree- DALG needs a large number of remade decisions resulting in a high test generation cost.

L=1 (on internal nets) and to detect the conflict DALG is forced to enumerate half of all Possible values on primary inputs A-D. The requirement of having all external inputs known to make the output known is peculiar to XOR trees.

Next consider what would happen for parity trees with up to 72 external inputs (as in original ECAT circuit). For some faults the number of primary inputs values that must be enumerated can be of the order of 2^{36}. With the above phenomenon repeating itself for most faults in ECAT circuits, it is seen that even if DALG were implemented on the fastest computers, it would have unacceptable in run time and therefore in test coverage.

An experimental study conducted demonstrated that DALG was ineffective for ECAT type circuits. When permitted to give up on faults after a large number of remade decisions, DALG was able to generate tests for only six out of 7000 faults in 2h of CPU time with an IBM 360/85 computer and DALG had still not completed.

3.2 PODEM Algorithm

The Path Oriented Decision Making (PODEM) algorithm is the outcome of the desire to improve the performance of D-Algorithm. The D- algorithm is shown to be ineffective for the class of combinational logic circuits that is used to implement error correction and translation (ECAT) functions. PODEM is a new test generation algorithm for combinational logic circuits. It uses an implicit enumeration approach analogous to that used for solving 0-1 integer-programming problems. PODEM is very efficient for ECAT circuits and is significantly more efficient than DALG over the general spectrum of combinational logic circuits. PODEM is more simpler than DALG. PODEM is a complete algorithm in that it will generate a test if one exists. Heuristics are used to achieve an efficient implicit search of the space of all possible primary input patterns until either a test is found or the space is exhausted.

Evaluation studies conducted shows that the D-algorithm is extremely inefficient in generating tests for combinational logic circuits that implement error correction and translation (ECAT) type functions. For practical purposes, a logic circuit is characterized as being ECAT type if portions of it are constituted by XOR (EXCLUSIVE-OR) gates with Reconvergent fan-out. The increasing emphasis on reliability of computer systems resulted in increased usage of ECAT circuits and existing test generation tools proved to be inadequate for them. An implicit enumeration algorithm (PODEM) was developed shortly thereafter and experiments demonstrate that PODEM is an effective test generator for ECAT circuits. Subsequent studies shows that PODEM algorithm is significantly faster than DALG over the general class of combinational logic circuits. For LSI logic circuits there is an increasing trend towards designing for testability using the level sensitive scan design approach or equivalent approaches.

The PODEM test generation algorithm is an implicit enumeration algorithm in which all possible primary input patterns are implicitly, but exhaustively, examined as tests for a given fault. The examination of PI patterns is terminated as soon as a test is found. If it is determined that no PI pattern can be a test, the fault is ***untestable*** by definition.

Combinational logic schematic is used in a manner similar to that in the D-algorithm. The flowchart of fig. (3.2.1) provides a high level description of the PODEM algorithm. The implicit enumeration process used in PODEM algorithm results in the decision tree structure illustrated by fig. (3.2.2). In figure 3.2.1 and 3.2.2 all PI's are initially at X, that is, they are unassigned. An initial assignment (" branch" – in the context of branch and bound algorithm) of either 0 or 1 on a PI is recorded as an unflagged node in the decision tree. Implications of present PI assignments (box 2) use the five-valued logic of Table 1 and the process is identical to the forward IMPLY process in the D-algorithm. The decision tree is an ordered list of nodes with: 1) each node identifying a current assignment of either a 0 or 1 to one PI, and 2) the ordering reflects the relative sequence in which the current assignments were made. A node is flagged (indicated by a check mark inside the node in fig. 3.2.1) if the initial assignment has been rejected and the alternative is being tried. When both assignment choices at a node are rejected, then the associated node is removed and the predecessor node's current assignment is also rejected.

The last PI assignment made is rejected if it can be determined (box 4 of fig. 3.2.1) that no test can be generated with the assignments made on the assigned PI's, regardless of values that may be assigned to the yet unassigned PI's. The rejection of a PI assignment results in a "bounding" of the decision tree, in the context of branch and bound algorithms, since it avoids the enumeration of the subsequent assignments to the yet unassigned PI's. The two simple propositions listed below are evaluated to carry out the process of box 4 in fig. (3.2.1).

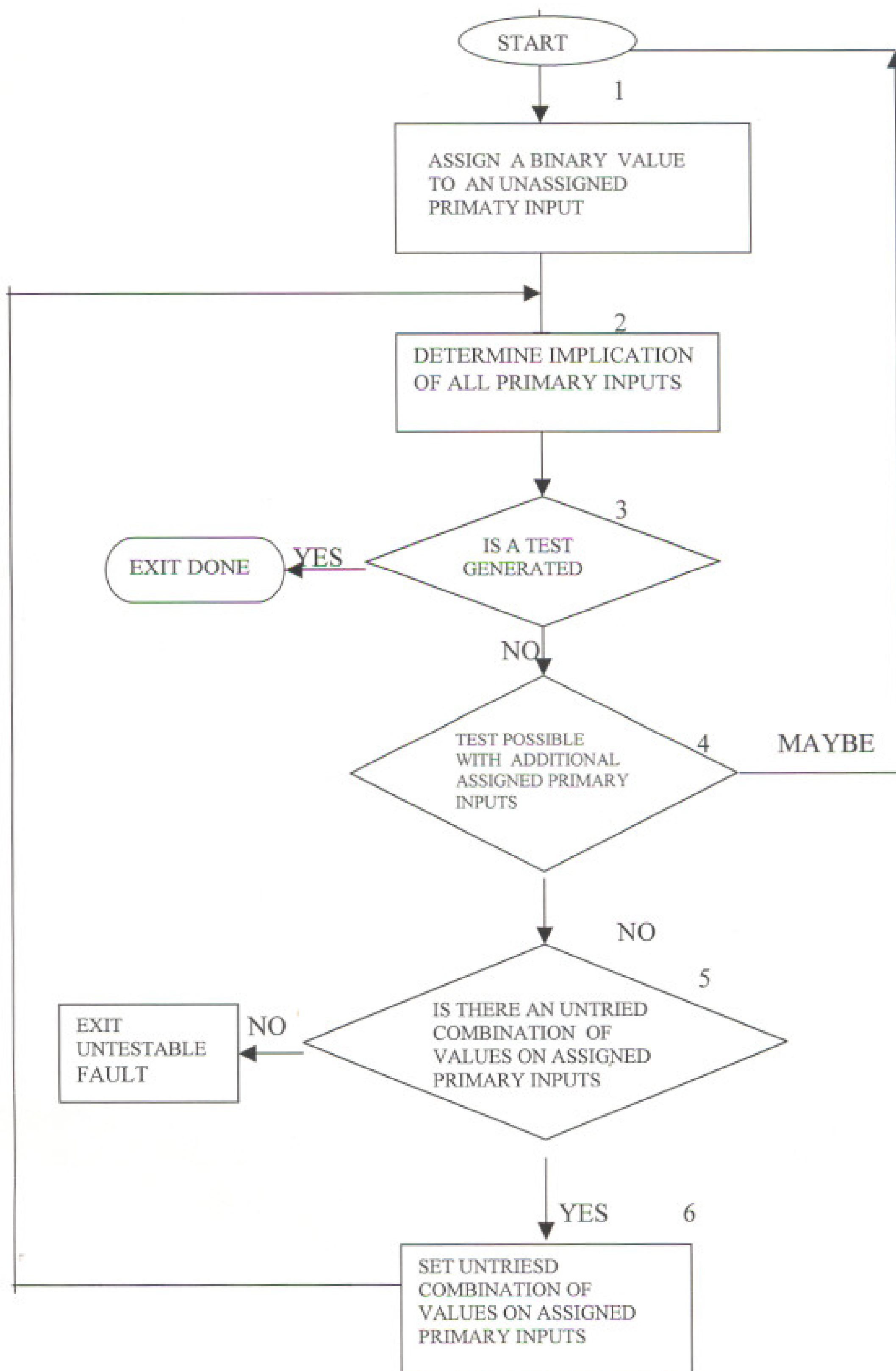


Fig. 3.2.1 High-level description of the PODEM algorithm

Proposition 1: The signal net (for the given stuck fault) has the same logic level as the stuck level.

Proposition 2: There is no signal path from an internal signal net to a primary output such that the internal signal net is at a D or $\overline{D}$ value and all other nets on the signal path are at X.

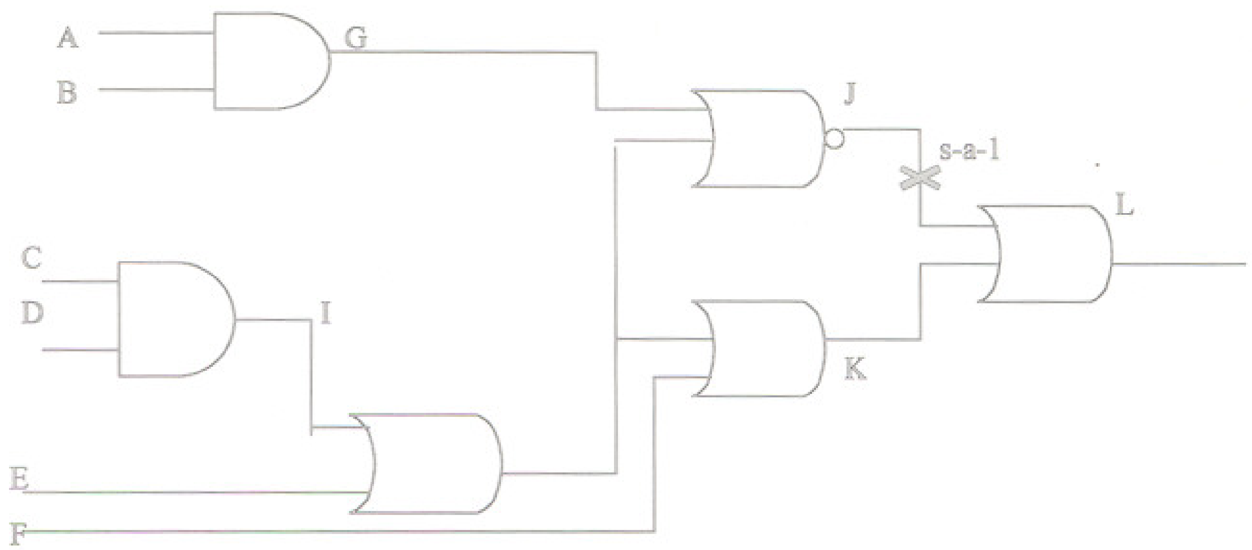
If proposition 1 is true, then it is obvious that assignment of known values to unassigned PI's cannot result in a test. If proposition 2 is true, then there exists no signal path along which the D or $\overline{D}$ can propagate to a circuit output. Hence, only if both propositions 1 and 2 are false, then in box 4 it is concluded that a test is possible with the present PI assignment even though further enumeration may show that it is not.

PODEM Example

It starts by assigning a value of 0 or 1 to a selected primary input (PI) line, and then determines its implication on the propagation of D or $\bar{D}$ to a primary output line. If no inconsistency is found, it again sometime selects another PI line and assigns a 0 or 1 and repeats the process which is referred to as branching . However, if at any time in this step of branching inconsistency is determined, the branching stops and bounding starts. This PI line which was most recently assigned a binary value is assigned an alternate value (if not assigned). If, however, both the values on the most recent line result in the bounding step then the PI line next to the most recent is treated in a similar manner. The complete process stops when either a test vector is found or when the fault is determined to be undetectable.

An example for PODEM, a representation of the binary space for J s-a-0 fault in the circuit under test is shown in figure (3.2.3). This search space was constructed using the simple heuristic of always trying the 1 logic value on a primary input first. The two sons of a node recursively correspond to the next PI lines, each for one of the two possible values on the further node. The process of finding a path from the root node to a test node in this binary tree. The branching step is simply to go as deeply into the tree as possible. The bounding step is going to the brother of the last node provided that the brother node has not been already been tried. Or else one is going to the first ancestor node of the last node that has an untried brother node and start branching from the new node. PODEM is thus a classic <u>branch and bound</u> algorithm which is employed in various artificial intelligence and operation research problems.

However, PODEM does not fully exploit the circuit structure. FAN (Fan-Out) algorithm is the outcome of the efforts taken to improve the effectiveness of PODEM.



(Circuit under test)

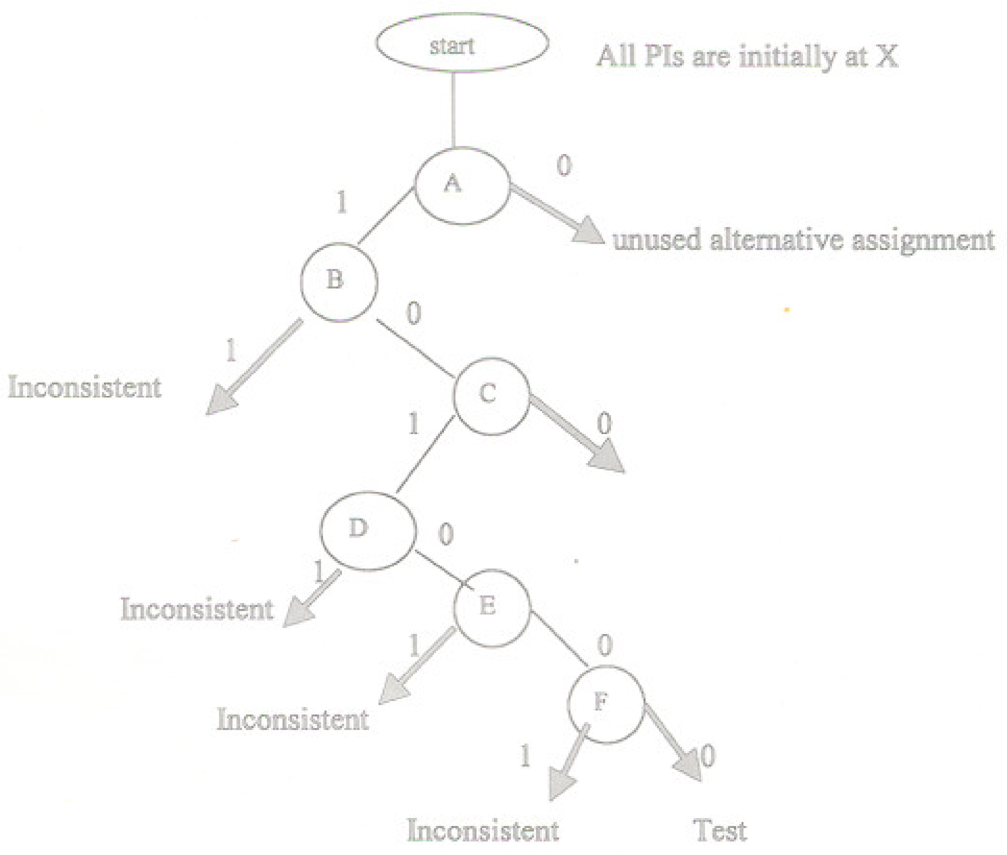


Fig. (3.2.3) (PODEM search-space graph)

3.3 FAN Algorithm

The Fan (fan-out-oriented test generation) algorithm is similar to the PODEM but includes improvements to increase its efficiency. *The major goal of Fan is to reduce the number of backtracks in the search tree.* This is accomplished using several techniques, including the consideration of fan-out branches in the circuit as a special case, hence the name Fan.

Goel showed that the PODEM algorithm is significantly faster than the D-algorithm by presenting experimental results. Indeed, the PODEM algorithm has succeeded in reducing the number of occurrences of backtracks in comparison to the D-algorithm. However, there still remain many possibilities of reducing the number of backtracks in the algorithm.

In order to accelerate an algorithm for test generation, it is necessary to reduce the number of occurrences of backtracks in the algorithm and to shorten the processing time between backtracks. FAN is a complete algorithm in that it will generate a test if one exists. Experimental results on large combinational circuits of up to 3000 gates demonstrate that the FAN algorithm is faster and more efficient than the PODEM algorithm over these circuits.

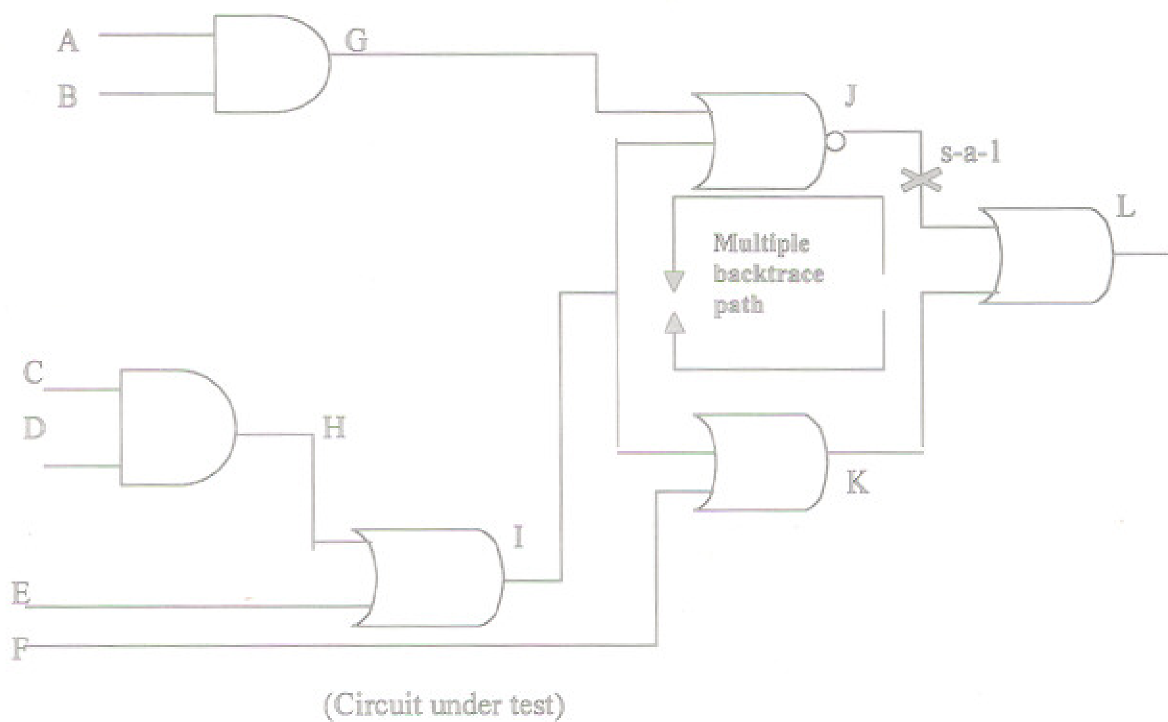
To examine this concept, We define several terminologies:

Free Line: is a circuit node that has no predecessors that are part of a fan-out loop. As such, free lines may have a uniquely assigned value.

Bound Lines: opposite of free line. Nodes J and K are bound lines and can not have unique (independent) values assigned to them.

Head lines: Head lines are free lines that drive a gate that is part of a reconvergent fan-out loop. Node I in the figure is a Headline. By definition, headlines can also be assigned values arbitrarily because they are free lines and can always be independently justified. *They can therefore be treated as primary inputs in the justification process.*

Identification of these nodes makes reconvergent fan-out loops much easier to handle. Once a test is found by treating headlines as primary inputs, the values on them can be justified at the end of the test generation process.



(Circuit under test)

In the accompanying figure, lines A through I are examples of free lines.

Multiple Backtrace:

In PODEM, the step of branching requires the selection of a new Pi line and a value on it. This is achieved by defining an initial objective and performing a backtrace operation to determine which Pi line will best satisfy the initial objective. The backtrace selects one line at each step. In FAN, backtrace occurs along as many path as possible and stops usually either at a fan out point or a head and used to determine the final objective. This is called multiple backtrace.

The multiple backtrace starts with more than one initial objective i.e. a set of initial objectives. Beginning with the set of initial objectives, a set of objectives which appear in the midst of the procedure is called the set of current objectives. A set of objectives obtained at the head lines is called a set of head objectives. A set of objectives on fan-out point is called a set of fan-out objectives. The multiple backtrace is stopped at the head lines because the sub circuit after head lines is completely fan out free. For fan-out free circuits, line justification can be performed without backtracks. Thus the values on the primary inputs which justify all the values on the head lines without backtracks can be found out therefore, the line justification of head lines is postponed to last.

Fan also uses a multiple – backtrace procedure for reconvergent fan-out branches buried in the circuit to reduce the number of backtracks that must be made in the search. For example , if a certain value is necessary at node L in the figure, and this circuit is part of some larger circuit , a single backtrace could be made along the path L-J-G-A,B. Values for inputs A and B could be chosen so that the goal is satisfied with a unique value on nodes I and K. Then if the value on K can not be achieved with the value chosen for I, a significant amount of backtracking in the search tree can result. First, a multiple – backtrace procedure would backtrace both the L-J-I and L-K-I paths and determine the value needed at I to satisfy the goal. This value would then be set as a requirement for the justification of the value at node L. This process can increase the Fan algorithm's efficiency significantly in a circuit with numerous buried reconvergent fan-out loops.

In order to reduce the number of backtracks, it is important to find the nonexistence of the solution as soon as possible. In the "branch and bound " algorithm, when we find that there exists no solution below the current node in the decision tree, we should backtrack immediately to avoid the subsequent unnecessary search. The PODEM algorithm seems to lack the careful consideration in this point.

- In each step of the algorithm, determine as many signal values as possible which can be *uniquely* implied.

To do this, we take the implication operation which completely traces such signal determination both forwards and backwards through the circuit. The idea of this complete implication is inherent in the D-algorithm, and hence represents an improvement only with respect to PODEM. Moreover, we can take other techniques as follows.

- Assign a faulty signal value D or $\overline{D}$ which is uniquely determined or implied by the fault under consideration (see fig. 3.3.1)

Note that we specify only the values that are uniquely determined. As an example, for a three- input AND gate in fig 3.3.1(a) and the fault E-s-a-0, we assign the value D to the output E and 1's to all inputs of the AND gate since these values are uniquely implied by the fault E s-a-0. However, for the fault E s-a-1, we assign only the value $\overline{D}$ to the output E. All the input values of the AND gate are left unspecified , i.e., X, since those values are not determined uniquely from E s-a-1 (see fig. 3.3.1(c)).

As example, consider the circuit of fig. 3.3.2. For the fault L s-a-1, we assign the value $\overline{D}$ to the line L and 1's to the inputs J, K, and E. Then after the implication operation, we have a test pattern for the fault without backtracks, as shown in fig. 3.3.2(a). On the other hand, in PODEM, the initial objective (L, 0) is determined to set up the faulty signal $\overline{D}$ to the line L, and then the backtrace procedure starts . As shown in fig. 3.3.2(b) the backtrace procedure causes a path to be traced from the initial objective line L backwards to a primary input B. The assignment B=0 implies that L=1. This contradicts the initial

objective, and setting L to $\overline{D}$ fails and a backward occurs. As seen in this example, the assignment of fig. 3.3.2(a) is a condition necessary for a test of the fault l s-a-1. By assigning the values which are uniquely determined, we can avoid the unnecessary choice.

Consider the circuit of fig. 3.3.3(a). Suppose that the D-frontier is {G2}.When the D-frontier consists of a single gate, we often have specific paths such that every path from the site of the D-frontier to a primary output always goes through those paths. In this example, every path from the gate G2 to a primary output passes through the paths F-H and K-M. In order to propagate the value D or $\overline{D}$ to a primary output, we have to propagate the fault signal along both F-H and K-M should be sensitized. Then we have the assignment C=1, G=1, J=1, and L=1 to sensitize them. This partial sensitization, which is uniquely determined, is called a *unique sensitization.*

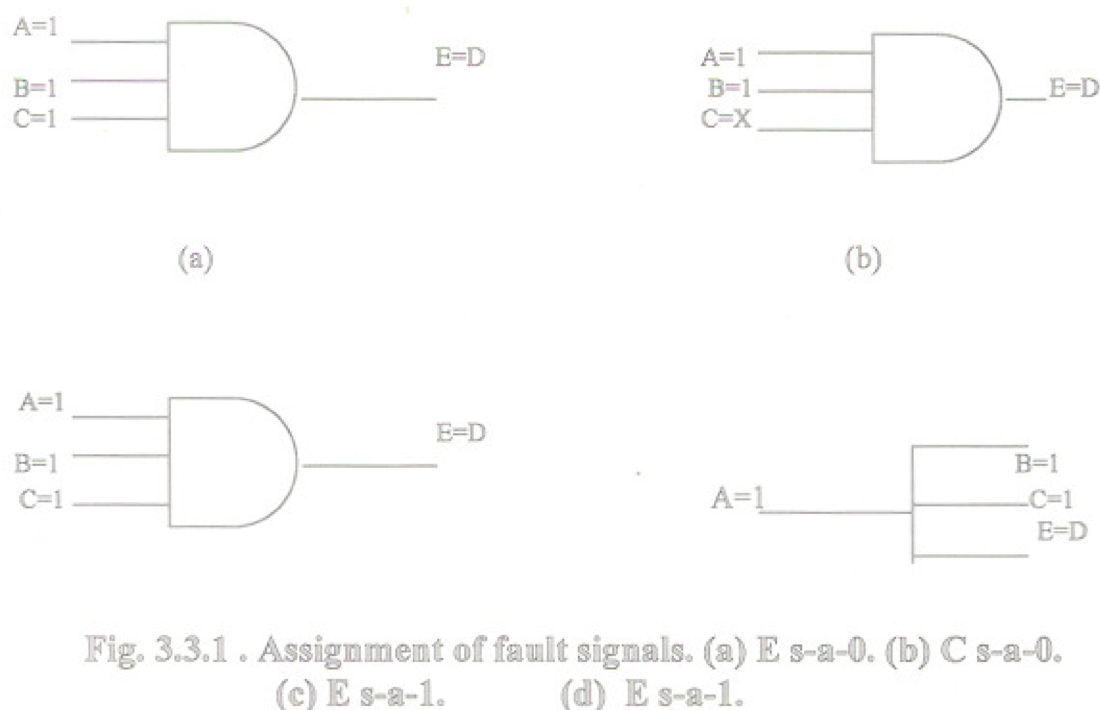


Fig. 3.3.1 . Assignment of fault signals. (a) E s-a-0. (b) C s-a-0. (c) E s-a-1. (d) E s-a-1.

Fig 3.3.2(a)

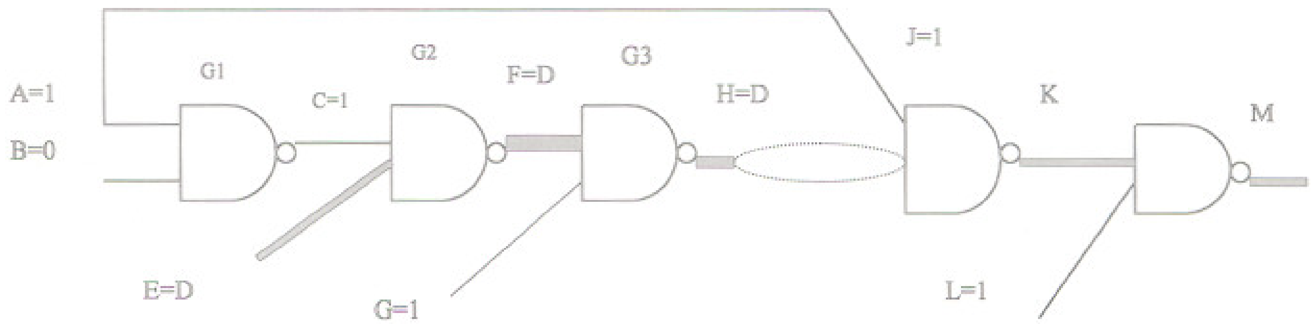
(b)

Fig. 3.3.2. Effect of fault signal assignment (a) Fault signal assignment and implication. (b) PODEM

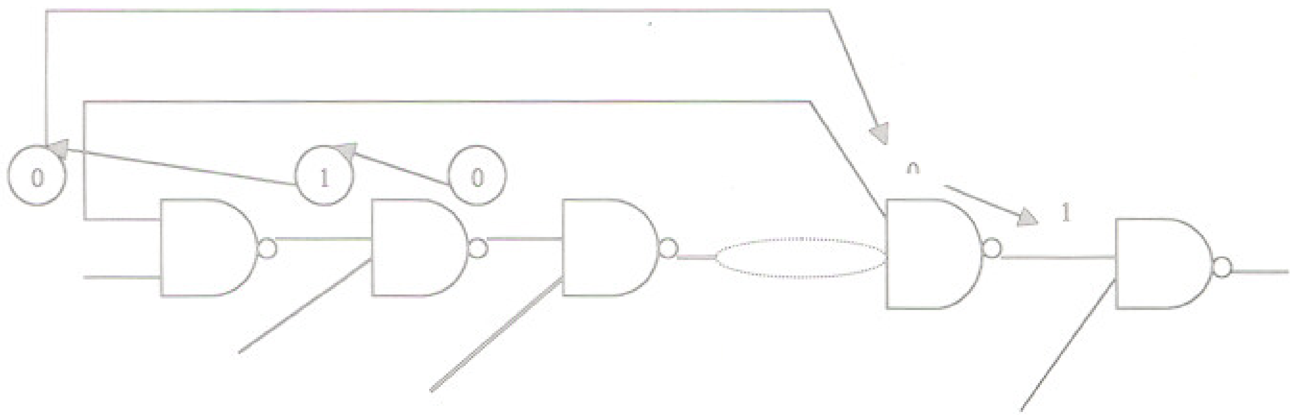
In fig. 3.3.3(a), after the implication of this assignment, we have A=1, B=0, F= $\bar{D}$, and H=D without backtracks. On the other hand, PODEM sets the initial objective (F, 0) to propagate the fault signal to the line F and performs the backtrace procedure. If the backtrace performs along the path as shown in fig. 3.3.3(b), we have J=0 and K=1 by implication. Although no inconsistency appears at this point, an inconsistency or the disappearance of the D-frontier will occur in the future when the faulty signal propagates from H to K. To find such an inconsistency, the PODEM algorithm uses a look ahead technique called the X-path check, i.e., it checks whether there is any path from a gate in the D-frontier to a primary output such that all the lines along the path are at X. However, in our example, the backtracking from A=0 to A=1 is unavoidable in PODEM.

- When the D-frontier consists of a single gate, apply a unique sensitization. As seen in the above examples, in order to reduce the number of backtracks, it is very effective to find as many values as possible, which are uniquely determined in each step of the algorithm. This is because the assignment of the uniquely determined values could decrease the number of possible selection.

The execution of the techniques mentioned above may result in specifying the output of a gate G, but leaving the inputs of G unspecified. This type of output line is called an *unjustified line*. It is necessary to specify input values so as to produce the specified output values. In PODEM, since all the values are first assigned only to the primary inputs and only the forward implication is performed, unjustified lines never appear. However, if we take the techniques mentioned above, the unjustified lines may appear, and thus in this case, some initial objectives will be produced simultaneously so as to justify them. This will be managed by introducing a multiple backtrace procedure, which is an extension of the backtrace procedure.



(a)



(b)

Fig: 3.3.3. Effect of unique sensitization . (a) Unique sensitization and implication (b) PODEM

Early detection of an inconsistency is very effective to decrease the number of backtracks. We shall continue to consider some techniques to find an inconsistency at an early stage.

When a signal line L is reachable from some fan-out point, that is, there exists a path from some fan-out point to L, we say that L is bound. A signal line, which is not bound, is said to be *free*. When a free line L is adjacent to some bound line, we say that L is a *headline*. As an example, consider the circuit of fig. 3.3.4(a). In the circuit, A, B, C, D, E, F, G, H, and J are all free lines and K, L, and M are bound lines. Among the free lines, J and H are headlines of the circuit since J and H are adjacent to the bound lines L and M, respectively.

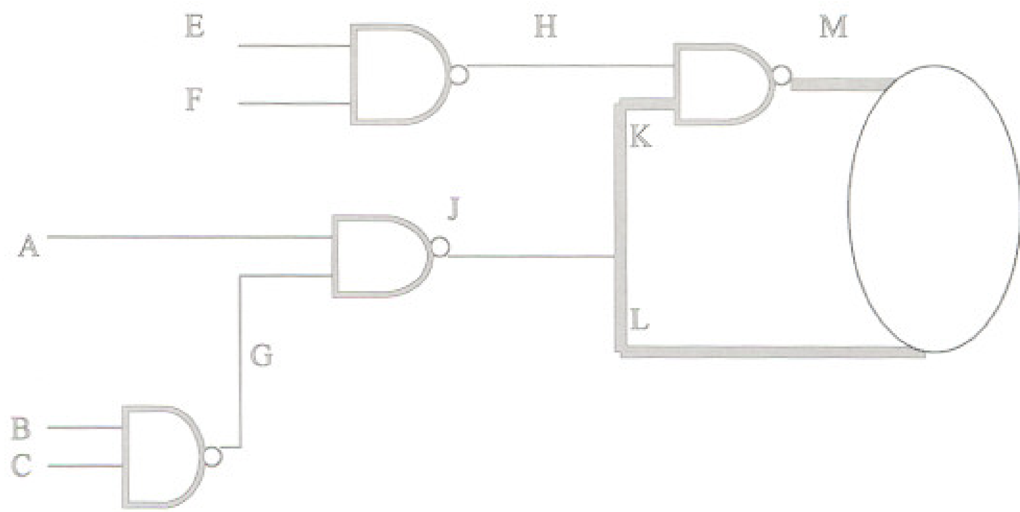
The backtrace procedure in PODEM traces a single path backwards to a primary input. However, it suffices to stop the backtrace at a headline for the following reasons. The sub circuit composed of only free lines and the corresponding gates is a fan-out-free circuit since it contains no fan-out point. For fan-out-free circuits, line justification can be performed without backtracks. Hence, we can find the values on the primary inputs which justify all the values on the head lines without backtracks. It is sufficient, or even efficient, to let the line justification for headlines wait to the last stage of test generation.

- *Stop the backtrace at a headline, and postpone the line justification for the headline to the last.*

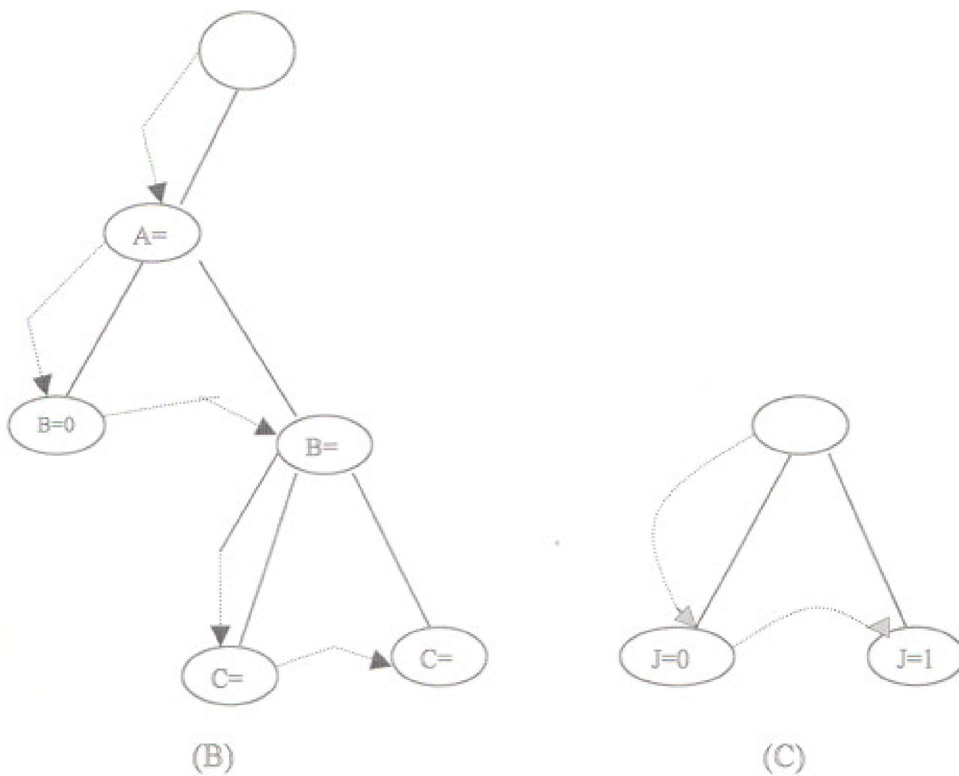
To illustrate this, consider the circuit shown in fig. 3.3.4 (a). Suppose that we want to set J=0 and do not know at the current stage that there exists no test under the condition J=0. In PODEM, the initial objective is set to (J,0), and the backtrace may result in the assignment A=1. Since the values of J is still not determined, PODEM again starts the backtrace procedure, and we get the assignment A=1. Since the values of J is still not determined, PODEM again starts the backtrace procedure, and we get the assignment B=0. A=1 and B=0 imply that J=0, and thus an inconsistency occurs for the current assignment, and PODEM must backtrack to change the assignment on B, as shown in fig.

3.3.4(b). In this case, if we stop the backtrace to the head line J, we can decrease the number of backtracks, as shown in fig. 3.3.4(c).

Performing a unique sensitization, we need to identify paths which would be uniquely sensitized. Also, we need to identify all the head lines in the circuit. These must be identified, and that topological information should be stored in some manner before the test generation starts. According to our experimental results, the computing time of the preprocess can be as small as negligible compared to the total computing time for test generation.



(A)



(B) (C)

Fig. 3.3.4 Effect of head lines. (A) Illustrative circuit (B) PODEM (C) Backtracking at head lines.

- Multiple backtrace, that is, concurrent tracing more than one path, is more efficient than the backtrace along a single path.

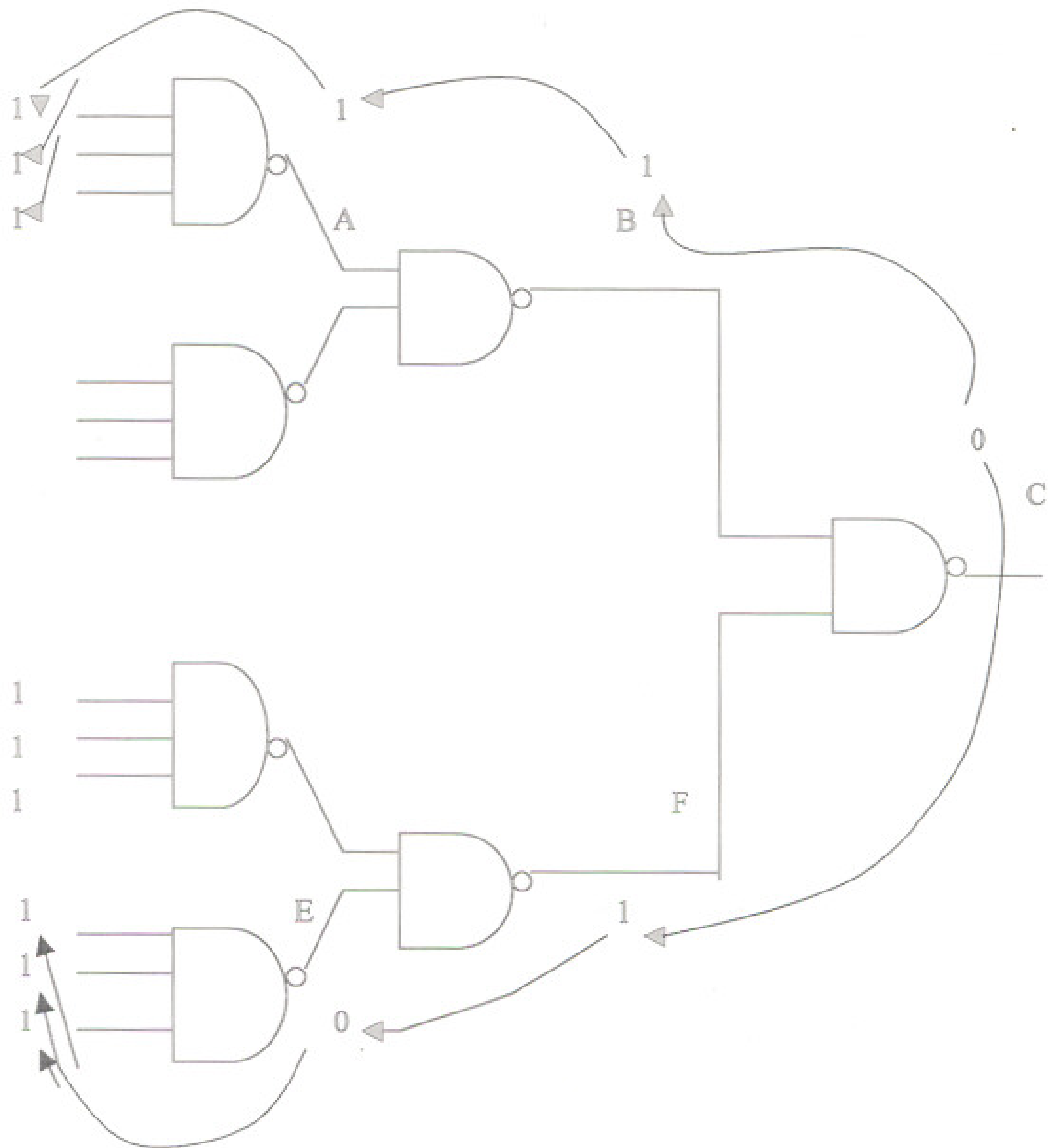
Consider the circuit of Fig. 3.3.5 For the objective of setting C=0, PODEM repeats the backtrace three times along the same path C-B-A, and also along the same path C-F-E before the value 0 is specified on C by the implication. Thus, we can see that the backtrace along a single path is inefficient and wastes time, which may be avoided. From this point of view, we could guess that the multiple backtrace along plural paths is more efficient than the single backtrace. The modes of procedure which implement multiple backtrace are various. In the following, we shall introduce a procedure for multiple backtrace.

In the backtrace of PODEM, an objective is defined by an objective logic level 0 or 1 and an objective line which is the line at which the objective logic level is desired. An objective which will be used in the multiple backtrace is defined by a triple:

$$(S, n_0(s), n_1(s))$$

where s is an objective line, $n_0(s)$ is the number of times the objective logic level 0 is required at s, and $n_1(s)$ is the number of times the objective logic level 1 is required at s.

The multiple backtrace starts with more than one initial objective, that is, a set of initial objectives. Beginning with the set of initial objectives, a set of objectives which appear in the midst of the procedure is called a set of current objectives. A set of objectives which will be obtained at head lines is called a set of head objectives. A set of objectives on fan-out points is called a set of fan-out-point objectives.



(Fig. 3.3.5) BACKTRACE

The flowchart of fig. 3.3.6 describes the multiple backtrace procedure. Each objective arriving at a fan-out point stops its back tracing while there exists other current objectives. After the set of current objective becomes empty, a fan-out point objective closest to a primary output is taken out, if one exists. If the fan-out point objective satisfies the following condition, the objective becomes the final objective in the backtrace process, and the procedure ends at the exit (D) in fig. 3.3.6. The condition is that the fan-out point p is not reachable from the fault line and both $n_0(p)$ and $n_1(p)$ are nonzero. In this case, we assign a value [0 if $n_0(p) > n_1(p)$ or 1 if $n_0(p) < n_1(p)$] to the fan-out point and perform the implications. The first part of the condition is necessary to guarantee that the value assigned is binary, that is, neither D nor $\bar{D}$.

In PODEM, the assignment of a binary value is allowed only to the primary inputs. I n our algorithm, FAN, we allow to assign a value to fan-out points as well as head lines, and hence the back tracing could occur only at fan-out points and head lines, and not at primary inputs. The reason why we assign a value to a fan-out point p is that there might exist a great possibility of an inconsistency when the objective in back tracing has an inconsistent requirement such that both $n_0(p)$ and $n_0(p)$ are nonzero. So as to avoid the fruitless computation, we assign a binary value to the fan-out point as soon as the objective involves a contradictory requirement. This leads to the early detection of inconsistency which would decrease the number of backtracks.

◊ In the multiple backtrace. If an objective at a fan-out point p has a contradictory requirement, that is, both $n_0(p)$ and $n_1(p)$ are nonzero, stop the backtrace so as to assign a binary value to the fan-out point.

When an objective at a fan-out point p has no contradiction, that is, either $n_0(p)$ or $n_1(p)$ is zero, the backtrace would be continued from the fan-out point. If all the objective arrive at head lines, that is, both sets of current objectives and fan-out point objectives are empty, then the multiple backtrace procedure terminates at the exit (C) in fig. 3.3.7. After this, taking out a head line one by one from the set of head objectives, we assign the corresponding value to the head line and perform the implication.

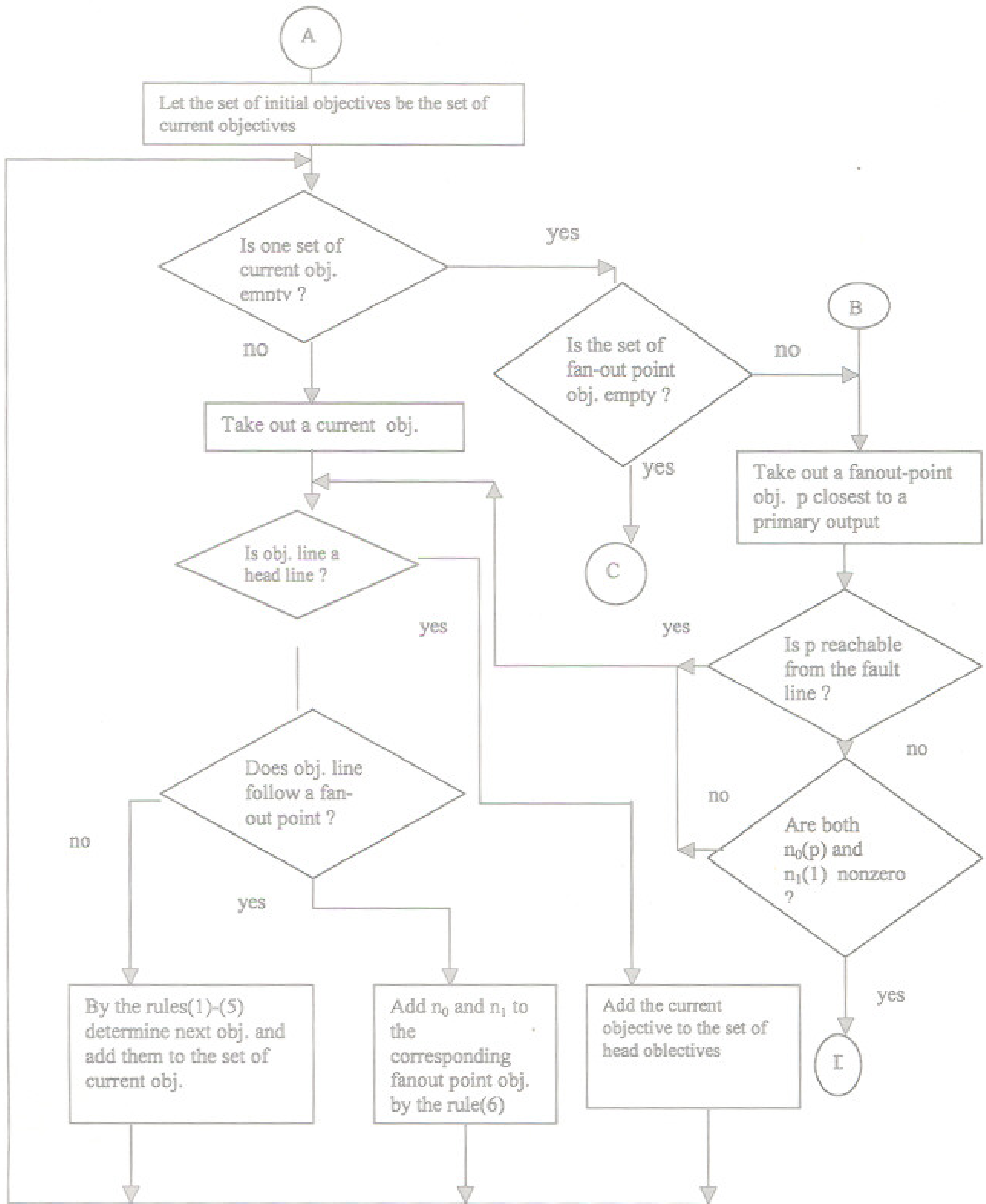


Fig. 3.3.6. Flowchart of multiple backtrace.

DESCRIPTION OF THE FAN ALGPRITHM

The FAN (fan-out –oriented) test generation algorithm is similar to PODEM based on the implicit enumeration process. However, the FAN algorithm is characterized by putting emphasis on the following points.

1) FAN pays special attention to fan-out points in circuits.

2) FAN is a branch-and-bound algorithm which adopts many techniques presented in the preceding section so as to detect an inconsistency as early as possible.

As mentioned in the preceding section, those techniques would be very useful to decrease the number of backtracks, and thus FAN could be faster and more efficient than PODEM. The flow chart of the FAN algorithm is given in fig. 3.3.7. Each box in the flowchart will be explained in the following.

1) **Assignment of Fault Signal:** In box 1, a fault signal D or $\bar{D}$ assigned in the manner shown in fig. 3.3.1.

2) **Backtrace Flag:** The multiple backtrace procedure of Fig. 3.3.7 has two entries: one entry is (A) where the multiple backtrace starts from a set of initial objectives, and other entry is (B) where the multiple backtrace starts with a fan-out-point objective to continue the last multiple backtrace which terminated at a fan-out point. The backtrace flag is used to distinguish the above two modes.

3) **Implication:** We determine as many signal values as possible which can be uniquely implied. To do this, we take the implication operation which completely traces such signal determination both forwards and backwards through the circuit. In PODEM, since all the values are assigned only to the primary inputs and only the forward implication is performed, unjustified lines never appear. Therefore, so as to justify those lines, the multiple backtrace is necessary, not only to propagate the fault signal (D or $\bar{D}$), but also to justify the unjustified lines.

4) **Continuation check for multiple backtrace**: In box 4, we check whether or not it is meaningful to continue the backtrace. We consider that it is not meaningful to continue the backtrace if the last objective was to propagate D or $\bar{D}$ and the D-frontier has changed or if the last objective was to justify unjustified lines and all the unjustified lines have been justified. When it is not meaningful to continue, The backtrace flag is set so as to start the multiple backtrace with new initial objectives.

5) **Checking D-frontier**: PODEM uses an look ahead technique called an X-path check, i.e. it checks whether there is any path from a gate in the D-frontier to a primary output such that all lines along the path are at X. In FAN, the same technique is adopted to eliminate a meaningless D- frontier. FAN counts only those gates in the D-frontier which have an X-path.

6) **Unique Sensitization**: The unique sensitization is performed in the manner mentioned In the previous section (fig. 3.3.4) . Although the unique sensitization might leave some lines unjustified, those lines will be justified by the multiple backtrace.

7) **Determination of final objective**: The detailed flowchart of box 6 is described in fig. 3.3.6. By using the multiple backtrace procedure, we determine a final objective, that is we choose a value and a line such that the chosen value assigned to the chosen line has a good likelihood of helping towards meeting the initial objectives.

8) **Backtracking**: The decision tree is identical to that of PODEM, that is an ordered list of nodes, with each node identifying a current assignment of either 0 or 1 to one head line or one fan-out, and the ordering reflects the relative sequence in which the current assignments were made. A node is flagged if the initial assignment have been rejected and the alternative have been tried. When both assignment choices at a node are rejected, Then the associated node is removed and the predecessor node's current assignment is also removed. The backtracking done by PODEM does not require saving and restoring of status because, at each point, the status could be revived only

by forward implications of all primary inputs. The same backtracking can be done in FAN, which does not require saving and restoring of status, by implication of all associated head lines and fan-out points. However, to avoid the unnecessary repetition of implications, FAN allows the process of saving and restoring of status to some extent.

9) **Line Justification of Free Lines**: We can find values on the primary inputs which justify all the values on the head lines without backtrace . This can be done by an operation identical to the consistency operation of the –algorithm.

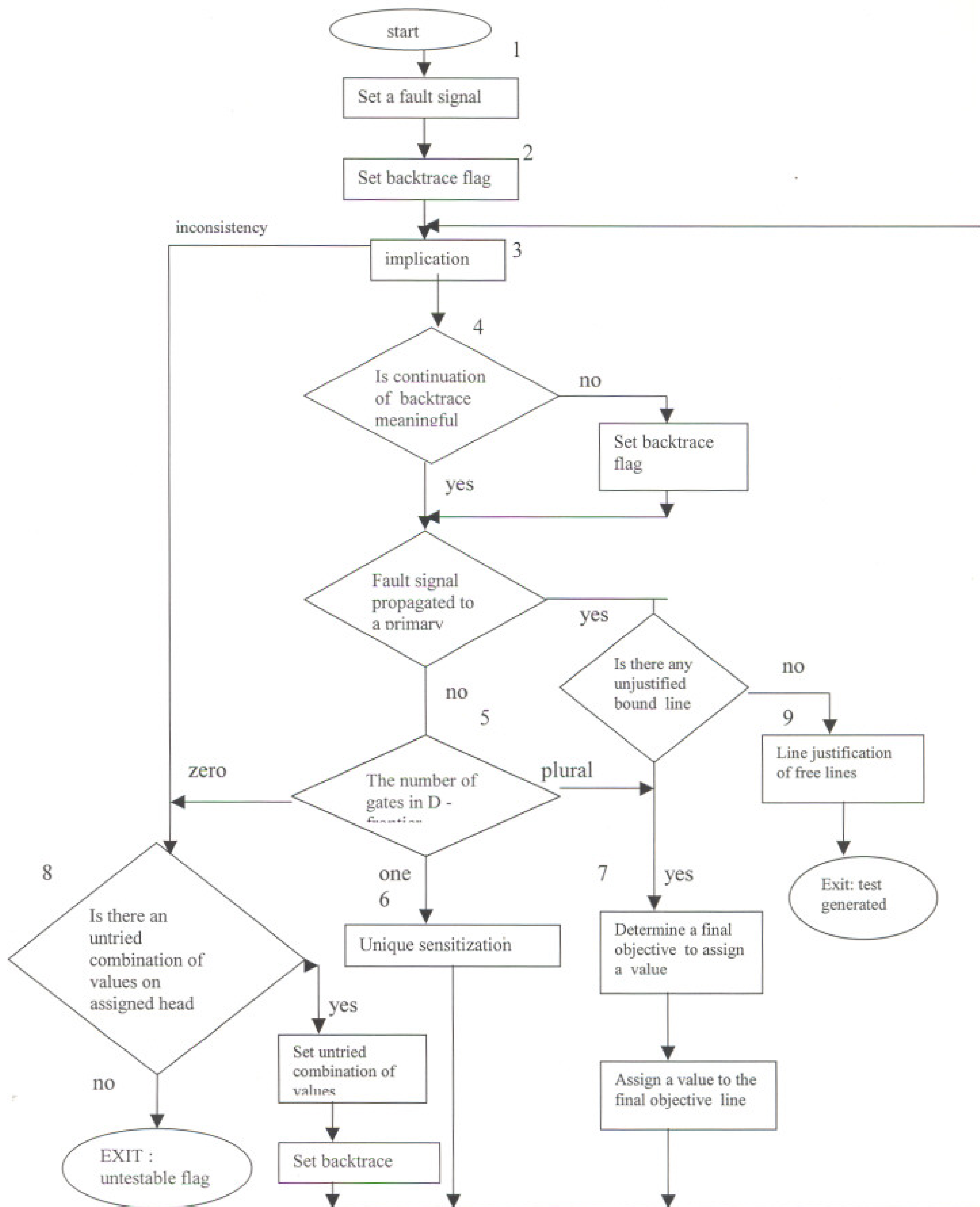


Fig. 3.3.7. Flowchart of FAN algorithm

4. Parallel Processing Techniques

Generating test patterns for testing digital circuits still consumes a significant portion of design time. Generation of test patterns for combinational logic is a search through the set of all input values to find one that causes the output of a good circuit to differ from that of one containing a fault.

Parallel processors are computing machines comprising a number of processors that are separate entities capable of performing disparate tasks. These entities are linked by some mechanism so that they can exchange data and thus cooperate to solve a problem.

One criteria for evaluating the quality of a parallel solution to a problem is how well it scales. An algorithm scales well if the computation time decreases linearly, or nearly so, with an increase in the number of processors in the system.

The speedup of a given parallel algorithm is defined as the ratio of the time taken by the fastest sequential algorithm running on an equivalent uniprocessor to the time taken by the parallel algorithm on the parallel machine. The goal is to have the algorithm's speedup scale linearly with the number of processors.

We present several techniques that have been used to parallelism ATPG. These techniques fall into five major categories:

- Fault Partitioning
- Heuristic Parallelization
- Search-space partitioning
- Functional (Algorithmic) partitioning, and
- Topological partitioning

4.1 Fault Partitioning

This is one of the basic method used for parallelization. The simplest way to parralelize the ATPG problem is to divide the fault list among the processors. Each processor then generates tests for each fault on its portion of the fault list until all tests have been generated. This scheme results in each processor having a completely separate task in that it performs the entire test generation procedure on its own. If the fault list is divided carefully, each processor will have roughly the same amount of work and will finish in about the same time.

In practice, optimal partitioning of the fault list is not easy to do a priori, so the scheduling can be done dynamically, with each processor requesting a new fault from a master scheduler whenever it is idle. Dynamic scheduling requires increased communications overhead because of requests from idle processors.

The fault-partitioning method is especially suitable for message passing systems because synchronization is necessary only when a new fault is needed from the remaining fault list.

Fault –partitioning system can be used in conjunction with fault simulation. In this system after a processor generates a test for a specific fault fi, It performs fault simulation to determine what other faults this test vector covers. If another fault fj is covered by the test vector, it must be removed from the fault list. If fi has been assigned to another processor, a message must be sent to that processor instructing it to remove fj from the fault list. This communication increases the parallel system's overhead and reduces the possible speedup.

To reduce this communication overhead , it is necessary to assign faults that are likely to be covered by the same test vector to the same processor. Like static scheduling, this division of the fault list is difficult to do a priori. If communication of covered tests is removed completely, tests will be generated for faults that have already been covered, and this redundancy will result in a larger test set.

Several methods of fault-partitioning include random partitioning, partitioning by input and output cones, and mandatory constraint propagation. Mandatory constraint propagation refers to grouping all faults with the same unique implication values on certain nodes into pseudo compatible fault sets. All the faults within these sets are then assigned to the same processor. Mandatory constraint propagation is more complex than the other three methods.

The results of using these techniques to partition faults across processors statically indicate that although processing by input/output cones or mandatory constraint propagation produced smaller test sets, The load on the processors was unbalanced. This occurred because these partitioning methods tended to assign related hard to detect faults (with many backtracks)

For this a static and dynamic load-balancing technique was developed. Initially, faults are allocated to a processor using one of the methods discussed previously. If a processor succeeds in generating tests for all of the faults in its list, it requests work from the processor with the largest remaining fault list. This processor sends half of its list to the idle processor. This load-balancing scheme results in high message traffic only toward the end of the test generation process. Results indicate that partitioning by input cones performed best and resulted in near linear speedup for up to 16 processors on the largest benchmark circuits.

One of the greatest disadvantages of fault partitioning is the long setup time for a message-passing system. The entire ATPG program and circuit database must be loaded into each processor's memory across the message fabric. If the total amount of work that can be divided among the processor's is large, then the percentage of time spent on setup can be small and this scheme has promise. But if the circuit has a small number of faults, or fault classes, then the speed-up will be limited. In any case, because of the long setup time, this method does not scale well on smaller circuits.

Moreover, the method also performs poorly if only a few hard-to-detect faults account for most of the processing time. Results shows that systems that uses this technique show that linear speedup is possible only for a small number of processors, usually less than 10. Clearly, this method of parallelization is less than optimum, although it is the simplest to implement.

4.2 Heuristic Parallelization

Heuristics are used to guide ATPG. Research has indicated that many heuristics will produce a test for a given fault within some computation time when other heuristics have failed to do so. These complementary heuristics can be used in a multiprocessor system to aid ATPG. Two basic strategies are used: *a variation of fault-partitioning scheme and concurrent parallel heuristics.*

In the variation of fault- partitioning method, which is now called uniform partitioning, the fault list is divided among the processors and each generates tests, however, multiple heuristics are used in sequential order. If a heuristics fails to generate a test within a time limit, that heuristics is discontinued and the next one on the list is begun. This scheme has the same advantages and disadvantages as that of the fault- partitioning scheme. However it is slightly better in some cases because the multiple heuristics shorten the test generation time for hard to detect faults.

In the concurrent parallel heuristic method, the system is required to have k*h processors, where h is the number of different heuristics available. If k equals 1, each processor computes a test for the same fault using one of the h heuristics. Whenever a processor succeeds in generating a test for the fault, it sends a "stop work" message to the other processors in the cluster and they stop processing that fault. A new fault is selected from the fault list and the process begins again.

If k is greater than 1, the processors are clustered into groups of h and each cluster works on a separate fault. In this case, the system is actually using a combination of the fault-partitioning and heuristic parallelization schemes.

The **concurrent parallel heuristics method** has the potential to achieve greater speed ups than the uniform – partitioning because of possible anomalies in the ordering of the heuristics for different faults. For example, the time limit for each of five heuristics in the uniform- partitioning method is 10 seconds and only the last heuristic on the list can generate a test for a specific fault within the time limit, say in 5 minutes. Then the processing time for the uniform –partitioning method will be 45 seconds. However, the concurrent heuristic method will find a test for this same fault in only 5 seconds.

The major drawback of the concurrent heuristic method is that for each fault, the work of the ***h-x*** processors (where x is the number of heuristics the uniform method used to generate a test) is wasted. Also, in a message –passing system, computation time is wasted while the "stop work" message travels over the message fabric. So, for most of benchmark circuits the concurrent heuristics method generally doesn't perform as well as the uniform-partitioning method.

4.3 Search-Space Partitioning:

One way to achieve cooperation among processors is to use the divide-and-conquer approach, dividing the process into smaller tasks that can be completed in parallel. This concept is called algorithmic parallelization also called AND parallelism.

Another way to parallelize work on a single Fault is to divide the search space into disjoint pieces and evaluate them concurrently. This approach is parallel implementation of the branch-and-bound method, which involves concurrent evaluation of sub problems. This technique is called OR parallelism. This method is based on PODEM because PODEM orders the search space and allows it to be divided easily.

Another way is to divide the search space such that sub problems skipped by one processor are evaluated by another. The search spaces for the processors are therefore disjoint and are spread as far as possible across the solution space to maximize the area of the current search. This organization increases the chances of finding a valid solution quickly.

Fig. 4.1 illustrates the process of dividing a search tree. The search space belonging to processor X is divided into two parts for processors X and Y. Note That the processors are in fact always working on different problems (that is, disjoint search spaces) and that each processor will backtrack to a different place. If processor X finds a conflict, it backtracks and tries an alternate value for input A. Processor Y backtracks and tries an alternate value for input C in case of a conflict. This approach keeps the current search space as large as possible, which tends to make the search more efficient.

4.3 Functional (algorithmic) partitioning

This method can be used to allow more than one processor to work simultaneously on finding a test for a single fault. Functional partitioning refers to dividing an algorithm into independent subtasks that can then be executed on separate processors in parallel. This method of parallelization is also known as algorithmic, or AND, parallelism. Most serial ATPG algorithms developed thus far are difficult to parallelize functionally. The few subtasks that can be identified, such as fault sensitization and path sensitization, are not independent. That is, action taken to perform one of these processes may change the circuit state such that it has a side effect or causes an inconsistency in another process. Two goals cannot be justified simultaneously unless they consist of assigning values to two separate basis nodes. One way to allow parallelism in justification is to perform justification for goals in different faults simultaneously. This is an adaptation of the fault-partitioning scheme already discussed.

The Motorola system uses a type of functional partitioning to remove the easy-to-detect faults from the fault list. This procedure is done before the parallel method for hard-to-detect faults presented in the previous section is run. The method begins by dividing the fault list into groups of related faults. Typical related faults include those along the same path between a fault site and a primary output. After the fault list is divided into groups, each group is sent to a cluster of processors that includes a test generator and a fault simulator. The test generator takes the first fault and generates a test for it using a PODEM algorithm with a limited number of backtracks. If a test for a fault is not generated within the backtrack limit, it is considered a hard-to-detect fault and is processed as such later. If a test is found, it is sent to a fault simulator node. This node runs a version of a concurrent fault simulator to determine which other faults the test detects. These faults are then removed from the fault list.

This technique reduces the size of both remaining fault list to be processed by the ATPG algorithm and the test set itself. However, in some cases, performing fault simulation after algorithmic ATPG might not be very productive if the algorithmic ATPG is done carefully. For example, If the ATPG algorithm sensitizes a path for a stuck-at fault on a

node to a primary output, every node in that path is also tested for a stuck-at fault. If the ATPG algorithm keeps a list of these incidentally tested faults and removes them from the fault list, fault simulation may not be productive enough to warrant the additional computation time it entails.

4.4 Topological Partitioning

Each processor in all the algorithms discussed so far must have access to access to the entire circuit database. This may be a problem for large circuits because each processor may not have enough memory to hold the entire circuit database. Also, loading the database into memory in a message-passing system takes time. Topological partitioning of the circuit into separate partitions and instantiating each on a different processor would help alleviate this problem.

Researchers have been investigating topological partitioning for parallel logic simulation for some time. Hirose et al. Present results of using a parallel logic simulator to generate tests for combinational logic circuits. The machine used is the special-purpose simulation processor developed by Fujitsu Ltd. In this system, the circuit is divided topologically among several processors, called gate processors, in preparation for fault simulation. Faults are injected into the circuit and then a depth-first search similar to that of PODEM searches the entire input space to find an input that detects the injected faults. The authors analyze six different partitioning schemes: random partitioning, natural partitioning, partitioning by gate level, partitioning by element strings, and partitioning by fan-in and fan-out cones.

Natural partitioning consists of dividing gates into groups according to their position in the gate list. This method is only slightly more ordered than random partitioning. Partitioning by gate level refers to assigning gates to groups according to their level in the circuit, that is, their distance from the primary inputs. Partitioning elements by strings refers to grouping sets of connected gates that include at most one fan-in or fan-out connection. Partitioning by fan-out cones is done as in the lover system except that when gates are connected to more than one fan-out cone, they are assigned to the one they are

more highly connected to. Fan-in cones are found in the same manner except that the process begins at each primary input.

The results show that for simulation, random partitioning scores best in concurrency but worst in interprocessor communication. The condition would make random partitioning a bad choice for most systems. Partitioning by fan-in and fan-out cones offers the best trade-off between concurrency and interprocessor communication, with fan-out cones being slightly better.

It appears that the optimum partitioning method for ATPG will depend on the algorithm used. PODEM uses a simulation-like process for justification, and partitioning by fan-out cones may be best. However, circuit activity in the D-algorithm tends to be along circuit levels, as during the advance of the D frontier. Using D-algorithm, partitioning by gate levels may produce better results. In any case, the circuit must be partitioned carefully to reduce the communication required between the partitions. If the circuit is partitioned incorrectly , the algorithm becomes communication bound and the computation time is dominated by the interprocessor communication time. The topological- partitioning approach would seem to be better suited to shared-memory machines with their inherently lower communication times.

Summary

Except for search-space partitioning, no techniques have demonstrated the capacity to produce linear speedup for more than 16 processors. As a summary, Search-space partitioning shows the most promise for scalability to large numbers of processors. However, it does not answer the problem of large circuit database created by increasing VLSI circuit sizes. Also, this technique is applicable only to hard-to-detect faults and does not address acceleration of the ATPG problem for easy-to-detect faults, which constitute the majority of the fault list for most practical circuits. Clearly, a combination of techniques or altogether new ATPG algorithm designed for parallel processors will have to be developed to utilize the massively parallel machines with hundreds or thousands of processors that will be available in the future.

5. Modeling Techniques (OMT)

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software industry and software engineers to continuously look for new approaches to software design and development, which is becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a serious crisis within the industry. The following issues need to be resolved to overcome this crisis.

How to represent real-life entities of problems in system design?

How to design systems with open interfaces?

How to ensure reusability and extensibility of modules?

How to develop modules that are tolerant to any changes in future?

How to improve software productivity and decrease software cost?

How to manage time schedules?

How to industrialize the software development process?

Changes in user requirements have always been a major problem. Another study shows that more than 50% of the systems required modifications due to changes in user requirements and data formats. It only illustrates that, in a changing world with a dynamic business environment, requests for change are unavoidable and therefore systems must be adaptable and tolerant to changes.

Some of the quality issues that must be considered for critical evaluation are:

1. Correctness
2. Maintainability
3. Reusability
4. Openness and interoperability
5. Portability
6. Security
7. Integrity
8. User Friendliness

A model is an abstraction of something for the purpose of understanding it before building it. A model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity. Engineers, artists, and craftsman have built models for thousands of years to try out designs before executing them. To build complex systems, the developer must abstract different views of the system, build models using precise notations, verify that the model satisfies the requirements of the system, and gradually add detail to transform the models into an implementation.

Designers build many kind of models for various purposes before constructing things. Examples include architectural models to show customers, airplane scale models for wind-tunnel-tests, pencil sketches for composition of oil paintings, blueprints of machine parts, and outlines of books. Models serve several purposes:

Testing a physical entity before building it: The medieval masons did not know modern physics, but they built scale models of the Gothic cathedrals to test the forces on the structure. Scale models of airplanes, cars, and boats have been tested in wind tunnels and water tanks to improve their aerodynamics. Recent advances in computation permit the simulation of many physical structures without having to build physical models. Not only is simulation cheaper, but also it provides information that is too fleeting or inaccessible to be measured from a physical model. Both physical models and computer models are usually cheaper than building a complete system and enable flaws to be corrected early.

Communication with customers: Architects and product designers build models to show their customers. Mock-ups are demonstration products that imitate some or all of the external behavior of a system.

Visualization: Storyboards of movies, television shows, and advertisements allow the writers to see how their ideas flow. Awkward transitions, dangling ends, and unnecessary segments can be modified before detailed writings begin. Artists '

sketches allow them to block out their ideas and make changes before committing them to oil or stone.

Reduction of complexity: Perhaps the main reason for modeling, which incorporates all the previous reasons, is to deal with systems that are too complex to understand directly. The human mind can cope with only a limited amount of information at one time. Models reduced complexity by separating out a small number of important things to deal with at a time.

Abstraction

Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant. Many different abstractions of the same thing are possible, depending on the purpose for which they are made.

A good model captures the crucial aspects of a problem and omits the others. Most computer languages are poor vehicles for modeling algorithms because they force the specification of implementation details that are irrelevant to the algorithm. A model that contains extraneous detail unnecessarily limits your choice of design decisions and diverts attention from the real issues.

Encapulation

Encapsulation (also information hiding) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects. The implementation of an object can be changed without affecting the applications that use it. One may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or for porting. Encapsulation is not unique to object-oriented languages, but the ability to combine data structure and

behavior in a single entity makes encapsulation cleaner and more powerful than in conventional languages that separate data structure and behavior.

Combining Data and Behavior

The caller of an operation need not consider how many implementation of a given operation exist. Operator polymorphism shifts the burden of deciding what implementation to use from calling code to the class hierarchy. For example, non-object-oriented code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An object-oriented program would simply invoke the *draw* operation on each figure; the decision of which procedure to use is made implicitly by each object, based on its class. It is unnecessary to repeat the choice of procedure every time the operation is called in the application program. Maintenance is easier, because the calling code need not be modified when a new class is added.

Sharing

Object-oriented technique promotes sharing at several different levels. Inheritance of both data structure and behavior allows common structure to be shared among several similar subclasses without redundancy. The sharing of code using inheritance is one of the main advantages of object-oriented languages. More important than savings in code is the concept clarity from recognizing that different operations are all really the same thing. This reduces the number of distinct cases that must be understood and analyzed.

Object-oriented development not only allows information to be shared within an application, but also offers the prospect of reusing designs and code on future projects. Object-orientation is not a magic formula to ensure reusability, however. Reuse does not just happen; thinking beyond the immediate application and investing extra effort in a more general design must plan it.

Emphasis on Object Structure, Not Procedure Structure

Object-oriented technology stresses what an object is, rather than how it is used. The use of an object depends on the details of the application and frequently changes during development. As requirements evolve, the features supplied by an object are much more stable in the long run. Object-oriented development places a greater emphasis on data structure and a lesser emphasis on procedure structure than traditional functional-decomposition methodologies. In this respect, Object-oriented development is similar to information modeling techniques used in database design, although object-oriented development adds the concept of class-dependent behavior.

The Object Modeling Technique (OMT)

Object Model:

The Object model describes the structure of objects in a system – their identity, their relationships to other objects, their attributes, and their operations. The object model provides the essential framework into which the dynamic and functional models can be placed. Objects are the units into which we divide the world, the molecules of our models.

The object model is represented graphically with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. Classes define the attribute values carried by each object instance and operations that each object performs or undergoes.

Dynamic Model:

The dynamic model describes those aspects of a system concerned with time and the sequences of operations-events that mark changes, sequences of events, states that define the context for events, and the organization of events and states. The dynamic model captures control, that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for one class of objects. State diagrams also refer to the other models. Actions in the state diagrams correspond to functions from the functional model; events in a state diagram become operations on objects in the object model.

Functional Model:

The functional model describes those aspects of a system concerned with transformations of values- functions, mappings, constraints, and functional dependencies. The functional model captures what a system does, without regard for how or when it is done.

The functional model is represented with data flow diagrams. Data flow diagrams show the dependencies between values and the computation of output values from input values and functions, without regard for when or if the functions are executed. Traditional computing concepts such as expression trees are examples of functional models, as are less traditional concepts such as spreadsheets. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the 0object model.

Relations among Models

Each model describes one aspect of the system but contains references to the other models. The object model describes data structure that the dynamic and functional models operate on. The operations in the object model correspond to events in the dynamic model and functions in the functional model.

The dynamic model describes the control structure of objects. It shows decisions, which depend on object values, and which cause actions that change object values and invoke functions.

The functional model describes functions invoked by operations in the object model and actions in the dynamic model. Functions operate on data values specified by the object model. The functional model also shows constraints on object values.

Object Modeling

An object model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects. The object model is the most important of three models. We emphasize building a system around objects rather than around functionality, because an object-oriented model more closely corresponds to the real world and is consequently more resilient with respect to change. Objects models provide an intuitive graphic representation of a system and are valuable for communicating with customers and documenting the structure of a system.

OBJECTS AND CLASSES

The purpose of object modeling is to describe objects. An object is simply something that makes sense in an application context.

We define an object as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no one correct representation.

All objects have identity and are distinguishable. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look same. The term *identity* means that objects are distinguished by their inherent existence and not by descriptive properties that may have.

The word *object* is often vaguely used in literature. Sometimes object means a single thing, other times it refers to a group of similar things. Usually the context resolves

any ambiguity. When we want to be precise and refer to exactly one thing. We will use the phrase *object instance*. We will use the phrase *object class* to refer to a group of similar things.

CLASSES

An object class describes a group of objects with similar properties (attributes). Common behavior (operations), common relationship to other objects, and common semantics. Person, company, animal, process, and window are all object classes. Each person has an age, IQ, and may work at a job. Each process has an owner, priority, and list required resources, Objects and object classes often appear as nouns in problem descriptions.

The abbreviation class is often used instead of object class. Objects in a class have the same attributes and behavior patterns. Most objects derive their individuality from differences in their attribute values and relationships to other objects. However, objects with identical attribute values and relationships are possible.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior. Thus even though a barn and a horse both have a cost and age, they may belong to different classes. If barn and horse were regarded as purely financial assets, they may belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes. The interpretation of semantics depends on the purpose of each application and is a matter of judgment.

Each object "knows" its class. Most object-oriented programming languages can determine an object's at run time. An object's class is an implicit property of the object.

If objects are the focus of object modeling, why bother with classes? The notion of abstraction is at the heart of the matter. By grouping objects into classes, we abstract a problem. Abstraction gives modeling its power and ability to generalize

LINKS AND ASSOCIATIONS

Links and association are the means for establishing relationships among objects and classes.

A *link* is a physical or conceptual connection between object instances. For example Joe Smith *works-for* Simplex Company. A link is an instance of an association.

An *association* describes a group of links with common structure and common semantics. For example, a person *Works-for* a company. All the links in an association connect objects from the same class.

Associations are inherently bi-directional. The name of a binary association usually reads in a particular direction, but the binary association can be traversed in either direction. The direction implied by the name is the *forward* direction; the opposite direction is the inverse direction. For example, Works-for connects a person to a company. The inverse of works-for could be called Employs, and connects a company to a person.

Associations are often implemented in programming languages as pointers from one object to another. A *pointer* is an attribute in one object that contains an explicit reference to another object. For example, a data structure for Person might contain an attribute employer that points to a Company object, and a Company object might contain an attribute employees that points to a set of Employee objects.

A link shows a relationship between two (or more) objects. Modeling a link as a pointer disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched pointers, such as pointer from

Person to Company and the pointer from Company to a setoff Employee, hides the fact that the forward and inverse pointers are dependent on each other, even in designs for programs. Figure 5.2 shows a one-one association and corresponding links. Each association in the class diagram correspondence to a set of links in the instance diagram, just as each class correspondence to a setoff objects. Each country has a capital city. Has-capital is the name of the association. The OMT notation for an association is a line between classes. A link is drawn as a line between objects.

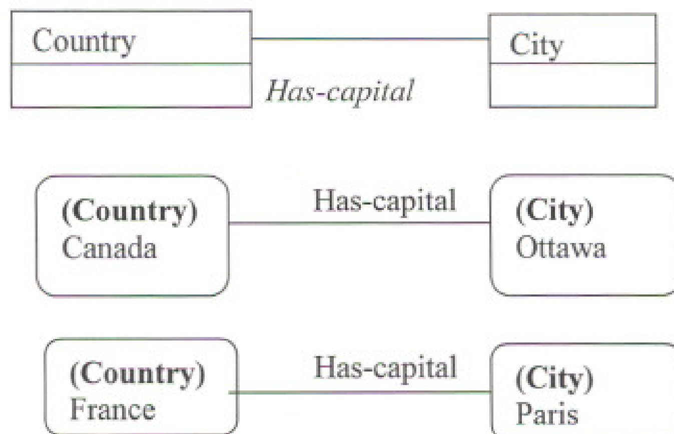


Fig. 5.2 (One-to-one association and links)

Associations may be binary, ternary, or higher order. Higher order associations are more complicated to draw, implement, and think than associations and should be avoided if possible.

Figure 5.3 shows a ternary association: Persons who are programmers use computer languages on projects. This ternary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The OMT symbol for general ternary and n-ary associations is a diamond with lines connecting to related classes. The name of association is written next to the diamond.

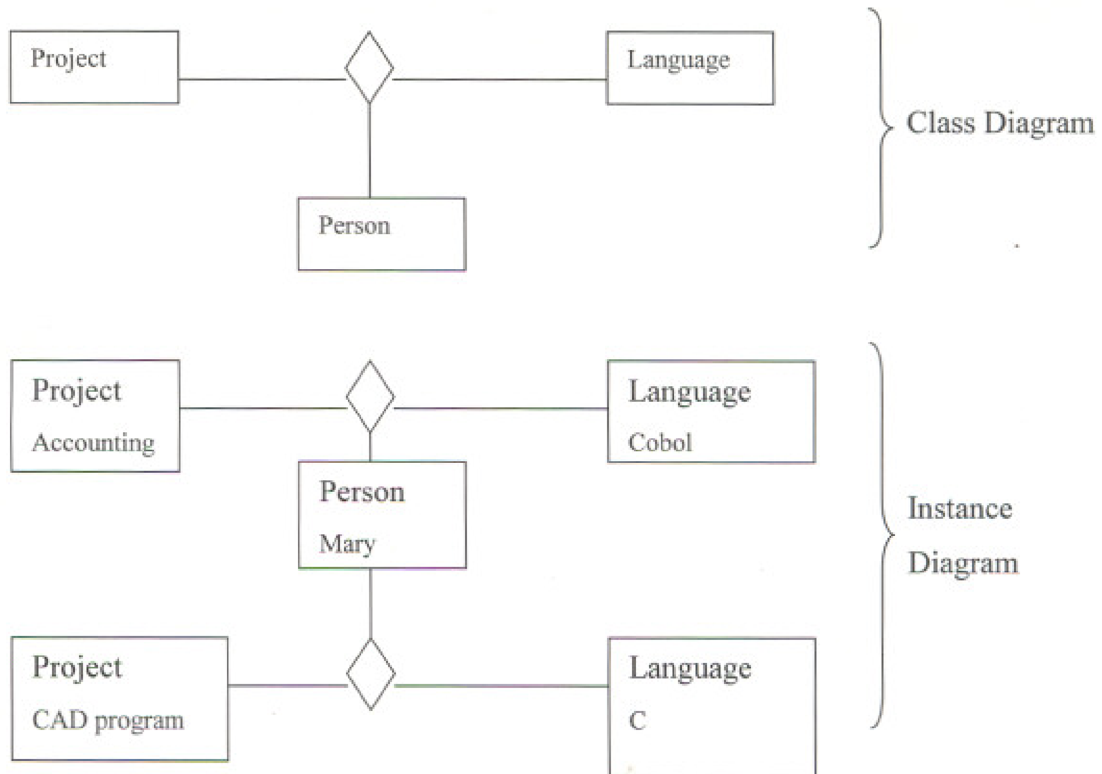


Fig 5.3 (Ternary association and Links)

Multiplicity

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. Multiplicity is often described as being "one" or many and can be described with a number or a set of intervals, such as "1", "1+"(one or more), "3-5" (three to five) and "2,4,18"(two, four or eighteen). A solid ball is the OMT symbol for "many", meaning zero or more. A hollow ball indicates "optional", meaning zero or one. A line without multiplicity symbols indicates a one-to-one association.

Figure 5.4 shows zero-or-one, or optional, multiplicity. A workstation may have one of its windows designed as the console to receive general error messages. It is possible, however, that no console window exists.

Fig 5.4 (Zero–or-one multiplicity)

Link Attributes

An attribute is a property of the objects in a class. Similarly, a *link attribute* is a property of the links in an association. Fig shows *access permission* is an attribute of *Accessible by*. Each link attribute has a value for each link. The OMT notation for a link attribute is a box attached to the association by a loop; one or more link attributes may appear in the second region of the box. This notation emphasizes the similarity between attributes for objects and attributes for links.

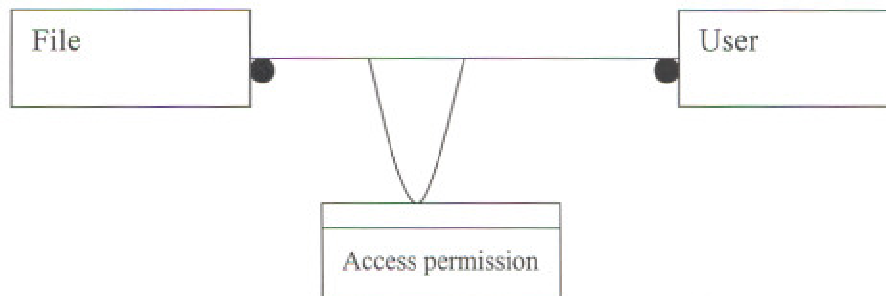


Fig 5.5 (Link attribute for a many-to-many association)

Figure 5.6 presents link attributes for two many-to-one associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Link attributes may also occur for one-to-one associations.

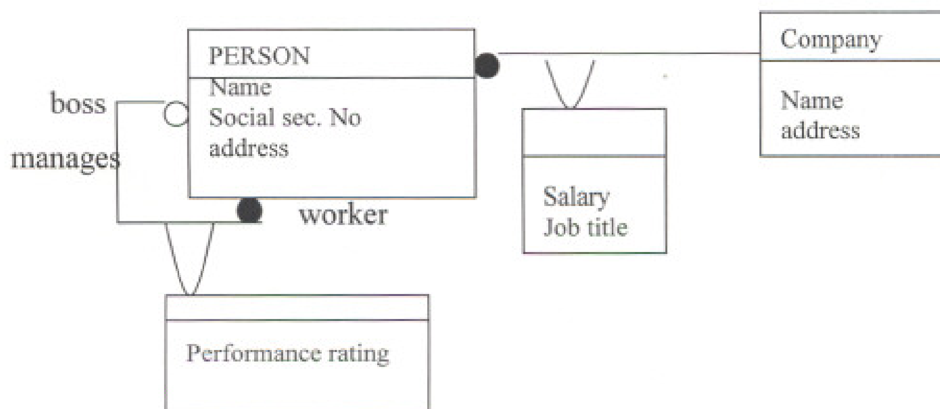


Fig 5.6 (Link attributes for one-to-many associations)

Ordering

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. For example, fig 5.7 shows a workstation screen containing a number of overlapping windows. The windows are explicitly ordered, so only the topmost window is visible at any point on the screen. The ordering is an inherent part of the association. Writing "(ordered)" next to the multiplicity dot for the role indicates an ordered set of objects on the "many" end of an association.

Fig 5.7 (Ordered sets in an association)

Qualification

A *qualified association* relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. The qualifier distinguishes among the set of objects at the many end of an association. A qualified association can also be considered a form of ternary association.

For example in the figure 5.8 a directory has many files. A file may belong to a single directory. Within the context of a directory, the file name specifies a unique file. Directory and file are object classes and file name is the qualifier. A directory plus a file name yields a file. A file corresponds to a directory and file name. Qualification reduces the effective multiplicity of this association from one-to-one. A directory has many files, each with a unique name.

Fig 5.8 A qualified association

Aggregation

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. For example, a name, argument list, and a compound statement are part of a C-language function definition, which in turn is part of an entire program. The most significant property of aggregation is transitivity, that is, if A is part of B and B is part of C, then A is part of C. Aggregation is also antisymmetric, that is, if A is part of B, then B is not part of A.

Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship. Fig 5.9 shows a portion of an object model for a word processing program. A document consists of many paragraphs, each of which consists of many sentences.

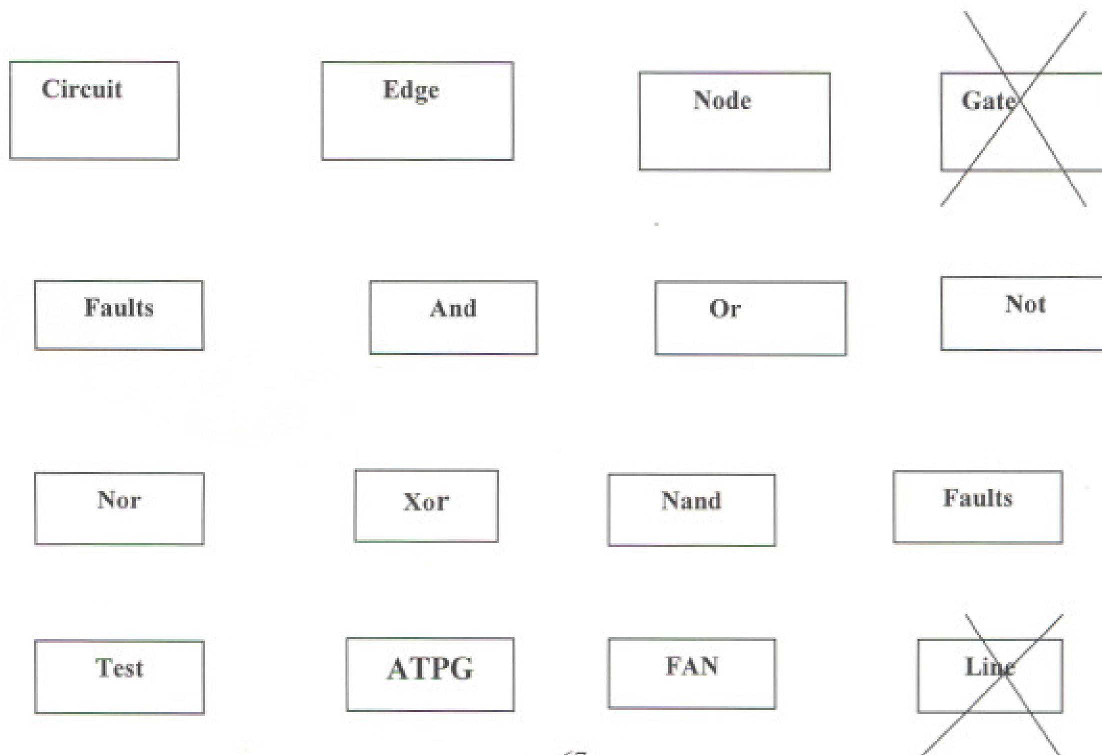
Fig 5.9 Aggregation

The following steps are performed in constructing an object model for ATPG:

- Identify objects and classes.
- Prepare a data dictionary
- Identify associations between objects
- Identify attributes of objects and links
- Organize and simplify object classes using inheritance
- Verify that access paths exist for likely queries.
- Iterate and refine the model
- Group classes into modules

Identification of Classes

This is the first step in constructing an object model. In this step the relevant object classes are identified from the application domain.



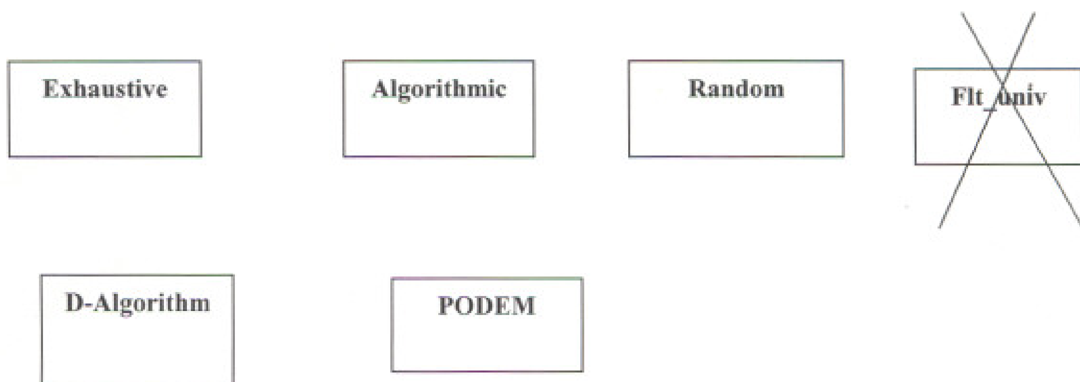


Fig 6.1 (ATPG classes extracted from the problem statement nouns)

Keeping the Right Classes

In this section the unnecessary and incorrect classes are discarded. Since the class Gate express the same information as is by Node class. Node class is more descriptive so we eliminate the Gate class from the list of classes in the ATPG problem(marked with cross in diagram). Similarly the redundant classes Line, flt_univ are being eliminated from the ATPG classes as shown in the above diagram with cross mark.

Preparation of a Data Dictionary

In this section we write a paragraph precisely for each object class, which describe the scope of the class within current problem. The data dictionary also describes associations, attributes, and operations. Below is the data dictionary for the classes in the ATPG problem.

ATPG: automatic test pattern generation refers to a technique that, given a model of a system, can generate test patterns for it.

Exhaustive: This is one of the test generation technique used when number of primary inputs are small, application of all possible input vectors ensures 100 percent fault coverage.

Algorithmic: The Algorithmic approach uses the concept of path sensitization for the VLSI test generation. Path sensitization is a process of generating a path along which the Fault effect is propagated.

Random: This is another test generation technique for VLSI testing. In this tests are generated through a random process.

D-Algorithm: The first algorithmic approach for VLSI testing. Its basic operation is the repeated intersection of the D-cubes necessary to perform the tasks required to test for a specific fault. These tasks consist of three operations: *fault sensitization, fault propagation, and fault justification.*

PODEM: The PODEM test algorithm is an implicit enumeration algorithm in which all possible primary input patterns are implicitly, but exhaustively, examined
As tests for a given test.

FAN: FAN is a complete algorithm in that it will generate a test if one exists. This includes improvements over PODEM to increase its efficiency. The major goal of FAN is to reduce the number of backtracks in the search tree.

Test: A Test is an input (a vector) that will produce different outputs in presence and absence of fault; they make the fault observable at the primary output.

Circuit: A combination of different Nodes and edges. It may have one or more faults.
Fault: A fault is a stuck value which may be D or D(bar). A D value signifies a logic value of 1 in the good circuit and 0 in the faulty circuit (0/1). A D (bar) value signifies a value of 0 in the good circuit and 1 in the faulty circuit (1/0).

Node: A node is a logic gate which may be a AND, OR, NOT, XOR, NOR, NAND gate.

Edge: An edge is a line of connection among different nodes. An edge may have values 0,1,D, D, X.

Free line: is a circuit node that has no predecessors that are part of a fan-out loop. As such, free lines may have a uniquely assigned value.

Bounded Lines: opposite of free lines.

Head Lines: These are free lines that drive a gate that is a part of Reconvergent fan-out loop.

Identification of Associations

Any dependency between two or more classes is an association.

Circuit includes nodes, edges, primary inputs and primary outputs.

Nodes are aggregate of AND, OR, NOR, NAND, NOT etc. of logic gates.

An edge may be stuck at a fault.

An edge may be a free, bound or a headline.

A fault may be D or D (bar).

A circuit may have many faults.

A test vector detects one or more faults.

A test is a sequence of test vectors.

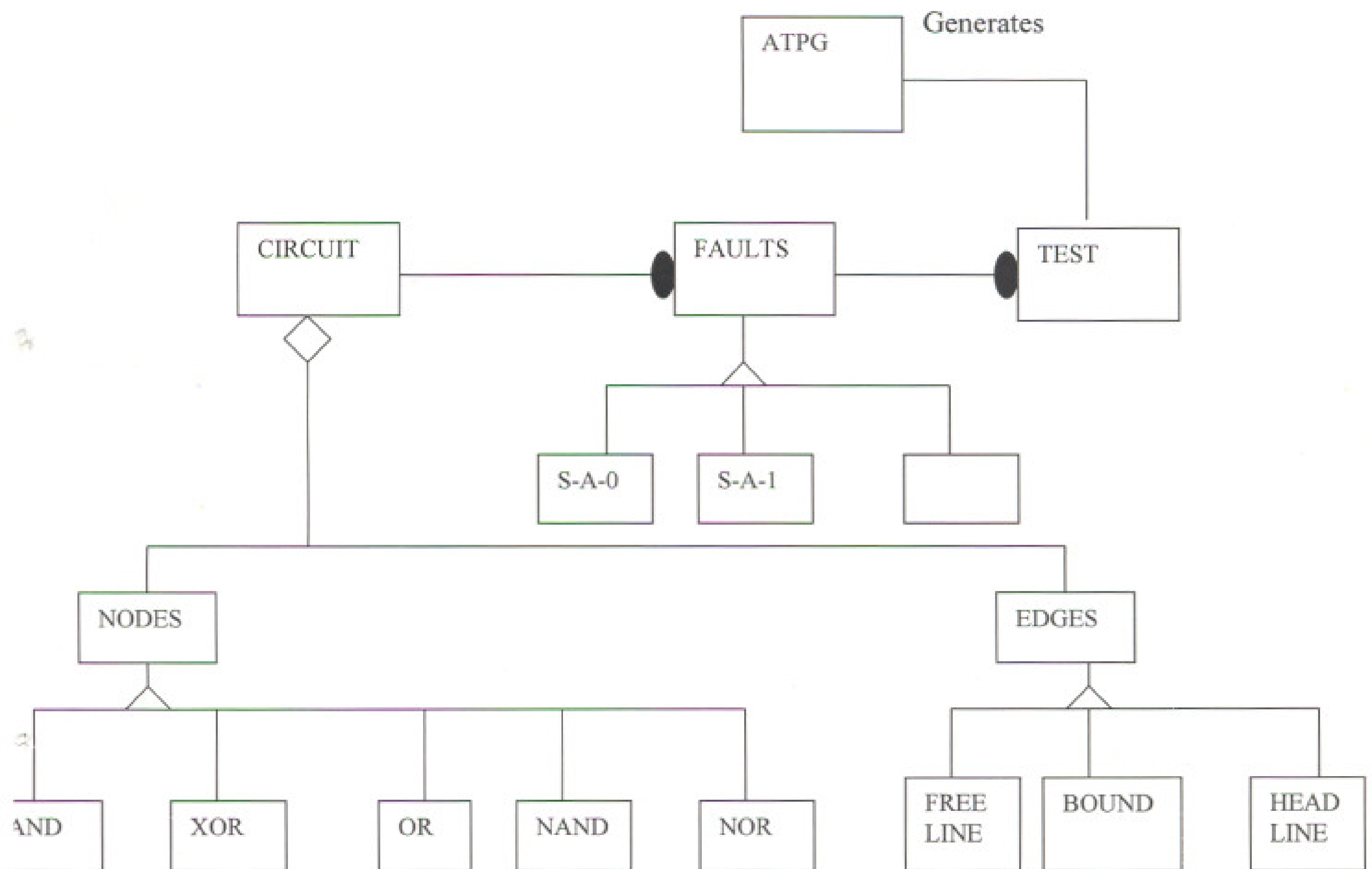
ATPG generates test.

ATPG is aggregate of exhaustive, algorithmic and random test techniques.

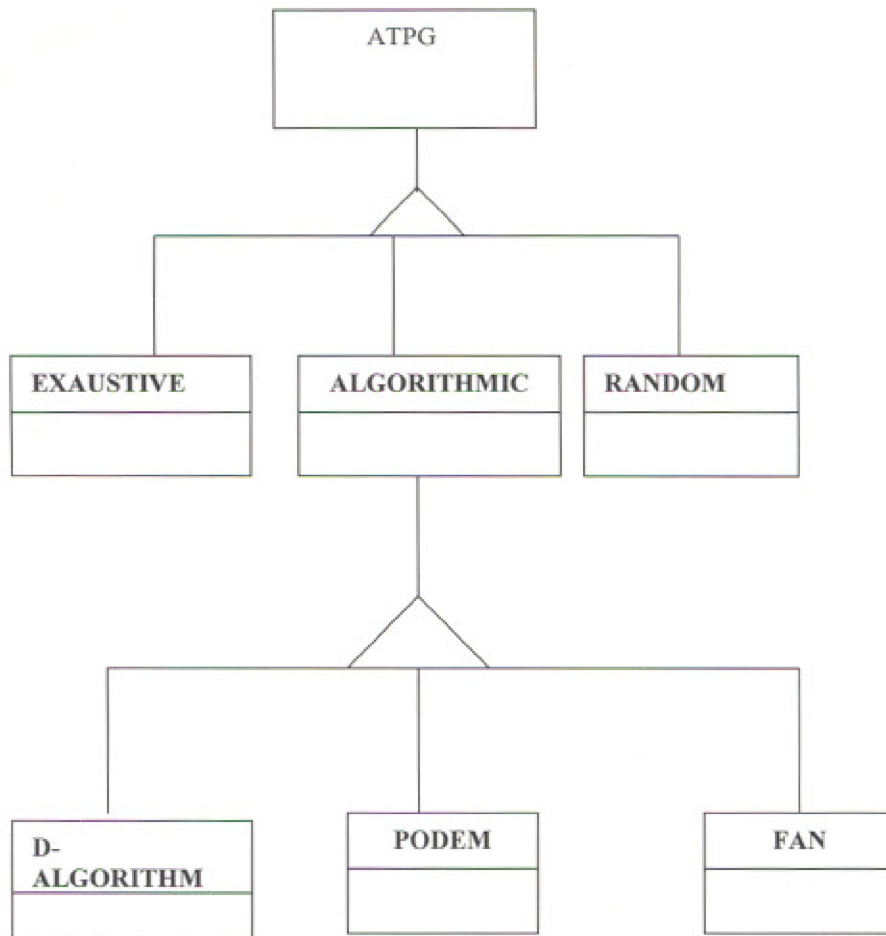
Algorithmic technique is the aggregate of D-algorithm, PODEM and FAN algorithms.

Keeping the right Associations

Any of the unnecessary and incorrect associations is discarded.



Sheet 1: Initial object model of ATPG system

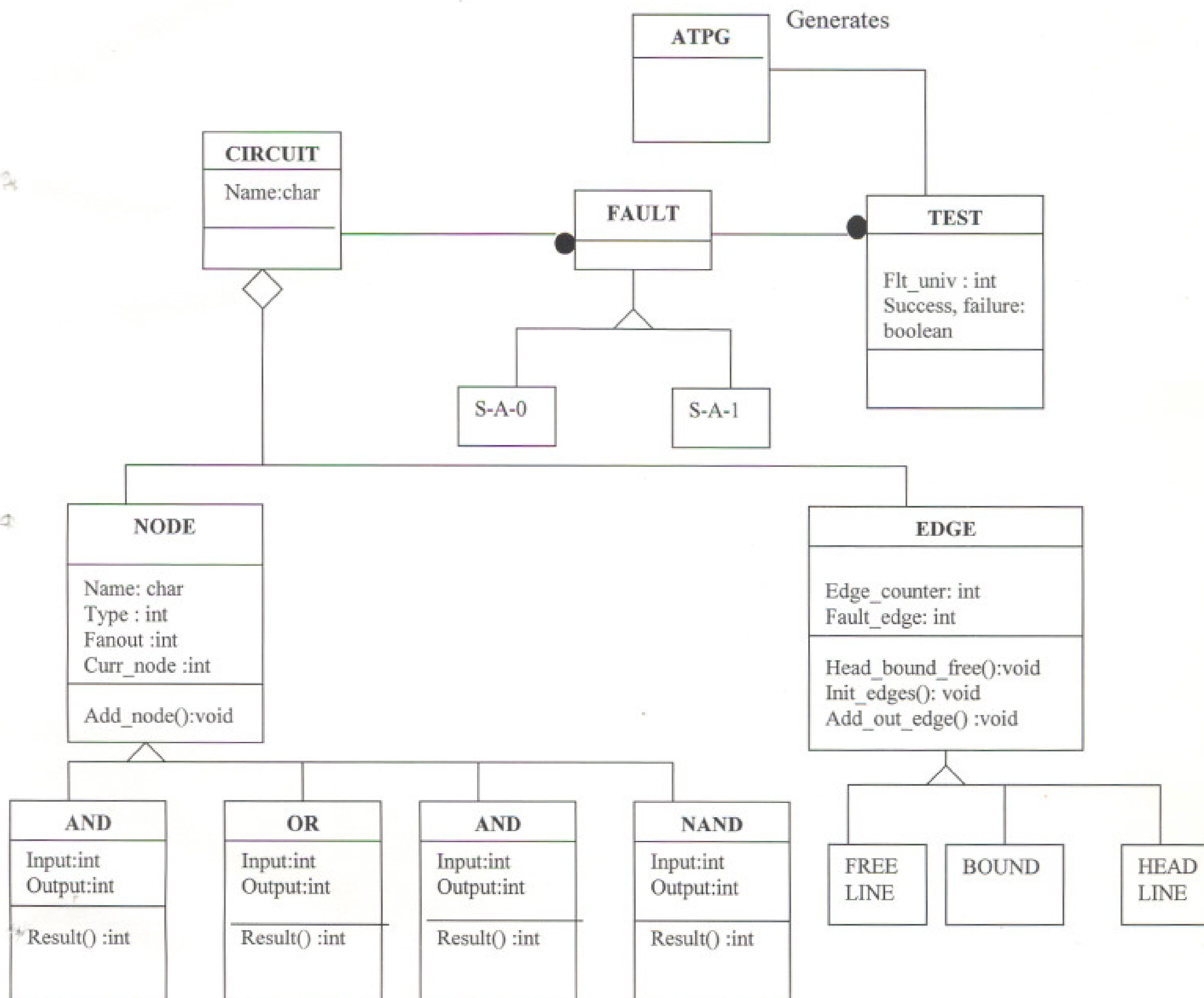


Sheet2: Initial object diagram for ATPG system

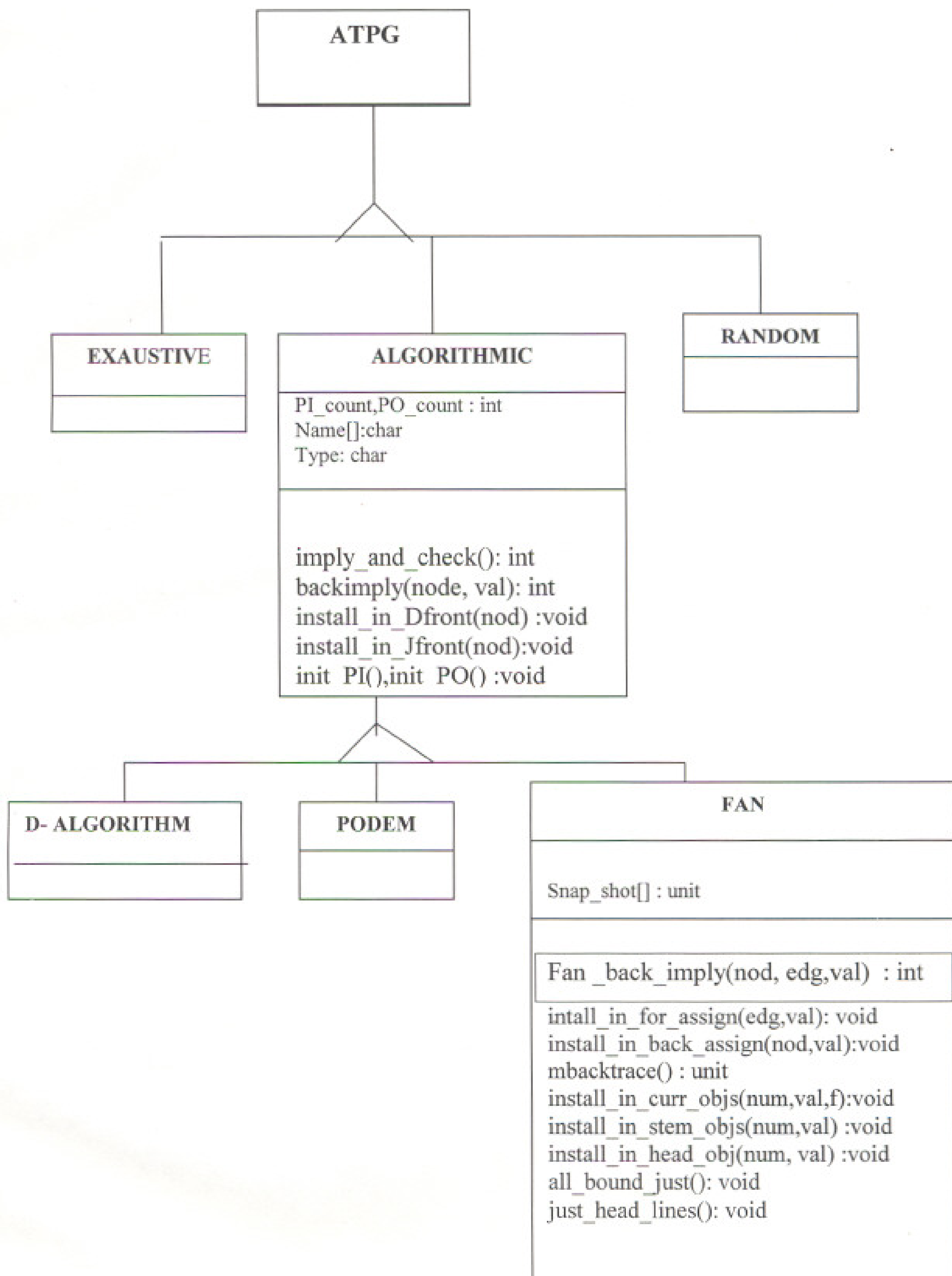
Identification of Attributes and operations

Next we identify object attributes. Attributes are properties of individual objects, such as name, weight, velocity, or color. Attributes should not be objects; use an association to show any relationship between two objects.

Unlike classes and associations, attributes are less likely to be fully described in the problem statement.



Sheet 3: Object model of ATPG with attributes and inheritance.



Sheet 4: Object model of ATPG with attributes and inheritance

Conclusion and future scope of work

Algorithmic automatic test generation for combinational circuit is an active area of research in test generation for digital systems. In the report various test generation algorithms have been discussed and the common concepts employed are explained in detail.

Various parallel processing techniques of ATPG have been explained which further reduces the computation time.

Object oriented concepts have been discussed and object-modeling technique has been used for analysis and design of the ATPG. FAN algorithm is modeled into different classes, which takes the benefit of object oriented analysis and design techniques.

The OO approach presented for the algorithmic ATPG in the work can be implemented by using any of the object-oriented languages, which will develop efficient and reliable and extensive CAD tools for the VLSI testing. Also this approach can be used for the parallel processing techniques presented in the work, which will further improve the CAD tools.

References

1. P. Goel, " An Implicit Enumeration Algorithm to Generate Test For Combinational Logic Circuits ",IEEE Trans. Comp., Vol. C-30, pp 215-222, March 1981.

2. H. FUJIWARA and T. SHIMDNO, "The Acceleration of test Generation Algorithms", IEEE Trans. Comp., Vol. C32, pp 1137-1144, Dec, 1988.

3. L.H. GOLDSMITH and E.L. THIGPEN, " SCOAP:…", Proc 17th Des Auto Conf., Minnneapolis MN, pp 190-196, June, 1980

4. J. SAVIR, G.S. DILLON and P.H. BARDELL, " RANDOM Pattern Testability", IEEE Trans. Comp. ,Vol. C-33, pp 79-90, Jan 1984.

5. S.C. SETH, L.PAN and V.D. AGARWL, " PREDICT:….",F.T.C.S. –15, Digest of papers, Ann Arbor, MI, pp 220-225, June 1985.

6. E Balagurusamy, "Object-Oriented Programming with C++", Tata McGraw-Hill Publishing.

7. James Rumbaugh, Michael Blaha" Object-Oriented Modeling and Design",Prentice Hall Publication.