COPERNICUS JEP 9624 FUTEG
Functional Test Generation and Diagnosis
(1994-97)

Final Report

Contents

1. Introduction

2. The Model Used in Test Generation and Fault Simulation Experiments

   2.1. Description of digital systems by alternative graphs
   2.2. Synthesis of alternative graphs
   2.3. Fault model used in experiments
   2.4. General fault notation for alternative graphs

3. Test Generation and Fault Simulation Approach Used in Experiments

   3.1. Test generation for the data path of digital systems
   3.2. Test generation for the control path of digital systems
   3.3. General structure of the automated test pattern generator

4. Experimental Results

   4.1. Low-level test generation experiments
   4.2. Test generation experiments with control units
   4.3. Test synthesis for digital systems with global feedbacks
   4.4. Test generation for RISC architectures
   4.5. Fault simulation experiments with BIST architectures

5. References

1. Introduction

Test generation for digital systems encompasses three activities:

- selecting a description method,
- developing a fault model and
- generating tests to detect the faults covered by the fault model.

The efficiency of test generation (quality, speed) is highly depending on the description method and fault models which have been chosen. As the complexity of digital systems continues to increase, the gate level test generation methods have become obsolete. Promising approaches are high-level and hierarchical methods which use behavioral, functional or multi-level descriptions of systems. However, the drawback lies in the need of different languages and models for different levels.

High-level test generation is based on the fact that high-level modules are treated as primitives [1-4]. In [5], these modules are still flattened till gate level during test generation, which may be considered as a drawback of the method. In [5-7], only combinational primitives are considered. The system partitioning into the data and control path is presented in [1,2,4,8], and interactions between them are taken into account during test generation. In high-level test synthesis, two different methods are distinguished: test generation using symbolic simulation [9] and test generation using path sensitisation technique [1, 3, 4, 8]. The first one generates a test by comparing the results of a symbolic simulation of a fault-free and faulty models and leads to analysis of symbolic expressions, which can be difficult for complex descriptions.

In hierarchical testing, top-down and bottom-up strategies are known. In the bottom-up approach, pre-calculated tests for system components (modules) generated on low-level will be later assembled at a higher abstraction level. Such algorithms typically ignore the incompleteness problem: constraints imposed by other modules and/or the network structure may prevent all test vectors from being assembled. Top-down approach has been proposed to solve this problem by deriving environmental constraints for low-level solutions. However, such techniques are of little use where the system is still under development in a bottom-up fashion, or where “canned” parts have to be applied.

For high-level and hierarchical test generation, different functional fault models have been introduced. In the case of microprocessors, individual functional fault models and

corresponding test strategies have been developed for different function classes like register decoding, instruction decoding, control, data storage, data transfer, data manipulation etc. [10]. The main disadvantage of this approach is that only microprocessors are handled and the results obtained cannot be extended to cover the general digital systems test problem. When using register transfer languages (RTL-approach), a formal definition of a RTL statement is defined and nine categories of functional faults for components of RTL statements are identified [11]. Some attempts to develop special functional fault models for different data-flow network units like decoders, multiplexers, memories, PLAs etc. are described in [12]. In the last period, a lot of attention has been devoted for generating tests directly from descriptions in high level languages [13,14]. All the listed approaches lead to using different mathematics and procedures for each fault model. The diversity of fault types makes it difficult to develop uniform test generation algorithms with possibility to treat all faults by standard procedures as in the case of stuck-at faults at the gate-level approach. Test synthesis based on a lot of different types of fault models will be more complicated compared to the case when only one generic fault model is used.

In the approach developed in framework of the FUTEG project, a method for describing digital systems and for modeling faults based on alternative graphs (AG) [15] is used. AGs serve as a basis for a general theory of test design for mixed-level representations of systems, similarly as we have the Boolean algebra for the plain logical level. The fault model defined on AGs represents a generalization of the classical gate-level stuck-at fault model - the latter was defined for literals in Boolean expressions whereas the former is defined for nodes in AGs.

Logical level AGs [15] represent the topology of gate-level circuits, and therefore, unlike analogous BDDs [16], they directly support test generation for gate-level structural faults without representing these faults explicitly. Moreover, AGs and BDDs are based on the same binary graph ideology, hence, the same theoretical basis can be used for mixing symbolic and topological techniques. This can simplify further developments and improvements of the method. On the other hand, AGs support uniform approach to test design at different system levels, whereas BDDs support only the Boolean level.

In the project, a method was implemented for jointly describing structural properties, functions, and faults of digital systems by decision diagrams. The new fault model covers a large class of faults including the register transfer level (RTL) functional fault class [17,18] and topological fault classes like stuck-at, delay and bridging faults. The joint information about structure and functions presented in AGs helps to create uniform procedures for generating local test patterns for components, as well as for generating I- or F-paths [19,20] through the network. The method supports both, high-level (behavioral) and hierarchical (or mixed-level) test generation schemes. Differently from known approaches, control and data parts are handled in the uniform way. The model provides an efficient theoretical basis to combine high-level approaches to test generation with BDD-based symbolic and topological techniques on low-level. Experimental results are provided in this report for demonstrating the efficiency of using decision diagrams in test generation.

2. The Model Used in Test Generation and Fault Simulation Experiments

2.1. Description of digital systems by alternative graphs

Alternative graph (AG) in general case is defined as a non-cyclic directed graph whose nodes are labelled by variables. There can be different finite sets of values the variables can take. In special cases, variables can also be substituted by constants or by algebraic expressions. The graph has only one starting node (root). The number of terminal nodes, i.e. nodes for which successor nodes are missing, is not limited. For each value from a set of predefined possible values of a non-terminal node variable (or expression), there exists one and only one output branch from the node. Different branches of the same node may lead into the same successor node.

Consider a situation where all node variables are fixed to some value. By these values, for each non-terminal node a certain output branch will be chosen which enters into its corresponding successor node. Let us call such connections between nodes *activated branches* under the given values. Succeeding each other, activated branches form in turn *activated paths*. For each combination of values for all node variables there exists always a corresponding activated path from the starting node to some terminal node. Let us call this path the *main activated path*. Now, for each combination of values for all node variables there exists one and only one value which is equal to the value of the variable (or expression) at the terminal node of the main activated path. This relationship describes a mapping from a Cartesian product of the sets of values for variables in all nodes to the joint set of values for variables in the terminal ones. Therefore, by AGs it is possible to represent arbitrary digital functions $Y = F(x)$, where Y is the variable whose value will be calculated on the AG and x is the vector of all variables which belong to the labels of the nodes in the AG.

2.1.1. Alternative graphs and gate-level combinational circuits

An AG that represents a Boolean function (binary decision diagram [16]) is a directed noncyclic graph with single root node, where all nonterminal nodes are labelled by (inverted or not inverted) Boolean variables (arguments of the function) and have always exactly two successor-nodes whereas all terminal nodes are labelled by constants 0 or 1. For all nonterminal nodes, an one-to-one correspondence exists between the values of the label variable of the node and the successors of the node. This correspondence is determined by the Boolean function inherent to the graph.

Unlike the traditional BDDs [16], structural structural AGs (SAG) reported in [15] support structural representation of gate-level networks in terms of signal paths. By superposition procedure described in [15], we create SAGs where one-to-one correspondence exists between graph nodes and signal paths in tree-like subciruits represented by SAGs. We can consider a digital circuit as a network of tree-like subcircuits, each of them represented by an equivalent parenthesis form (EPF). Consequently, a digital circuit can be represented by a system of SAGs.Using SAGs, it is possible to ascend from the gate-level descriptions of

circuits to higher level descriptions without loosing accuracy of representing gate-level signal paths.

Denote the literal which labels a node m in a SAG by x(m). We say that a value of the node variable activates the node output branch. According to the value of x(m), one of two output branches of m will be activated. A path in a SAG is called activated if all the branches that form this path are activated. The SAG is called activated to the value 0 (or 1) if there exists an activated path which includes both the root node and the terminal node labelled by the constant 0 (or 1). A SAG G_y with nodes labelled by literals $x_1, x_2, ..., x_n$, represents an EPF y = f(X) = f($x_1, x_2, ..., x_n$), if for each pattern of X, the SAG will be activated to the value which is equal to y.

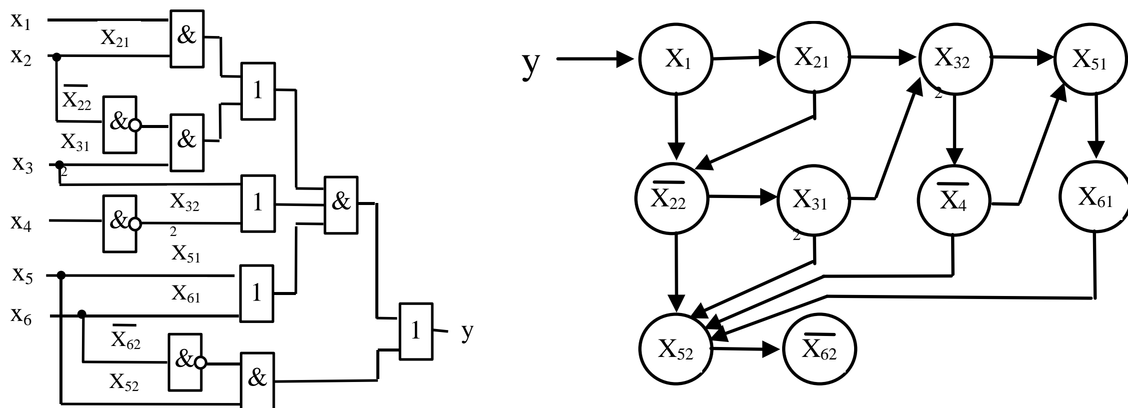

Fig.1. Combinational circuit and structural BDD

As an example, Fig.1. shows a representation of a combinational circuit by a SAG which corresponds to the EPF

$$y = (x_1\, x_{21} + \neg x_{22}\, x_{31})\, (x_{32}\, x_{51} + \neg x_4\, x_{61}) + x_{52} \neg x_{62}.$$

For simplicity, values of variables on branches are omitted (by convention, the right-hand branch corresponds to 1 and the lower-hand branch to 0). Also, terminal nodes with constants 0 and 1 are omitted (leaving the graph to the right corresponds to y = 1, and down - to y = 0). The graph contains 10 nodes, and each of them represents a signal path in a circuit (and a literal in the EPF). The literals in EPF and the related node variables in the graph correspond to input branches of the circuit in Fig.1.

2.1.2. Alternative graphs and finite state machines

There are two ways of representing finite state machines (FSM): 1) structural way - by a circuit which can be decomposed into combinational and memory parts, and 2) functional way - by state transition diagrams (STD).

In the first case, there is no principal difference in using AGs for representing FSMs compared to the case of combinational circuits. The output and transition functions of the FSM are Boolean and therefore can be represented by AGs for Boolean functions (or similar BDDs [15]). For the second case, we use integer variables for representing inputs, outputs and internal states of the FSM. A FSM is represented by AGs for describing, correspondingly, the transition and output behaviors of the machine. By introducing complex variables and representing the FSM by a single complex function q.y = f(q',x), where state variable q and output variable y are concatenated, we can represent a FSM by a single AG. As an example, in Fig.1 two representations of a FSM by a STD and by an AG are depicted. AG represents the complex behavior function of the FSM q.y = F(q', Res, x_1, x_2) where q.y is the concatenation of the integer state variable q (with possible values 1,2,3,4,5,6 for representing states) and the binary output variable y. The input of the FSM is structured and represented by three Boolean variables Res, x_1 and x_2. By q' we denote the previous state variable. Terminal nodes of the AG are labelled by complex (concatenated) constants which represent the new state of the FSM and the value of the output variable y at the new state. To be able formally to model the faulty behavior of the FSM, we have to specify in AGs if possible also the behavior of FSM at illegal states denoted by q = *. In the example in Fig.2, for illegal states, it has been assumed that y = 0.

Two extreme cases can be considered in representing FSMs by AGs:

1) the case of abstract FSM, where we have only three abstract variables for representing the input, output and internal states of the automata and, correspondingly, two AGs for representing the transition and output functions, and,

2) the case where the input, output and internal states of the FSM are binary coded and we can represent it by a set of Boolean output and transition functions. Mixed cases can be placed between these two extremes. By introducing complex variables (e.g. microinstruction words consisting of fields), and representing the FSM by a single complex function q.y = f(q',x), where state variable q and output variable y are concatenated, we can represent a FSM by a single AG.

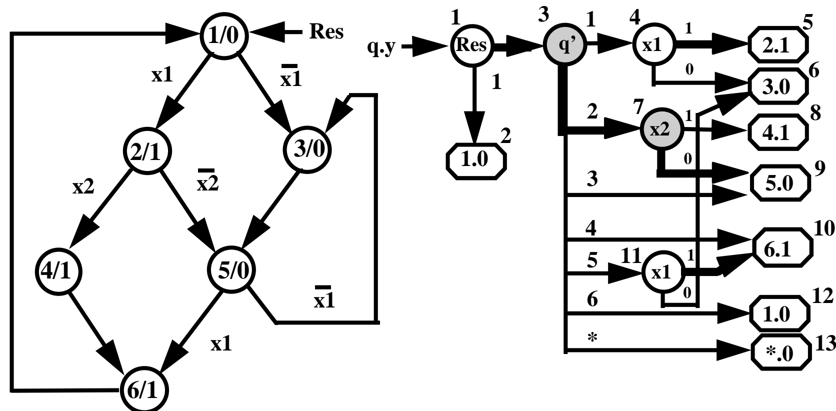

Fig.2 FSM representations by a STD and a functional level AG.

As an example, in Fig.2 two representations of a FSM by a state transition diagram and by an alternative graph are depicted. The AG represents the complex behavior function of the FSM q.y = F(q', Res, x_l, x_2) where q.y is the concatenation of the integer state variable q (with possible values 1,2,3,4,5,6 for representing states) and the binary output variable y. Input of the FSM is structured and represented by three Boolean variables Res, x_l and x_2. By q' we denote the previous state variable. Terminal nodes of the AG are labelled by complex (concatenated) constants which represent the new state of the FSM and the value of the output variable y at the new state. To be able formally to model the faulty behavior of the FSM, we have to specify in AGs also the behavior of the FSM at illegal states, denoted by q = *. In the example, for illegal states it has been assumed that y = 0.

The are two properties of AGs that essentially differ them from STD-s which, however, may be not noticed at a glance on the example:

-similarity in representation with Boolean AGs (BDDs) that allows to generalize methods developed for the logical level as well to the higher functional (state transition) level;

-in AGs, only one model in the form of graphs is used whereas STDs consist in two models - graphs for representing transitions between states and Boolean expressions to determine the branching conditions.

2.1.3. Alternative graphs and register transfer level data-paths

When using AGs to describe complex digital systems, we have, at the first step, to represent the system by a suitable set of interconnected components (combinational or sequential ones). At the second step, we have to describe these components by their corresponding functions which can be represented by AGs.

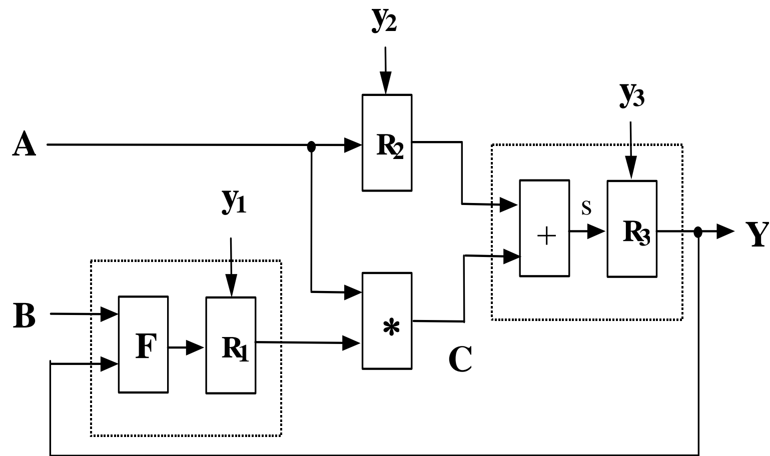

Fig.3. Register transfer level representation of a data-path

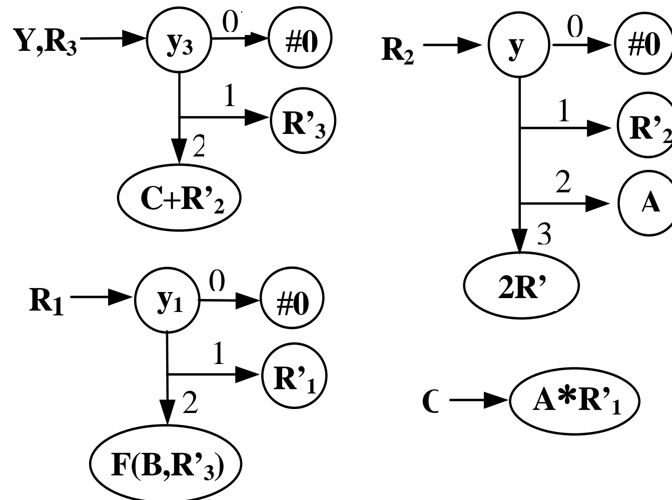

Fig. 4 AG-model for the datapath in Fig.1.

An example of representing a RTL description of a data-path in Fig.3 of a digital system by AGs is depicted in Fig.4., whereas the functions of components of the data-path are described in Table 1. The model consists of graphs G_{R2}, G_C, G_{R1} and G_{R3} for representing, correspondingly, the functions of the register R_2, multiplier C and of two sub-networks R_1 and R_3 surrounded by dotted lines

In this example, y_1, y_2, and y_3 serve as control inputs, A and B are data inputs, R_1, R_2, and R_3 serve as data register variables (by apostrophe the previous state is denoted), C is the output variable of the multiplier and input variable for the adder, and Y is the primary output of the datapath. In nonterminal nodes, only control variables are used as labels. Terminal nodes are labelled by data transfer or data manipulation expressions. Each node in the model has a strong relation to the structure of the datapath: nonterminal nodes represent the control logic

in modules (subnetworks), the nodes labelled by data variables represent buses, and the nodes labelled by expressions represent data manipulation logic. In such a way, to generate a test for a given node means to create a test for a related structural part of the module or subnetwork.

Table 1

y_1	R_1	Funktion	y_2	R_2	Funktion	y_3	Y, R_3	Funktion	C	Funktion
0	#0	Reset	0	#0	Reset	0	#0	Reset	A+ R'$_2$	Multiplier
1	R'$_1$	Hold	1	R'$_2$	Hold	1	R'$_3$	Hold		
2	F(B, R'$_3$)	Special	2	A	Load	2	C+ R'$_2$	Adder		
			3	2R'$_2$	Shift					

2.2. Synthesis of alternative graphs

Consider a digital system S = (F, N) as a set of components (or subnetworks) F represented by functions y = f(x) and a network N connecting these components. The system is represented by a set of variables Z = {IN, OUT, INT, REG} defined by relationships of component functions $f \in F$. Here IN, OUT, INT and REG represent correspondingly the sets of primary input, primary output, internal bus and system state (register) variables. The set of components can be devided into control part F_C and datapath F_D, $F = F_C \cup F_D$. Hence, we can distinguish in Z also control and data variables.

A set of AG-s $G_S = \{G_y\}$ represent a digital system S = (F,N) if for each function y=f(x) in F there exists a graph $G_y = (M,\Gamma,x)$. The set $G_S = \{G_y\}$ is called AG-model for the system S.

Note, in the AG-model we do not have the network N explicitly given. In the AG-model we suppose that two variables connected through the network N have the same name. In other words, the set $\{G_y\}$ of the AG-model represents a set of graphs connected by variables.

To generate AGs for components of the datapath, at first, the decision diagram for the control logic of the component should be created. If the binary level will be implemented in the AG-model, the methods for creating structural AGs [Uba 96b] can be used. In that case, each node in the graph will represent a signal path in the gate-level control logic. Hence, the structure of the control circuit will be represented in terms of signal paths in the model. If the RT-level graphs are to be created, we will consider higher level (integer) variables which represent control fields of instructions, microinstructions or control buses. By using the control variables, a decision diagram will be built up with one or more decision nodes in each path of the graph, and, in general, with more than two output edges from each decision node. To each decision node in the model, a multiplexer or decoder as the structural part of the module corresponds. After creation of the decision part of the AG, all the paths in the model will be terminated with nodes labelled by corresponding expressions for data transfer or data manipulation. In such a way, all the graphs in Fig.2 are synthesized.

As to the control part of the system, we generate a joint AG for the output and the next-state behaviour for each finite state machine (FSM). As labels for the decision nodes, input and

previous state variables of the FSM are used. Each pattern for these variables prescribes a path through the decision tree which should be terminated by a node labelled by expression (or constant) to define the next state and the output behaviour of the FSM. In Fig.5 and in Table 2, a FSM for controlling the data path in Fig.1 is depicted. As the result of cooperation of the control and data parts, multiplication of A and B is performed. To represent the FSM, an AG is created for the vector function: $q,y_1,y_2,y_3 = f(q',R'_2=0)$. The predicate $R'_2=0$ is used here to represent a flag variable for reporting the state of the datapath. We have two decision nodes in the graph for analysing the previous state q' of the FSM and the flag $R'_2=0$. The terminal nodes are labelled by constants (the values of the vector variable q,y_1,y_2,y_3).

Table 2

q'	x	q	y_1 y_2 y_3	Activities in the datapth
0		1	1 1 0	y_3:R_3 := 0
1		2	2 2 1	y_1:R_1 := B, y_2:R_2 := A
2	$R'_2=0$	0	1 1 2	y_3:R_3 := A∗B
2	$R'_2\neq0$	2	1 2 1	y_2:R_2 := A

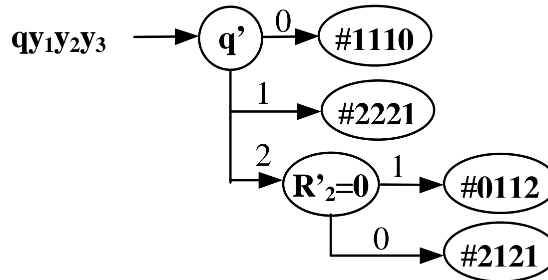

Fig. 5. AG-model for the control part

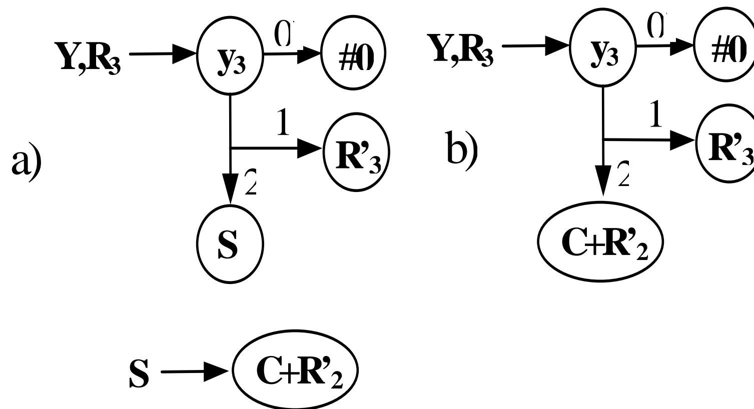

Fig. 6 Superposition of AGs for the data part

If the components of the system are represented by AGs, different manipulations with graphs are possible. For the compression of the model, the graph superposition procedure proposed in [15] for gate-level strucural graph synthesis can be generalized for the RT-level case. Since control signals are usually either primary inputs for the data-path or inputs from the control part, no superposition for nonterminal nodes is needed. The control part will be represented separately, and should not be mixed in the model with datapath. On the other hand, terminal nodes which represent data buses can be replaced by a graph which describes the component whose output is connected to the bus. Since the bus variable will disappear from the model, the fault collapsing event will take place as a side effect of superposition. In this way, both the complexity of the model as well the complexity of the fault set to be attacked will be reduced. An example of creating the AG for the subnetwork R_3 of Fig.3 (surrounded by dotted lines) is shown in Fig.6. The register with reset, hold and load functions and the adder connected to the register via an internal bus S is represented in a compressed model by a single AG.

Fault propagation modes are sometimes explicitly represented in the functional description of the component. For example, in Fig.6a the fault propagation mode for the register is explicitly given in the graph $G_{Y,R3}$. To propagate a fault from the input S to the output Y, the condition $y_3 = 2$ should be fulfilled, which activates the full path in $G_{Y,R3}$. On the other hand, e.g. the adder in Fig.6b is represented only by expression $C+R'_2$ and not by explicit fault propagation modes. In general, fault propagation modes are a subset of all the component functions, which is not always explicitly highlighted in the functional description. This subset is usually given in the form of tables as an additional information, where all possible conditions to build up transparency paths (I- or F-paths) are listed. All the conditions to build up transparency paths for a given output can be represented as a decision diagram, and hence, they can be merged with the AG-model to give a uniform representation to all of the important diagnostic information – to functions, structural details, faults and fault propagation modes of the component.

2.3. Fault model used in experiments

2.3.1. Faulty node as the basic fault model for AGs

Depending on different classes of digital systems (or different levels of representation) we can classify different types of AGs with nodes having different interpretations in relation to the structure of the system. The possibility of representing faults in AGs depends directly on these relationships. At the Boolean level, all nodes in structural AGs (SAG) are labelled by Boolean variables, whereas each node represents a signal path in the gate-level circuit. At the RT-level, the nodes in AGs are labelled by high-level variables like control fields in instructions (or microinstructions), data words or by data manipulation expressions. Different fault models defined at different system levels are replaced on AGs by the uniform fault model of a node.

Definition 1.

The fault model for AGs is defined as the following faulty behaviour of a node m labelled with a variable x(m) which can have values from a finite set V(m):

1) the output edge of the node m is always activated to x(m) = i (notation: x(m)/∅⇒ I);
2) the output edge of the node for x(m) = i is broken (notation: x(m)/ i ⇒∅);
3) instead of the given output edge of the node for x(m) = i, another edge or a set of edges for values $x(m) \in V_j(m) \subset V(m)$ are activated (notation: $z(m)/ i \Rightarrow V_j(m)$).

RTL statement: K: (T,C) R_D < --- F (R_{S1}, R_{S2}, ... R_{sm}), --- > N

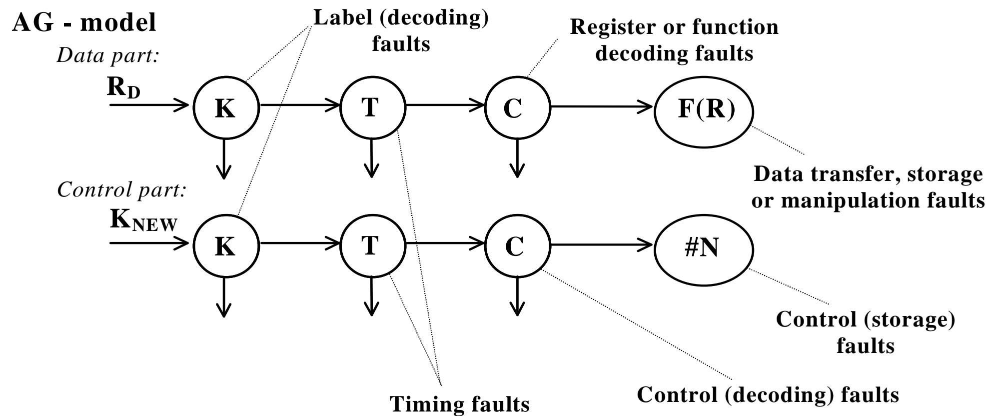

Fig.7. RTL faults in the AG-model

In fact, this fault model leads to exhaustive test of the functionality of the node. Depending on the abstraction level, the complexity of the fault model can be higher or lower. Each path in an AG describes the system behavior in a specific mode of operation. The faults having effect on the behavior are related to nodes along the given path. A fault will cause an erroneos change in the path activated by a test. The physical meaning of faults associated with node ouputs depends on the relationship between the node and the circuit. Depending on the adequacy of representing the system structure by AG-s, this fault model can cover a wide class of structural and functional faults of digital circuits and systems. In general, the fault model defined on AGs can be regarded as a generalization of the classical stuck-at fault model - the latter being defined for Boolean variables, the former - for AG nodes.

Consider some well-known low- and high-level fault models in terms of the AG notation. Since the nodes in binary AGs represent signal paths in a gate-level circuit, the two possible faults of the AG-node represent all the stuck-at faults along the corresponding signal path. The RTL statement faults like label, timing (or logical) condition, register (or function) decoding and control faults [18] are covered by faults of nonterminal nodes in AGs, labelled by corresponding label, condition, decoding or control variables. The data storage, transfer and manipulation faults [18] are covered by faults at terminal nodes labelled by corresponding data variables or data manipulation expressions. The different fault classes introduced in [17] for microprocessors are covered in the similar way by a single fault model of a node in AGs.

In Fig.7, the fault model of AGs for a general RTL statement is illustrated. A subgraph G_{RD} represents a decisions chain for the data manipulation mode F(R) of a component R_D in the data part, and the graph G_{KNEW} represents a decisions chain for the control part of a system to

assign the new state K_{NEW} = N. The following notation is used: K – label of the RTL statement (state variable of the system), T - timing condition (state variable for a FSM of the control part), C - logical condition (flag in the datapath, R_D - destination register, R_{Si} - source register, F(R) - operation (microperation) with the content of R, N - reference to the next statement (to the next state of the system).

Traditionally [17,18], for each component of a RTL-statement a fault model by enumerating the fault lists (lists of possible deviations for values of the component) is created. In the case of AGs, only a single general fault model of a node is used. The fault list is given implicitly, i.e. each fault can be derived from the AG-model directly if needed. The set of possible faulty deviations for a node variable z(m) is defined by V(m).

2.3.2. Representing faults in finite state machines

Different fault models for different representation levels of FSMs can be replaced on AGs by this uniform node fault model. The physical meaning of faults associated with a particular node depends on the “structural meaning” [15] of the node.

From above it follows that the fault model defined on AGs can be regarded as a generalization of the classical gate-level stuck-at fault model for more higher level representations of digital systems. As the nodes with Boolean labels represent only a special class of nodes in AGs, the logical level stuck-at fault model represents also only a special class of faults in AGs. In the following we consider how the different fault classes in finite state machines can be represented uniformly using alternative graphs.

2.3.2.1. Fault classes for finite state machines. Any irredundant structural fault in the implementation of the FSM will cause some changes in its STD. One or multiple transitions will be corrupted. So, a test sequence that detects all multiple transition faults will detect all irredundant permanent physical defects. However, the analysis of multiple transition faults is too complex, therefore usually a single transition fault will be considered. In the following, we try to find relationships between structural and functional level faults, to analyse how different single structural fault types affect the behavior of the FSM, are they manifesting himself as single or multiple transition faults, and how test sequences can be generated for them.

The faults of the FSM circuitry can be divided into the following fault classes: 1) single transition faults (class a) - faults that effect on a single transition condition only; 2) input faults (class b) - faults that effect on the input of the FSM; 3) state faults (class c) - faults that effect on the state of the FSM.

We shall show in, the following that a single fault in a FSM, represented by an iterative array of identical combinational circuits, can manifest himself in a test sequence in different ways: *as a single fault both* in each time frame and, under special restrictions, also in the whole array (transition faults), or *as a multiple fault* both in each time frame and in the whole array (input and state faults). From the different complexity of faults, it follows that the faults are to be processed during test generation by different strategies, e.g. to be processed at different

FSM representation levels. Using AGs, it will be possible to process the faults at different FSM levels by uniform algorithms.

2.3.2.2. Representing transition faults on gate-level AGs. The class *of transition faults* (not to mix up with functional faults related to branches in STDs, as used in [4]) is related exclusively to the circuitry which calculates the transition condition effect, provided that all condition signals are fault free. These faults are difficult to define at the functional level because of the implementation dependency. Assume that all the next state circuits for different flip-flops are disjoint. If it is not the case, the faults in joint parts of the logic shared for different flip-flops should be handled in the same way as the input and state faults. It is easy to notice that in the assumed case the transition faults influence always on a single transition condition only and therefore, they cannot mask themselves as long the same transition will not repeated. It means that as long not yet tested loops are not containing in test sequences, the faults of type (a) manifest himself as single faults in the whole iterative array related to the test sequence. This property gives the possibility to carry out the test synthesis on different levels without crossing the level borders if backtracking is needed. Particularly, the fault activization procedure will be carried out on the gate level where these faults are specified, whereas the signal justification (state initialization) and the fault propagation procedures can be carried out on the functional STD level. Transition faults can be concisely represented in structural AGs of the next- state logic.

2.3.2.3.. Representing transition faults on signal-path-level AGs. Generation of a compressed structural AG-model for a given gate-level digital circuit is based on the superposition of AGs [15]). AGs for logical gates are assumed to be given as a source library. Starting from the gate-level AG-description and using iteratively graph superposition, we can produce a more concise higher level representation of the circuit. As a result of this procedure, we create structural AGs (SAG) which have the following property [15]: each node in a SAG represents a signal path in the corresponding gate-level circuit. To avoid repeating in the AG-model same subgraphs, it is recommendable to create separate AGs for tree-like subcircuits. In this case, the number of all nodes in the set of AGs will be equal to the number of paths in all tree-like subnetworks of the circuit, and one to one correspondence will exist between paths in these subnetworks and nodes in AGs. Hence, using the concept of SAGs, it is possible to rise from the gate-level descriptions of digital circuits to higher level structural descriptions without of loosing accuracy of representing gate-level stuck-at faults. The task of simulating structural stuck-at-faults in a given path of a circuit can be substituted by the task of simulating faults at a node in the corresponding SAG.

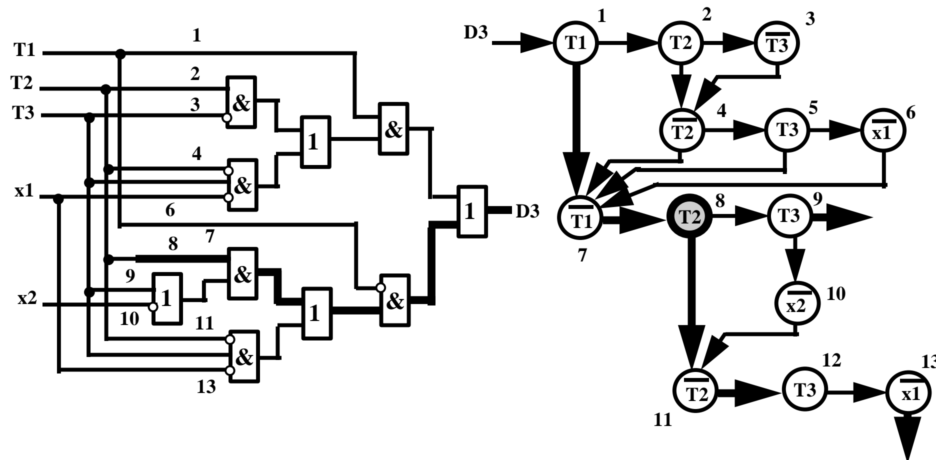

Fig.8. Representing a gate-level structure by a structural AG

An example of a SAG for a combinational circuit is depicted in Fig.8. The nodes of AG are labelled by input variables of the circuit. For simplicity, the values of variables on branches are omitted (by convention, the right-hand branch corresponds to 1 and the lower-hand branch to 0). Also, terminal nodes with constants 0 and 1 are omitted (leaving the AG to the right corresponds to y=1, and down to y=0). To each node in the AG, a path in the circuit corresponds (the correspondence is shown by numbers). For example, a node 8 (bold circle) in the AG represents the bold path from the input branch 8 up to the output of the circuit. The node variables are inverted if the number of invertors in the corresponding path is odd. The set of stuck-at-1 faults along this path in the circuit is represented in the AG by only one representative fault T_2,/1 (T_2 stuck-at-l, i.e. the branch from the node 8 constantly activated to the right direction). The activated paths in the AG (shown by bold arrows) represent the situation when the fault T_2/1 is activated (the test pattern 0011x (Tl,T2,T3, xl,x2), where x denotes an arbitrary value).

2.3.2.4. Representing input and state faults of FSMs in functional level AGs. *Input faults* (b) in FSMs are related to the input lines of the FSM and, in general case, they affect upon more than one transition conditions during the test sequence. Hence, a single structural fault manifests himself as a multiple fault in the iterative array representation of a FSM, which results in difficulties of test generation at the structural level. From the other point of view, input faults are easily to be specified, activated and propagated at the functional level. Hence, in test generation for input faults of the FSM, the functional FSM representation in the form of STD is more preferred than the complex gate level model.

State faults (c) in FSMs are related to the memory flip-flops and, at the functional level, they could be related also to the state decoder, if the latter is a part of the next-state logic or if it is used for implementing output functions. For flip-flops, the stuck-at-0 (1) fault model can be used. For the state decoder, at the functional level, a more general functional fault model is used: stuck-at-0 (1) on outputs and faults "instead of given output another output or a set of outputs is active". The state faults (flip-flop faults) affect upon more than one transition conditions and represent also the multiple fault case for the iterative combinational array

model. Hence, to simplify the test generation, it is recommendable to define and process these faults only at the functional level FSM representation in the form of STD.

For representing the input (b) and state (c) faults in FSMs, alternative graphs will be used, which represent directly the state transition diagram of the FSM (see example in Fig.1). If decoders are used in a FSM for decoding input and/or internal states, then in the AG model, nodes with integer variables will represent these decoders. The functional faults of a decoder (stuck-at-0 (1), "instead of the given output another output is active") are represented by analogical faults at the corresponding node in the AG (compare to the fault model for nodes of AGs in Section 3.1). The structural bit-level stuck-at faults of functional integer variables are not difficult to insert if the tests for them are generated.

From above, it follows that for efficient test generation, a multi-level approach is advisable, where different faults will be at different levels processed. Traditionally for different levels, different languages, fault models and test generation algorithms are used. Introducing AGs as a model for FSMs allows to remove this drawback.

2.3.2.5. General fault notation for AGs.

To generalize the fault model introduced by Definition 5 for the cases where not the all information concerning realistic defects is inherent in the AG-model (e.g. topology dependent bridging and crosstalk faults), define the following general fault notation:

Definition 2.

A fault r in the AG-model $G_y = (M,\Gamma,x)$ is a quadruple: $r = (G_y,m,C,A)$ where

- G is the graph where the fault is defined,
- m is the node affected by the fault,
- C is a condition needed for activating the fault (an assignment of a subset of variables $x^C \subset x$),
- A is the action of the fault, i.e. the list of faulty activated output edges of the node m (the list of active values of x(m)).

The parameters G_y and m of r determine the coordinates of the fault location in the AG-model. The nodes in AGs are closely related to the structure of the system (components, signal paths), hence, we can easily establish the physical meaning of faults at given nodes. The parameter A shows what happens in the model if the fault will take place. Knowing the A we can both, simulate the faulty behavior of the system, and generate tests to distinguish the normal behavior from the faulty one.

The parameter C determines the expected behavior of the node - the value of the variable (or function) at a node. In general, we can distinguish two parts in C: the implicitly given part which can be formally generated from the AG-model during test generation, and the explicitly described part as a list of additional conditions needed for activating the given fault. In some cases (for stuck-at faults), the fault is defined only by the implicit part. In other cases (e.g. for bridging, crosstalking or functional faults), the explicit part should be added as an additional condition for activating the fault. The parameter C allows to adjust the basic abstract fault

model defined for nodes of AGs (Definition 1) to represent different real fault cases. Through the parameter C, hierarchical approach can be implemented in the form of the functional fault model. An individual test pattern (local test) for a module, calculated at a lower level of hierarchy, can be interpreted as the value of C (a condition for activating a functional fault of the module) when assembling high-level tests.

In Fig.9 and Fig.10, correspondingly, a digital system and its corresponding AG-model are depicted, which represent the following function:

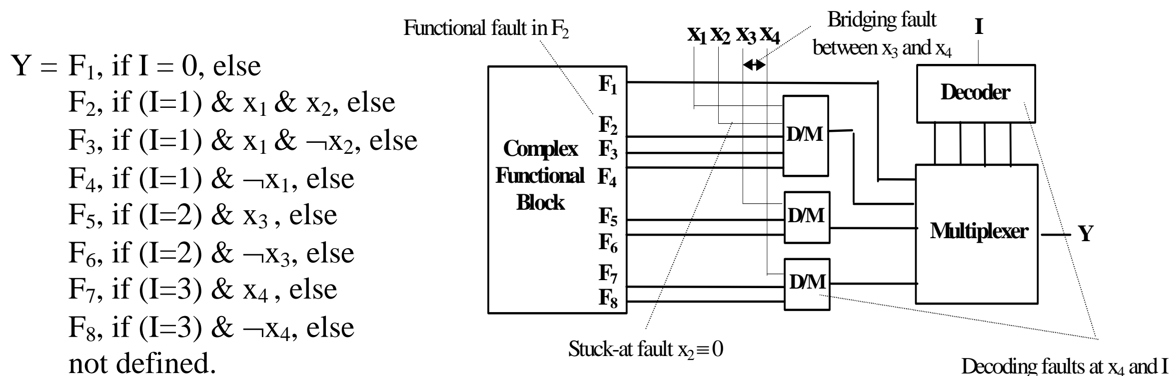

Fig.9. Digital system and fault examples

Here, I – serves as an instruction variable, x_i represent flags (Boolean conditions), and F_k denote the functions performed in the Complex Functional Block. For the faults in Fig.9 and Fig.10, in the following comments, the notation of Definition 2 is used:

1. Stuck-at fault at a line x_2: (Y, 6, x_2 = 1, 0). For this class of faults, it is not needed to give the list of faults explicitly. This list can be generated automatically from the AG.

2. Bridging fault between leads x_3 and x_4: (Y, 4, {x_3=1, x_4=0}, 0), or (Y, 5, {x_3=0, x_4=1}, 0). The condition C (the values of x_3 and x_4) is needed for activating the faulty connection between the two leads while observing the signal value on one of the leads. In this example, the positive coding of signals is supposed. The list of conditions for realistic bridging faults should be added to the AG model.

3. Functional fault in the decoder (instead of the active output 1, another additional output 0 is also active): (Y, 1, I = 1,{0,1}). The fault can be caused, for example, by a bridge or crosstalk between the leads 0 and 1 at the output of the decoder. This type of fault has been introduced as a functional fault for decoders as high-level primitives in [12]. In the case of microprocessors, a similar class of functional faults has been introduced for functions like instruction decoding, source or destination registers decoding in [17] .

4. Functional fault in the subblock $F_2(R_1, R_2)$ where R_1 and R_2 are the arguments (source registers) for the function F_2: (Y, 12, {$R_1 = r_1$, $R_2 = r_2$}, $F_2(R_1, R_2)$)). The condition {$R_1 = r_1$, $R_2 = r_2$}, where r_1 and r_2 denote the contents of R_1 and R_2, represents a local test pattern for

individually testing the function F_2 (the local tests can be taken from the test library, or can be generated for F_2 at a lower abstraction level).

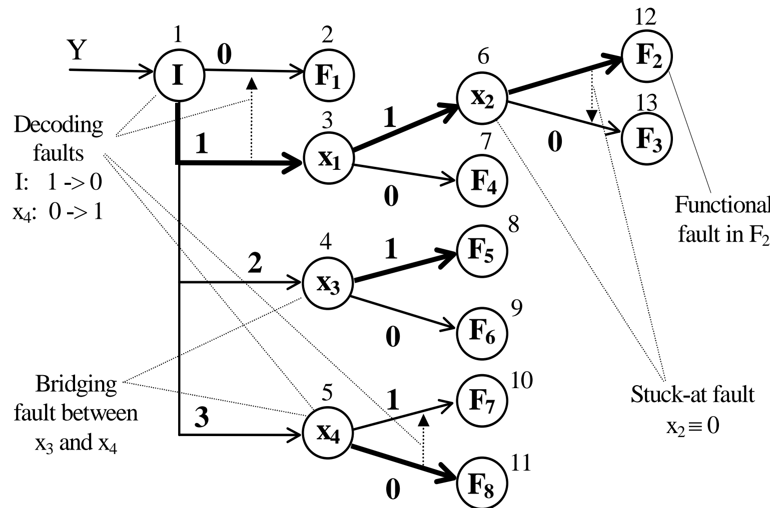

Fig.10. Representing faults in the AG-model

3. Test Generation and Fault Simulation Approach Used in Experiments

3.1. Test generation for the data path of digital systems

In the following, the test generation algorithm described in [15] will be, specified in relation to the technology implemented in the control logic.

Consider a single-graph AG-model G_y which represents a function y = f(x). Denote by $l(m^0,m)$ an activated path from the initial node m^0 to a node m, and by $l(m^i,m^{T,i})$ – an activated path from the node m^i (successor of m for the value z(m)=i) to a terminal node $m^{T,i}$. The two activated paths together with the assignment z(m)=i make up a full activated path which produces the value of function $y=z(m^{T,i})$. Consider now two fault types (G_y,m,i,j) and $(G_y,m,i,\{i,j\})$. The fault type (G_y,m,i,j) covers faults where the node variable under test will change its value (this corresponds to the case of stuck-at gate-level faults). The fault type $(G_y,m,i,\{i,j\})$ corresponds to the functional fault class introduced in [12,17], where two or more decoder outputs or instructions can be simultaneously activated. This fault class is difficult to describe by traditional methods, because the same variable should take simultaneously several values. In the AG-model, at least in the step of fault activation, this problem can be overcome: we simply declare several output edges of the same node active. However, the problem still holds when considering how to calculate the value of the function by graph. Suppose, the fault activates simultaneously two full paths with terminal nodes $m^{T,i}$ and $m^{T,j}$. Formally it means, the function should take simultaneously two values: $y = (m^{T,i})$ and $y = (m^{T,j})$ which is impossible. However, knowing the technology of how the controlled functions are implemented, this case can be represented either by ANDing or ORing both functions.

<u>Theorem 1.</u>
A test pattern detects the fault (G_y,m,i,j), if the following conditions are fulfilled:

- the paths $l(m^0,m)$, $l(m^i,m^{T,i})$ and $l(m^j,m^{T,j})$ are activated, where $m^{T,i} \neq m^{T,j}$, and
- one of the two equations for each data bit is satisfied:

$$\neg z(m^{T,i}) \wedge z(m^{T,j}) = 1, \quad (1)$$
$$z(m^{T,i}) \wedge \neg z(m^{T,j}) = 1. \quad (2)$$

<u>Proof.</u> In accordance to Definition 6, we have z(m) = i. Hence, from the first condition, it follows that a full path from m^0 to $m^{T,i}$ is activated, and $y = z(m^{T,i})$ if the fault is missing. In case of the fault, we have z(m) = j, and in accordance to the first condition, another full path will be activated up to $m^{T,j}$, where $y = z(m^{T,j})$ is valid. Hence, in accordance to the second condition of the theorem, the value of y will be different for each data bit in the fault-free and faulty cases.

<u>Theorem 2.</u>
A test pattern detects the fault (G_y,m,i,{i,j}), if the following conditions are satisfied:

- the paths $l(m^0,m)$, $l(m^i,m^{T,i})$ and $l(m^j,m^{T,j})$ are activated, where $m^{T,i} \neq m^{T,j}$, and
- one of the two equations, depending on the technology, for each data bit is fulfilled:

a) for the OR-technology: $\neg z(m^{T,i}) \wedge z(m^{T,j}) = 1,$ (3)
b) for the AND-technology: $z(m^{T,i}) \wedge \neg z(m^{T,j}) = 1.$ (4)

<u>Proof.</u> From the first condition, it follows that a full path from m^0 to $m^{T,i}$ is activated, and $y = z(m^{T,i})$ if the fault is missing. In case of the fault, we will have z(m) = j, and in accordance to the first condition, another full path will be activated up to $m^{T,j}$, where $y = z(m^{T,j})$ is valid. To force the output functions differ for the fault-free and faulty cases, we should fulfill the conditions:

a) for the OR-technology: $z(m^{T,i}) \otimes (z(m^{T,i}) \vee z(m^{T,j})) = \neg z(m^{T,i}) \wedge z(m^{T,j}) = 1,$
b) for the AND-technology: $z(m^{T,i}) \otimes (z(m^{T,i}) \wedge z(m^{T,j})) = z(m^{T,i}) \wedge \neg z(m^{T,j}) = 1.$

Hence, in accordance to the second condition of the theorem, the value of y will be different in the fault-free and faulty cases.

If we don't know what is the techology implemented in controlling the fuctions, both of the conditions (3) and (4) in Theorem 2 should be fulfilled. If we no exactly the technology implemented, we can use for both fault types (G_y,m,i,j) and (G_y,m,i,{i,j}) the same single condition.

From Theorem 2, a straightforward algorithm follows for generating tests of nonterminal nodes. For all values of $i \in V(m)$, the Theorem 2 should be applied for all values of $j \in V(m)\backslash i$. When generating a test for a given i, as large subset $V'(m) \subseteq V(m)\backslash i$ as possible should be taken to test simultaneously the faults (G_y,m,i,V'(m)). If the solution for solving equations (1-

4) can not be found for all bits by a single set of operands, the test pattern can be repeated for several sets of operands to quarantee the solution for all bits. For terminal nodes, the test generation algorithm will be the same as formulated in [15].

As an example, consider the AG in Fig.10. For testing the fault (G_Y,1,1,{2,3,4}), we have to activate four paths from all the successors of the node 1 up to terminal nodes. In Fig.9 these paths are shown by bold arrows. For I=0, no additional path activation is needed because the successor of the node 1 is himself a terminal node. As the result we obtain a partial test pattern: T = 11110 (I, x_1, x_2, x_3, x_4). To find operands for the case of OR-technology we have to solve the equation $\neg F_2(k) \wedge F_1(k) \wedge F_5(k) \wedge F_8(k) = 1$ for each bit k.

3.2. Test generation for the control path of digital systems

3.2.1. Test generation for FSMs by pipelining partial test sequences.

The test sequence for a single fault consists of three subsequences: initialization sequence, activation sequence and fault propagation sequence. The *initialization sequence* brings the FSM from current state to the state needed for activation the fault, the *activation sequence* contains only one input pattern needed additionally for activation the fault and the *fault propagation sequence* is the state-pair differentiating sequence that differentiates the good destination state from faulty ones and, thus, propagates the fault effect to the output. From the Section 3, it follows that all these subsequences can be carried out at the functional level, except only the fault activation stage for transition faults in the current time frame, which has to be processed at the structural level. However, for transition faults, after they are activated at the lower structural level, the results can be easily transformed as well into the functional level by specifying the internal and input states needed for fault activation.

Test sequences for different faults will be automatically pipelined (overlapped) if we organize the test procedure by moving along paths in the STD rather than by generating tests for different faults separately (Fig.11). The necessary but not sufficient condition to create a test is traversing a set of paths that contains all branches in the STD. If not all faults are yet tested by this sequence, we have to find a set of branches needed for activating the remaining faults, and to traverse a new set of paths that contain all these branches. This procedure has to be repeated until all the faults in FSM will have a test sequence.

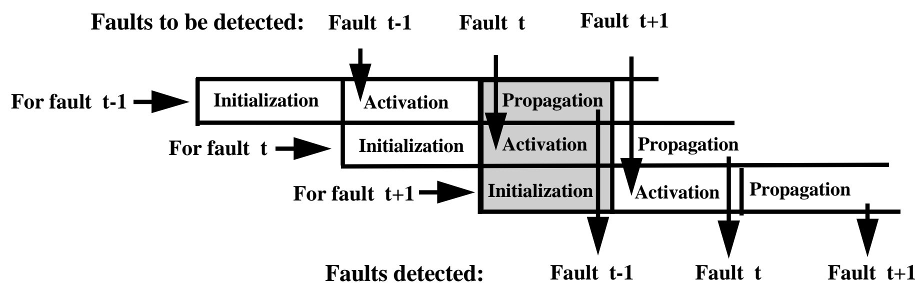

Fig.11. Pipelining test sequences for different faults

In this procedure described, at each current step we have the following information: 1)the current state q' reached by traversing the STD, and 2) the list Q'(q') = {q_k'(F)} of faulty states q_k' for faults f ∈ F activated, but not yet detected, and propagated up to this step (for all q_k' : y(q_k') = y(q') is valid); the faults f are needed to be indicated at the related faulty state only if they manifest himself as multiple faults.

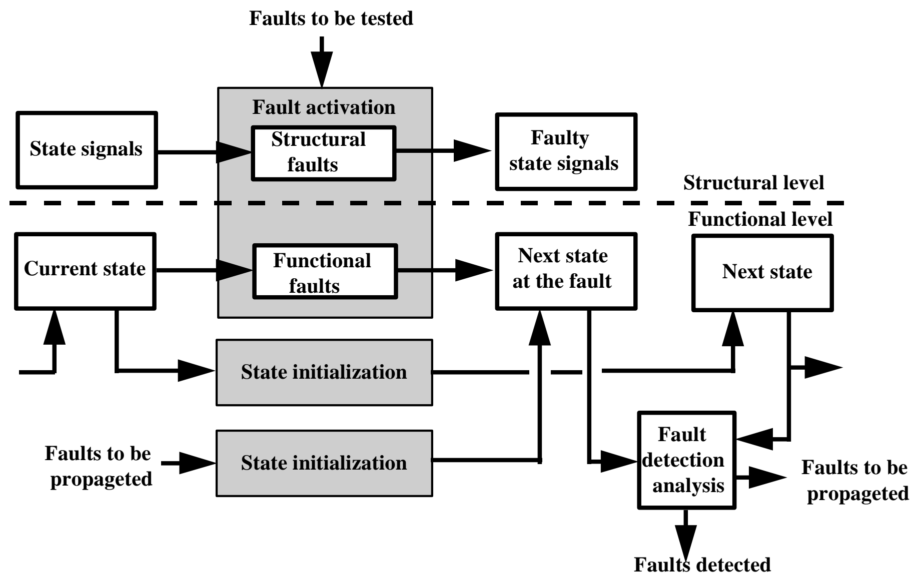

Fig.12. Test generation for the current time frame (current state of the FSM)

The operations to be carried out at the current step of the test generation procedure are the following (see also Fig.12):

- *at the structural level*
1) the current state q is decoded into state signals of flip-flops T_i;
2) fault activation is carried out and input pattern is generated for not yet tested structural faults, or the test pattern is analysed for faults detected, if it is already available;
3) the results are transformed into the functional level
- input pattern is transformed into input state;
- for each detected fault, a faulty next state is calculated and included into Q;

- *at the functional level,*
1) fault activation is carried out and input pattern (input state) is generated for not yet tested functional faults, or the test pattern is analysed for faults detected, if it is already available);
2) for each detected fault, a faulty next state is calculated and included into Q;
3) the next state q for the current q' is calculated;
4) for all current faulty states q_k' ∈ Q', faulty next states are calculated and included into the list Q;

5) for all faulty next states $q_k \in Q'$, the following analysis is carried out:
- if $y(q_k) \neq y(q)$ then the faults related to q_k, are detected;
  q_k is excluded from the list Q;
- if $y(q_k) = y(q)$ then the faults related to q_k, are not detected and they are propagated into
  the next time frame.

Fault activation (or test pattern analysis) at both, structural and functional levels are carried out by uniform procedures using corresponding structural or functional alternative graphs. Also next state calculation and fault detectability analysis are carried out on AGs which represent STDs.

3.2.2. Test generation for FSMs using AGs.

Fault activation and test pattern generation on AGs are based on path activization procedures. Fault analysis is based on path traversing procedures. *In path activation* on AGs, we have a goal-node and we have to find the values of node-variables, so that a path from the root-node up to the goal-node is activated. *In path traversing on AGs,* the values of node-variables are given, and we have to move along a path determined by these values and find a goal-node.

Consider, at first, AGs labelled only by Boolean variables and introduce the following notations:

l(m) - activated path from the root node up to the node m;
l(m, =1) (or l(m, =0)) - activated path from the node m up to the terminal node labelled by the constant l (or 0);
m^1 (or m^0) - successor of the node m for the value z(m)=1 (or z(m)=0).

To activate a fault (generate a test for a fault) z(m)/e (z(m) stuck-at-e), $e \in \{0,1\}$ at a node m, means to activate simultaneously two nonoverlapping paths: $l(m).l(m^{\neg e}, =\neg e)$ and $l(m^e, =e)$ at the value $z(m) = \neg e$. For example, in Fig.8, for testing a fault $T_2/1$ at the node 8, we can activate paths l(m).l(m0, =0) = (l, 7, 8). (11,12,13, =0), and $l(m^1, =1)$ = (9, =1), which gives the test pattern 0011x (Tl,T2,T3,xl,x2). Activated paths in Fig.8 are depicted by bold arrows.

To analyse a test pattern for faults detected, means:
1) to find an activated by the pattern path l with a terminal node m^T where $z(m^T) = e$,
2) for all nodes $m_k \in l$, find the value $e_k = z(m_k^T)$ where m_k^T is the terminal node of the path $l(m_k^{\neg e}, m_k^T)$ activated by the same pattern;
3) for all nodes $m_k \in l$, the given pattern detects the fault $z(m_k /\neg e$ if $e_k \neq e$ is valid.

As an example, in Fig.8, by the test pattern 0011x (T1, T2, T3, x1, x2), a path l = (1,7,8,11,12,13) is activated. The condition (3) is valid only for nodes 8 and 13. So, by this pattern, the faults $T_2/1$ and $\neg$xl/1 are detected.

In the general case of AGs labelled by integer variables, test generation is based on the same path activization principles. Denote by $l(m^i, m^{T,i})$ - activated path from the node m_i up to a terminal node $m^{T,i}$ (m^i is the successor of the node m for the value z(m)=i).

To activate a fault (generate a test for a fault) z(m)/i -> j (z(m)=j instead of z(m)=i), means to activate simultaneously nonoverlapping paths l(m) and $l(m^k, m^{T,k})$ where k = i, j, so that $z(m^{T,i}) \neq z(m^{T,j})$.

For example, in Fig.2, to activate the fault q'/2 -> 5 to output y, two test patterns are possible: 2010 (q', Res, xl, x2) or 2001. Here, in terminal nodes, for comparison, only y is considered. By the first pattern 2010, the following three paths for testing the node m = 3 (i = 2, j = 5) are activated: l(3) = (1,3), l(7, =0) = (7, 9, =0) and l(11, =1) =l(11,10, =1). As an example of the test pattern analysis, consider again a pattern 2010 that activates a path l = (1, 3, 7, 9) on the AG in Fig.8 (shown by bold arrows). The condition (3) of the fault detection is valid only for the node 7 and for the values l, and 4 of the variable q' in the node 3 (shown by bold circles). Hence, the following faults are detected by this pattern: q'/2 -> l, q'/2 -> 4, x2/l.

3.2.3. Complexity of test generation.

Using the described multi-level approach, it is possible to reduce the complexity of test generation and the complexity of discovering redundant faults nearly to the complexity of solving the same tasks for combinational circuits. Test generation for transition faults in gate-level next-state logic will be carried out, actually, in only a single time frame - a pattern will be generated on structural AGs, which specifies a state needed for testing the given fault. If the state is reachable, then the fault can be tested. On the contrary, if the state is not reachable, the fault is redundant and not testable. The reachability of states can be determined on the functional level, using AGs that correspond to STDs.

As an example, when trying to generate a test sequence for a fault ¬T3/1 at the node 3 in the graph D_3 in Fig.8, it is needed only to try to test this node in D3. Activating the path 1(3) = (1,2,3), the only possible path to reach the node 3, it cames out that a state q = 7 (Tl=l, T2=1,T3=1) is needed to test the given fault. On the other side, at the functional level, it is easy to see that this state is not reachable. Hence, without trying to create any sequence longer than l, it was possible to show that the fault ¬T3/1 is redundant and not testable.

3.3. General structure of the automated test pattern generator

Input data for our test generation system is a register-transfer level VHDL description of the device. Such representations are provided by various high-level synthesis (HLS) tools where behavioral descriptions are compiled into register-transfer level ones. Fig. 13 shows the place of the test generation system in the design process.

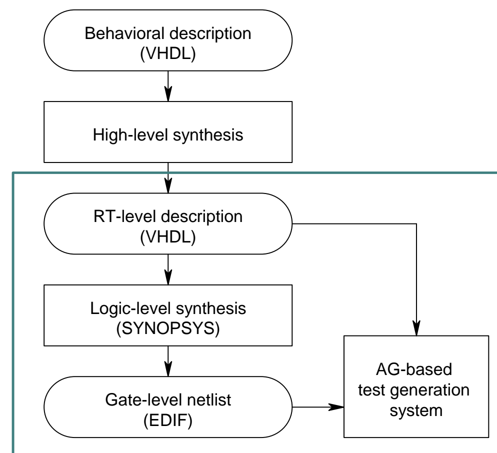

Fig. 13. The Test System and Design Process

The general structure of the test generation environment is presented in Fig.14. It consists of a hierarchical datapath test generator, a control part test generator, and a high-level AG model generator. From RT-level VHDL description, high-level AG generator generates high-level AG model, which will be applied as an input for the datapath and control part test generators.

3.3.1. Control part test generator

The *control part test generator* works in the following way. Two types of faults will be considered: the faults caused by defects in the next state logic of the control part (transition faults), and the faults caused by defects in the control logic in the data part (output faults). Because of these two types of faults, the ATPG consists also of two main parts working together in the test generation system. The first of them, transition fault processor (TFP) begins to work from the initial state and will traverse step by step all the transitions of the finite state machine (FSM). In each step, the processor introduces all activated at the current transition faults. Suppose, that N1 faulty machines and one fault-free machine were processed at the current step. The TFP passes all necessary inputs to the second part of the ATPG – to the output fault processor (OFP) for simulating the behavior of the data part for all the activated N1+1 machines. The second task is to introduce now all the activated at this transition control faults in the data part. Suppose, that M1 additional faulty machines were produced in the data part at the current step. The OFP passes now all necessary inputs for all the N1+M1+1 machines back to the TFP for simulating the behavior of the control part. In the same time, the OFP checks if some of faulty machines will produce different as expected output signals in the data part. If a fault will be detected, the corresponding faulty machine will be removed from the list of simulated machines. The described procedure will continue for the next transition. Fault introducing continues till all not yet detected faults are considered. The procedure continues till all the faults will be detected. More details of the control part test generator are given in [21].

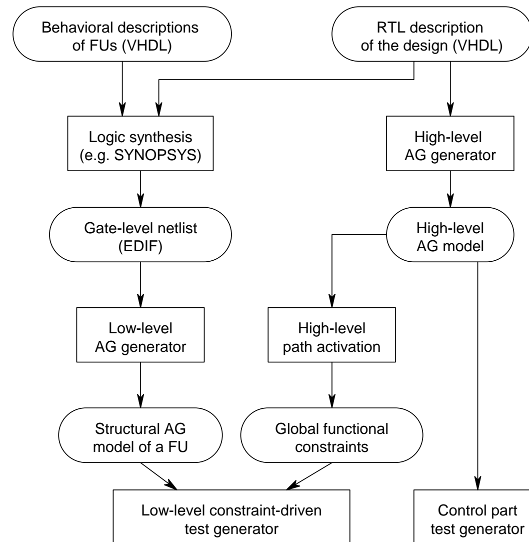

Fig. 14. The AG-Based Test Synthesis System

The datapath test generator has a hierarchical structure. The high-level part of the generator performs symbolic path activation on RT-level. During the path activation, functional constraints will be extracted which will be applied to the low-level part of the generator. The low-level test generator handles justification and propagation constraints derived at the high level. Additional tasks of the low-level ATPG are to generate gate-level tests for the functional units (FU) and to assemble the final test of the datapath. In order to test the FUs, gate-level models of the FUs must be synthesized. Current system uses Design Compiler by Synopsys Inc. for the logic-level synthesis. As an input for logic-level synthesis are the RT-level VHDL description and a VHDL library of FUs, containing generic bit-width behavioral descriptions of the FUs.

Due to the fact that the low-level test generator operates with structural AG (SAG) representations, low-level AG generator is required to generate SAG models from gate-level netlists. The low-level AG generator creates SAG representations from EDIF 2.0.0 netlist descriptions. EDIF is a technology-dependent design format and therefore, appropriate technology libraries have to be included while performing EDIF to SAG conversions.

3.3.2. Data path test generator

The test generation process for data path takes place in the following way. Tests are generated sequentially for each functional unit (FU). For each FU, justification and propagation constraints are extracted at the high level and passed to the lower level test generator. During constraints extraction for the target FU, for all non-target FUs, functional information is applied to perform propagation and justification through the FUs at the functional level. Such an information, in form of simplified behavior of the block is preliminarily extracted and recorded in a special *transparency library*. This information will consist of a set of input/output mappings (so called I-paths [19] and F-paths [20]).

The low-level test generation process consists of two stages. In the first stage, input values are generated to satisfy the high-level conditional constraints. This task can be treated as a typical constraint satisfaction problem (CSP) [22]. In the second stage, random values are generated and simulated through propagation constraints to derive input patterns for structural level fault simulation of the FU under test. High-level test generator calls the low-level generator repeatedly in a loop. In general case a single activated path is not enough to reach 100 per cent fault coverage for a functional unit, i.e. test set for a FU can consist of vectors generated during different activated paths, and therefore, different calls to the low-level program. Record has to be kept of the faults detected by previous low-level test generation runs. On each call the low-level generator reads and writes the list of currently covered faults and keeps it in a special file.

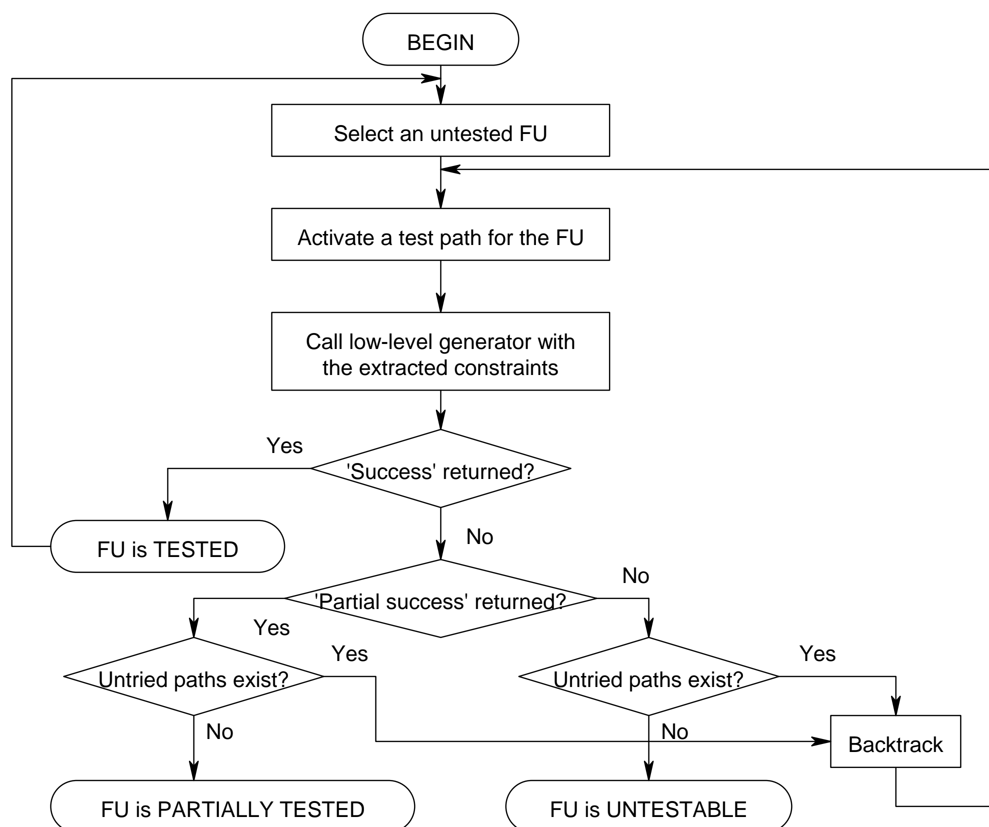

Fig. 15. Interaction between High-Level and Low-Level Generators

Untestable faults for a FU are determined in current approach by the following method. Previous to starting test generation of the FU, deterministic gate-level test generator finds the list of redundant faults in the FU. If some inputs of the FU are directly tied to constant signals, it will be taken into account while determining the untestable faults. Fault efficiency is considered to be 100 per cent if *number of tested faults = total number of faults - number of untestable faults*. Due to possible high-level constraints, 100% fault efficiency may be unreachable. When a path is activated, the high-level generator calls low-level test generator

and passes the extracted conditional and propagation constraints to the latter. If the low-level generator satisfies all the path activation constraints and generates test vectors achieving 100% fault efficiency for the FU, 'success' will be returned to the high-level generator. Current functional unit will be considered to be tested and next untested FU will be chosen by the high-level generator. In the case when the low-level generator can not solve the extracted conditional constraints, or if the achieved fault coverage in current FU remains low or unchanged, high level generator will be informed about it and it will try to activate an alternative path for testing the FU. Fig. 15 shows the data flow of interaction between both parts of the generator.

4. Experimental Results

4.1. Low-level test generation experiments

A CAD system Turbo Tester (TT) was developed for solving test design tasks at the gate-level with random logic [23].TT is an easy-to-learn, easy-to-set-up and low-cost system. The package has interfaces to commercially available VLSI design tools like Cadence, Synopsys, Mentor Graphics, Viewlogic, Compass, OrCAD, ASYL+, Dixi-CAD [15] etc. It includes a large set of tools as implementations of different methods of test synthesis and analysis.

Turbo Tester has a character-mode object-oriented user interface, which confirms with the Common User Access (CUA) standard and runs under DOS operating system. The environment can easily be executed in the MS Windows' DOS box. It has following features: Comprehensive on-line help, Full mouse support, Text editor, Extended video text modes, Menus, including local pop-up menus, giving instant access to all important functions of the Turbo Tester.

In TT, different methods for test pattern generation, fault simulation, multi-valued simulation, test quality evaluation, fault detection probability calculation and testability analysis are implemented. The fault classes considered include stuck-at faults and transition faults (delay faults, stuck-opens). The tools for BIST simulation and quality analysis are also implemented. Different structures for BILBO, circular self-test path and "store and generate" approaches can be simulated, evaluated and compared. A short description of tools currently implemented in TT is given in [23].

The component library consists of alternative graph representations (AG-models) for the components of the circuits to be processed. The library is open and can be updated for new components. From the netlist of the design, produced by schematic editor, the model generator creates an AG-representation of the design. The design can be represented either at the gate-level or at the macro-level. At the gate-level, to each gate of the network a single AG will correspond. At the macro-level, a "compressed" structural model will be generated, where the tree-like (fan-out free) subnetworks of the circuit are as macros considered. To each macro, a single AG by superpositioning gate-level AGs will be created where one-to-one correspondence between signal paths in the macro and nodes in the AG will be established. All

the tools implemented in TT use the AG-representation as the only information about the circuit to be processed.

The number of faults to be processed at the macro level will be less than the number of faults at the gate level (each macro-level fault represents, in general, a set of gate-level faults). The fault collapsing at the macro level causes that the productivity of the test generation at the macro level will increase compared to that of the gate-level. Comparison of test generation efficiency for gate and macro-level approaches are given in Table 3. Because of the fault collapsing test generation time decreases 2,6 to 5,1 times.

Table 3.

Circuit			Time (sec)	Number of patterns	Fault cover %	
Name	Level	Number of target faults			Gate-level	Macro-level
c499	macro	1202	131	106	99,6	99,3
	gate	2194	662	109	99,5	
c880	macro	994	46	112	98,6	98,6
	gate	1550	119	101	98,1	
c1355	macro	1618	278	105	99,6	99,5
	gate	2194	953	107	99,5	
c1908	macro	1732	219	169	99,0	98,5
	gate	2788	743	160	99,0	

In Turbo Tester, static, dynamic and statistical test quality analysis tools are implemented. All the tools have the target to estimate the fault coverage of the given set of test patterns. Static two-valued fault-simulation can be carried out either by parallel critical path tracing or by deductive fault analysis methods. In Table 4 the critical path tracing method is compared for the gate level AG-model and for the compressed macro level AG-model. Simulation time in ms per pattern is given for both models. Because of fault collapsing and decreasing of the model complexity, simulation time decreases 2,6 to 9,1 times for the given set of benchmark circuits.

Table 4

Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
ms/pattern (Gate)	8,5	45,0	17,5	37,0	62,0	110	185	360	700	750
ms/pattern (Macro)	2,2	6,5	3,4	12,0	9,6	17	21	43	275	83
Time ratio (M/G)	3,9	6,9	5,2	3,1	6,5	6,5	8,8	8,4	2,6	9,1

4.3. Test experiments with control units

A multi-level test generation system CPTEST for testing control units of digital systems [21], was implemented in C++ at the Tallinn Technical University. The system is considered as a part of a hierarchical ATPG developed in the framework of the FUTEG project.[24-26]. The finite state machines considered as examples for experimental research are those of MCNC benchmarks for test synthesis. For our experiments, the gate-level implementations were

synthesized by Synopsys. No control was exercised on this tool, and binary state coding was applied. Test generation results for 15 FSM's in Table 5 are described in Table 6. For each example in Table 5, the numbers of inputs (*Inp*), outputs (*Out*) and transitions (*Tran*) are given. In Table 6, on the left side the length of test sequence (number of patterns) needed in order to have traversed throughout all branches in STG each at least once (*Test length*), the fault coverage achieved by traversing all branches (*Coverage*), and the time required for that (*Time*) on a PC 486 66MHz are shown for each example.

Table 5. Characteristics of MCNC benchmark FSMs

FSM	States	Inp	Out	Tran
lion9	9	25	2	1
bbara	10	4	2	60
cse	16	7	7	91
sand	32	11	9	184
planet	48	7	19	115
vtiidec	5	11	32	77
mc	4	3	5	10
dk15	4	3	5	32
lion	4	2	1	11
tav	4	4	4	49
log	17	9	24	29
s27	6	4	1	34
beecount	7	3	4	28
bbsse	16	7	7	56
mul8x8*	8	4	13	21

* The circuit is not MCNC Benchmark

On the right side of Table 6, the length of test sequences (*Test length*), the total number of gate-level faults (*Total faults*) in FSM, the number of inserted (activated) faults (*Ins faults*), the number of detected faults (*Detected faults*), the fault coverage achieved (*Coverage*), and the time required (*Time*) are given. In the present version of CPTEST, for searching the target state (when activating a target fault), and for searching the state where the activated fault can be detected, the random path traversing technique is used. Also, in this version nonefficient traversing cycles which do not increase the fault coverage are not excluded from the total test sequence. A deterministic technique is currently under development which is expected to increase the efficiency of the tool in reducing dramatically the test length, test generation time and increasing the fault coverage.

Table 6. Test generation results for MCNC benchmarks

FSM	Test length	Total faults	Ins. faults	Detected faults	Coverage, %	Time, min
lion9	37	112	112	112	100.00	0.00,45
bbara	144	202	194	193	95.54	0.03,18
cse	615	540	538	527	98.70	0.43,44
sand	767	1140	1119	1119	98.16	1.22,09
planet	900	1070	1058	1058	98.88	1.22,07
vtiidec	823	210	207	207	98.57	0.12,58
mc	14	74	74	74	100.00	0.00,14
dk15	67	92	92	85	92.39	0.01,10
lion	20	58	58	58	100.00	0.00,15
tav	14	34	34	34	100.00	0.00.09
log	399	378	367	367	97.09	0.40,70
s27	48	60	60	60	100.00	0.00,36
beecount	150	126	126	120	95.24	0.02,70
bbsse	867	456	451	438	96.05	0.47,12
mul8x8	313	94	94	93	98.94	0.01,76

The results of the experiments listed in Table 6 can be compared with published results [27,28] of using different approaches and the same benchmarks described in Table 7. In our approach, no modifications of gate-level circuits produced by Synopsys have been made to improve the testability as, for example, in [27].

Table 7. Comparison with other ATPGs

	HITEC [27]			STED [28]			CHE90 [29]		
FSM	No. of vec-tors	Cover %	Time, s (Sparc2)	No. of vec-tors	Cover %	Time, s (on VAX 11/8800)	No. of vec-tors	Cover %	Time, s Sun 4/260
lion9	38	97,3	8.63	-	-	-	-	-	-
bbara	96	82.0	89.33	-	-	-	241	100.0	2
cse	349	100.0	23.48	397	100.0	29.5	880	97.86	45
sand	52	45.2	1339.9	722	99.43	7.7min	809	97.74	202
planet	91	64.5	917.7	1046	100.0	5.8min	600	98.26	35
mc	38	100.0	0.37	-	-	-	-	-	-
dk15	53	100.0	0.73	-	-	-	146	100.0	0.2
lion	47	100.0	0.45	-	-	-	-	-	-
tav	26	100.0	0.27	-	-	-	-	-	-
bbsse	255	100.0	18.38	-	-	-	-	-	-
s27	40	100.0	0.27	-	-	-	-	-	-
beec.	85	100.0	1.40	-	-	-	-	-	-

4.3. Test synthesis for digital systems with global feedbacks

Experiments were carried out with the control part and datapath generators for benchmark circuits with global feedbacks. In all the experiments, system models were used where both

datapath and control part were connected. All experiments were run on Sun Sparcstation 20 computer. As an input for the datapath test generator, a hierarchical model of a 16-bit multiplier was chosen. Test generation results for four functional units (FU) in the circuit are given in Table 8. By increasing the interaction limit between high- and low-level generators three of the FUs were tested with 100 per cent efficiency. For one of the FUs, the transparent paths could not be activated at the high level. Test generation times are not included because at present file transfer is used for interaction between high and low level parts which significantly reduces the speed.

Table 8 *Data Part Test Generation Results*

Interaction limit	FUs	Inter-actions	Vectors	Result	Interaction limit	Fus	Inter-actions	Vectors	Result
1	add1	1	7	Success	1000	add1	1	7	success
	add2	1	0	Failure		add2	1000	76	partial
	and1	1	3	Success		and1	1	3	success
	sub1	1	0	Failure		sub1	244	0	failure
100	add1	1	7	Success	∞	add1	1	7	success
	add2	100	0	Failure		add2	1093	78	success
	and1	1	3	success		and1	1	3	success
	sub1	100	0	failure		sub1	244	0	failure

Table 9 Control Part Test Generation Results

Benchmark circuits	Number of faults	Coverage %	Test vectors	Time s
s344	90	84.44	11	0.26
diffeq	104	91.35	16	0.46
gcd	142	93.66	40	0.80

The results of test generation for control parts (finite state machines) of three different digital systems from FUTEG benchmarks list [30] (multiplier s344, differential equation solver diffeq, and greatest common divisor gcd) are shown in Table 9. All the control faults were tested through the data path loop. To increase the fault coverage, corresponding improvements of the testability are needed.

4.4. Test generation for RISC architectures

With the goal to investigate the adequacy of the high-level fault model defined for AGs and to investigate the possibility to reach high quality gate-level tests by using high-level descriptions only, experiments were carried out with a restricted class of digital systems - with FUTEG benchmarks based on a family of n-bit simplified RISC processors [30]. Only arithmetical and logical operations were implemented and only combinational parts of RISC processors were examined. The family of benchmarks consists of processors which vary in the instruction set (processors with 4, 8 and 16 instructions) and in the bitwidth (4, 8, 16 and 32-bit processors). The benchmark family was created by describing the high level behavior of processors in VHDL and by synthesizing the gate-level implementations with SYNOPSYS. Then AG-models were synthesized both for higher (instruction) level and lower (gate) level

designs. Libraries of local tests for functional components were created by either a low-level ATPG or manually for simple functions.

Table 10

IS	BW	ATPG	Time (s)	Patterns	Optim. patterns	Faults	Detected faults	Fault coverage (%)
4	4	HTPG	0,02	63	25	612	611	99,8
		Synopsys	0,21	30	25	596	595	99,8
4	8	HTPG	0,02	63	29	1168	1167	99,9
		Synopsys	0,49	45	33	1168	1167	99,9
4	16	HTPG	0,03	63	29	2240	2239	99,9
		Synopsys	1,16	63	36	2240	2239	99,9
4	32	HTPG	0,07	63	29	4404	4401	99,9
		Synopsys	3,74	77	45	4404	4401	99,9
8	4	HTPG	0,02	120	30	708	708	100
		Synopsys	0,19	45	25	708	708	100
8	8	HTPG	0,05	120	30	1320	1320	100
		Synopsys	0,50	52	31	1232	1232	100
8	16	HTPG	0,08	120	29	2540	2540	100
		Synopsys	1,25	61	41	2364	2364	100
8	32	HTPG	0,10	120	30	5018	5018	100
		Synopsys	4,26	75	50	4676	4676	100
16	4	HTPG	0,08	224	39	900	900	100
		Synopsys	0,29	46	32	855	855	100
16	8	HTPG	0,10	224	43	1612	1612	100
		Synopsys	0,75	64	42	1531	1531	100
16	16	HTPG	0,13	224	42	3016	3016	100
		Synopsys	1,86	73	48	2861	2861	100
16	32	HTPG	0,15	224	42	5908	5908	100
		Synopsys	5,57	84	59	5607	5607	100

The results of experiments carried through with 12 different processors from the benchmark family are depicted in Table 10. Here, the columns have the following meaning: IS - number of instructions of the benchmark processor, BW - bitwidth of the processor data word, ATPG - type of the test generator (as reference for comparison, the SYNOPSYS gate-level ATPG was used), processor time in seconds used for test generation, number of patterns generated by ATPG, number of patterns after the optimization, total number of faults, number of faults detected, and the fault coverage. The cost of local test generation for building blocks is not included in the HTPG's Time column, because the library tests were generated once for multiple use in HTPG. For generating tests of building blocks by SYNOPSYS, we needed time less than 10% of the SYNPOSYS test generation time for the whole circuit. This overhead should be added to HTPG's Time column for representing the total cost of mixed-level ATPG.

The results of the experiments show the efficiency of both the high-level and hierarchical mixed-level approaches compared to the gate-level approach. The efficiency rises when the complexity (number of instructions and bitwidth) increases. In both structural and functional test generation cases, only 1-3 redundant faults in the control part were found which remained undetected.

4.5.Fault simulation experiments with BIST architectures

Built-in self-test (BIST) is the capability of a circuit to test itself. It is an alternative to conventional test, which uses precalculated stimuli and response data stored in external equipment. Most universal BIST techniques are based on the concept of using an LFSR (Linear Feedback Shift Register) for random test pattern generation, and a MISR (Multiple-Input Signature Register) for test response compaction, combined with scan path design techniques like LSSD (Level Sensitive Scan Design). The representative examples (Fig.16) of such techniques are STUMPS, LOCST, and the techniques based on using BILBO (Built-In Logic Block Observer) modules, CSTP (Circular Self-Test Path) conception [31,32] or universal scheme of “store-and-generate” [33].

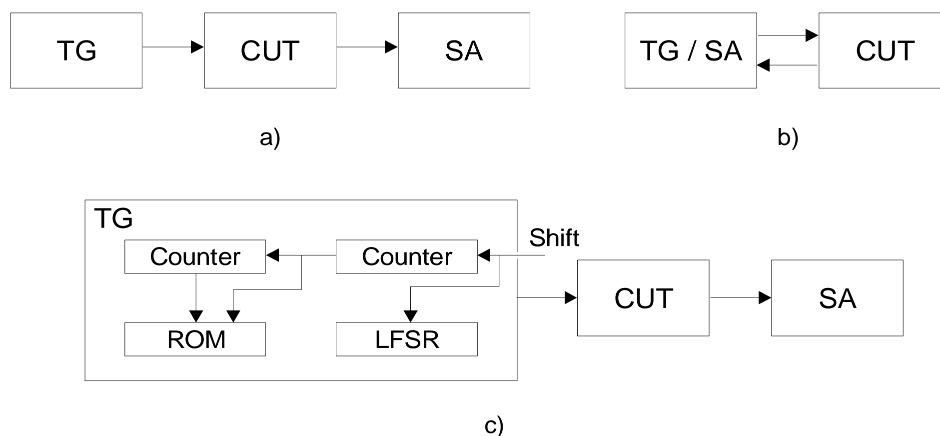

Fig.16. BIST architectures: a) BILBO, b) CSTP, c) Store-and-generate.

A serious question arises as to the effectiveness of the pseudorandom patterns used in these BIST architectures: designers will usually want quantitative proof of the effectiveness of such schemes. To evaluate the quality of pseudorandom test patterns in BIST architectures frault analysis tools and BIST simulators were developed as tools in the Turbo-Tester [23].

The BIST approach is represented here by applications for BILBO (Fig.16a.), CSTP (Fig.16b.) and “store-and-genearate” scheme (Fig.16c.) emulation. In Fig.16, the following notations are used: TG – test generator, CUT – circuit under test, SA – signature analyser. Different BIST architectures can be simulated and self-test quality for these structures evaluated. In TG and SA, Linear-Feedback-Shift-Registers (LFSR) are used, where the characteristic polynomial (i.e. the structure) of the LFSR to be used is easy to be specified. It is possible also to emulate the general "store-and-generate" approach by tools in [23]. The whole test will be generated on the basis of a given set of test patterns (the prestored part of the test). All the prestored patterns will serve as initial input test patterns for on-line test generation by BILBO or CSTP (the generated part of the test).

With the goal to investigate tradeoffs between hardware overhead and test quality early in the design process, a simplified 8-bit processor with reduced instruction set with built-in self-test circuitry was designed by CADENCE tools.

Table 11

Method	Control part			Data part		
	Test quality		Hardware overhead	Test quality		Hardware overhead
	Fault cover	Number of vectors		Fault cover	Number of vectors	
BILBO	98.7%	967	36,7%	95,1%	937	15,3%

During experimental investigations different BIST structures were simulated by Turbo-Tester (both BILBO and CSTP solutions) without actually designing them. The best solutions of separate BISTs for control and data parts of the processor in terms of the highest fault coverage are presented in Table 11. Very high hardware overhead of the BIST circuitry in case of the control part can be explained by the low hardware amount in the next state logic. Relatively low fault coverage reached in case of the data part can be explained by the high amount of redundant faults in the circuit under test.

5. Conclusions

A uniform approach based on decision diagrams (AGs) has been proposed to combine topological gate-level approaches, functional BDD-based techniques and high-level behaviour-oriented methods in test generation for digital systems. A new method for jointly describing the functionalities of components of a system network as well as the transparency conditions for fault propagation through high-level components by decision diagrams is proposed. A new advanced fault model for AGs and a new fault activation algorithm for functional faults, specified in relation to the technology implemented, are introduced. The method supports both, high-level (behavioral) and hierarchical (mixed-level) test generation schemes.

Experiments made up to now with a set of benchmark circuits which cover internationally known ISCAS’85 and MICN benchmark circuits for investigation gate-level algorithms, and mixed level benchmark circuits developed in FUTEG project, showed the efficiency of higher level AGs and the uniform fault model in generating high quality test patterns with very high speed. The advantage of using topological fault simulation technique on compressed macro-AG models compared to using gate-level topology is shown by experiments on ISCAS’85 benchmark circuits. The efficiency of algorithms developed for finite state machines are compared to known algorithms on MICN benchmark circuits. The efficiency of mixed-level test pattern generation was shown on FUTEG benchmarks which represented digital nsystems with global feedbacks.

A dynamic combination of functional AGs (BDDs) and structural macro-AGs can contribute for more efficient test generation or fault simulation in large digital circuits. For the new promising trend of combining BDD-based symbolic techniques and traditional topological algorithms, AG-s provide a uniform theoretical basis. To exploit this basis for further investigations to increase the efficiency of ATPGs by combining mixed symbolic and topological techniques with hierarchical approaches will be the goal of the future work.

6. References

[1] Anirudhan P.N., Menon P.R. Symbolic test generation for hierarchically modeled digital systems. *IEEE 1989 International Test Conference,* pp.461-469.

[2] Leenstra J., Spaanenburg L. Hierarchical test assembly for macro based VLSI design. *IEEE 1990 International Test Conference*, pp.520-529.

[3] Karam M., Leveugle R., Saucier G. Hierarchical test generation based on delayed propagation. *IEEE 1991 International Test Conference*, pp.739-747.

[4] Lee J., Patel J.H. Hierarchical test generation under intensive global functional constraints. *Proc.29th ACM/IEEE Design Automation Conf.,* pp. 261-266, June 1992.

[5] Sarfert T.M., Markgraf R., Trischler E., Schulz M.H. Hierarchical test pattern generation based on high-level primitives. *IEEE 1989 International Test Conference*, Sept. 1989, pp.470-479.

[6] D. Bhattacharya and J.P. Hayes. A hierarchical test generation methodology for digital circuits. *JETTA: Theory and Application,* vol. 1, pp. 103-123, 1990.

[7] Kunda R.P., Abraham J.A., Rathi B.D. Speedup of test generation using high-level primitive. *ACM/IEEE 27th Design Automation Conference,* pp.580-586, June 1990.

[8] Santucci J.F., Courbis A-L., Giambiasi N., "Speed up of behavioral ATPG. using a heuristic criterion", *30th ACM/IEEE Design Automation Conference*, pp. 92-96, 1993.

[9] Steensma, W. Geurts, F. Catthoor, H. De Man. Testability Analysis in High Level Data Path Synthesis. *J. of Electronic Testing: Theory and Applications,* 4, 1993. pp.43-56.

[10] Thatte S., Abraham J. Test generation for microprocessors. *IEEE Trans. on Comp.,* June, 1980, pp.429-441.

[11] Su S.Y.H., Lin T. Functional testing techniques for digital LSI/VLSI systems. *21st Des. Autom. Conf.,* 1984, pp. 517-528.

[12] Abraham J.A. Fault modeling in VLSI. *VLSI testing. North-Holland* 1986, pp.1-27.

[13] Ward P.C., Armstrong J.R. Behavioral fault simulation in VHDL. *ACM/IEEE 27th Des. Autom. Conf.*, 1990, pp. 587-593.

[14] Giambiasi N. et. al. Test pattern generation for behavioral descriptions in VHDL. *Proc. of the VHDL conference*, Stockholm, 1991, pp. 228-234.

[15] Ubar R. Test Synthesis with alternative graphs. IEEE Design & Test of Computers. Spring 1996, pp.48-57.

[16] Akers S.B. Binary Decision Diagrams, *IEEE Trans. on Comp.,* Vol.27,1978, pp.509-516.

[17] Thatte S.M., Abraham I.A.(1980). Test generation for microprocessors. *IEEE Trans. on Computers,* Vol.29, pp.429-441.

[18] Lin T., Su S.Y.H. VLSI functional test pattern generation - a design and implementation. *IEEE 1985 International Test conference,* pp.922-929.

[19] Abadir M.S., Breuer M.A. A Knowledge-Based System for Designing Testable VLSI Chips. IEEE Design & Test, August 1985, pp.56-68.

[20] Freeman S.. Test Generation for Data Path Logic: The F-Path Method. IEEE J. of Solid-State Circuits, Vol.23, April 1988, pp.421-427.

[21] R.Ubar, M.Brik. Multi-Level Test Generation and Fault Diagnosis for Finite State Machines. Lecture Notes in Computer Science No 1150. Dependable Computing – EDCC-2. Springer-Verlag, pp. 264-281.

[22] K.Tilly. A Comparative Study of Automatic Test Pattern Generation and Constraint Satisfaction Methods. Technical Report, Ser. Electrical Engineering, Technical University of Budapest, June, 1994.

[23] G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. Teaching Test and Design for Testability with TURBO-TESTER Software. *Proc. of the 3rd Workshop on Mixed Design of Integrated Circuits and Systems,* Lodz, May 1996, pp. 589-594.

[24] Sallay B., Petri A., Tilly K., Pataricza A. High Level Test Pattern Generation for VHDL Circuits. *IEEE European Test Workshop*, Montpellier, France, June 12-14, 1996, pp. 201-205.

[25] Gramatova E., Cibakova T., Bezakova J. Test Pattern Generation Algorithms on Functional/Behavioral Level. Tech. Report FUTEG-4/1995.

[26] Gulbins M., Straube B. Applying Behavioral Level Test Generation to High-Level Design Validation. *The European Design & Test Conference*, Paris, March 11-14, 1996, p. 613.

[27] Niermann T.M., Patel J.H. HITEC: A test generation package for sequential circuits. *Proc. European Design Automation Conference*,1991, pp.214-218.

[28] Ghosh A., Devadas S., Newton A.R. Test generation and verification for highly sequential circuits. *IEEE Trans. on CAD*, Vo1.10, No.S, May 1991.

[29] Cheng K.-T., Jou J.-Y. Functional test generation for FSMs. IEEE *Int. Test Conference.* 1990, pp.162-168.

[30] Gulbins M., Pataricza A., Gramatova E., Seinauskas R., Marzouki M., Ubar R. FUTEG Benchmarks. Tech. Report FUTEG-1/1997.

[31] M. Abramovici, M.A.Breuer, A.D.Friedman. “Digital Systems Testing and Testable Design”. Computer Science Press, 1995.

[32] V.D. Agrawal, C.R.Kime, K.K.Saluja. A Tutorial on Built-In Self-Test: Part 2 – Applications. IEEE Design & Test of Computers, pp. 69-77, June 1993.

[33] V.K. Agarwal, Store and Generate Built-In-Testing Approach. *IEEE Trans. on Computers,* 1981, pp. 35-40.