

Test Synthesis with Alternative Graphs

RAIMUND UBAR

Tallinn Technical University

Alternative graphs provide an efficient, uniform model describing the structure, functions, and faults in a wide class of digital circuits and for different representation levels. For test pattern generation, they provide a general theoretical basis for combining high-level approaches, symbolic techniques based on binary decision diagrams, and traditional topological algorithms.

THE INCREASING COMPLEXITY of digital circuits renders classical gate level test generation impractical, and researchers have devoted much effort to developing an alternative field of functional testing that decreases test generation complexity.^{1,3} However, estimating fault coverage at the functional level is difficult because the accuracy of functional fault models is unproven and gate level models better characterize physical faults.

A solution to these shortcomings, hierarchical test generation, appeared in the last decade.^{4,5} It incorporates the benefits of functional testing yet retains the accuracy of gate level fault models.

However, representing digital systems at various levels (behavioral, procedural, functional, or gate) usually needs different mathematical tools for each level. For example, for systems represented as a composition of control and data parts, we use different approaches, tools, and fault models to handle each part. This increases CAD system costs because we must purchase and develop many design tools. Also, it is difficult to integrate test generation tools for systems represented at mixed levels simultaneously.

The lack of a general theory for mixed-level representations (analogous

to Boolean algebra for the gate level) makes it difficult to define and correctly solve problems related to test synthesis and analysis, fault masking, and test quality for complex digital systems. Different test design tools (test generation, two- or multivalued simulation, test quality analysis, statistical fault grading, testability analysis, and so on) need different models to represent design components. These tools represent components as functions, algorithms, operators, or rules (given in different languages) for solving corresponding component test design subtasks.

Consequently, we have to develop many component model libraries to support various test design tools. Purchasing or updating these libraries adds to CAD system costs.

I propose a new approach based on using alternative graphs (AGs) to create tools for computer-aided test design of digital systems. AGs serve as a mathematical basis for solving a wide spectrum of test design tasks using a single model base and a restricted set of procedures (horizontal or cross-task universality of AGs). Figure 1 depicts AND gate models for different test design tasks; each task requires different component libraries. In contrast, a single library of AGs will suffice for all these tasks.

Unlike analog binary decision diagrams,^{6,9} AGs represent, in a compressed form, the topology of gate level circuits. They therefore directly support test design for gate level structural faults without representing them explicitly. Binary decision diagrams do not represent the topology and therefore using them leads to functional tests that may not adequately detect structural faults.

In addition, AGs support a uniform approach to digital test design at several system levels (vertical or cross-level universality of AGs), whereas binary decision diagrams support only the Boolean level. I earlier proposed the

use of AGs for test generation in digital circuits¹⁰ and later formulated a generalized approach for high-level tests.¹¹⁻¹³

Model

AGs may represent a set of digital (Boolean or integer) functions $y = F(X)$ of components or subcircuits in digital systems. Here, y is an output variable, and X is a vector of input variables of the component or subcircuit.

Definition 1. In the general case, an AG that represents function $y = F(X)$ is a directed, noncyclic graph $G_y = (M, \Gamma, X)$ with set of nodes M , single root node $m_0 \in M$, and relation Γ in M , where $\Gamma(m) \subset M$ denotes the set of successor nodes of m . Nonterminal nodes m for $\Gamma(m) \neq \emptyset$ have variables $x_i \in X$ as labels. Terminal nodes m for $\Gamma(m) = \emptyset$ have variables x_i , functional subexpressions of $F(X)$, or constants as labels. Let $x(m)$ be the label of node m . In graph G_y , for all nonterminal nodes m for which $\Gamma(m) \neq \emptyset$, a one-to-one correspondence exists between the values of label variable $x(m)$ and the successors, $m_k \in \Gamma(m)$ of m .

Representing Boolean functions.

Consider a special case of AGs that represent Boolean functions and digital circuits at the logical level.

Definition 2. An AG that represents a Boolean function $y = f(X) = f(x_1, x_2, \dots, x_n)$ is a binary decision diagram in which (inverted or noninverted) Boolean variables x_i , ($i = 1, 2, \dots, n$) label nonterminal nodes, and constants 0 or 1 label terminal nodes.

Let $m^0 \in \Gamma(m)$ denote the successor of m that corresponds to value $x(m) = 0$ and $m^1 \in \Gamma(m)$ denote the successor that corresponds to value $x(m) = 1$.

Definition 3. We call an output edge from m to m^e , $e \in \{0, 1\}$, activated when label variable $x(m)$ has value e . A path in an AG is activated if all the edges

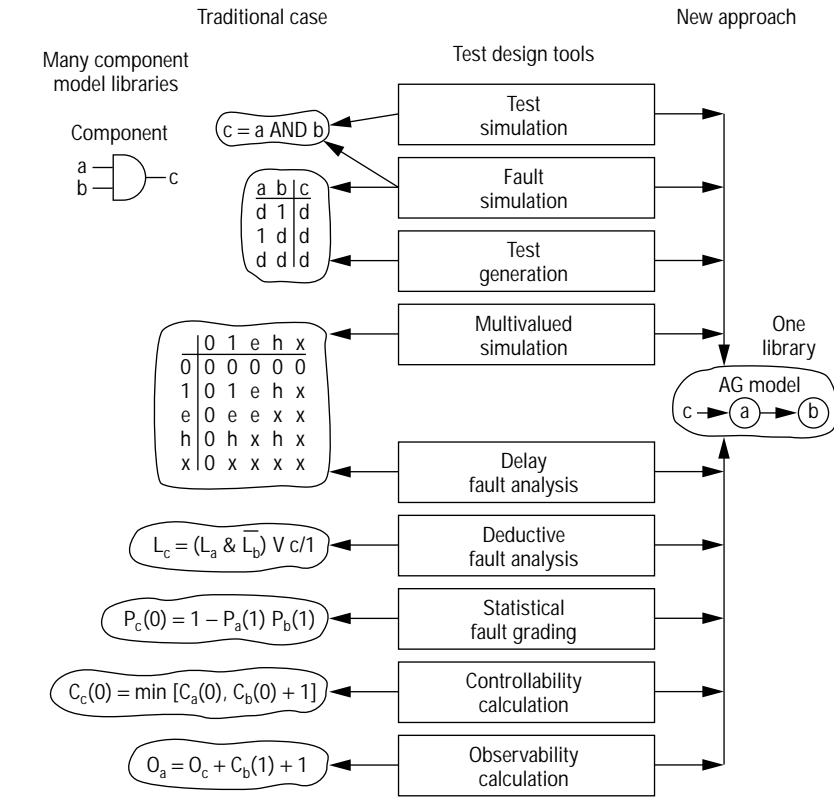


Figure 1. Horizontal (cross-task) universality of AGs.

About Estonia

Capital: Tallinn
 Area: 45,100 km²
 Population: 1,625,399
 Chief of state: President Lennart Meri

In 1918, original course offerings at Tallinn Technical University (<http://zaphod.cc.ttu.ee/>) included mechanical, electrical, civil, and hydraulic engineering; shipbuilding; and architecture.



Today, economics, information technology, and mechanical engineering are the most popular majors.

forming this path are activated. An AG is activated to the value 0 (or 1) if there exists an activated path that includes both the root node and the terminal node labeled by the constant 0 (or 1).

Definition 4. AG G_y with nodes la-

beled by variables x_1, x_2, \dots, x_n represents Boolean function $y = f(X) = f(x_1, x_2, \dots, x_n)$, if for each pattern for X , the AG will be activated to the value that equals y .

Using definitions 3 and 4, we can calculate y with graph G_y . For that purpose, we trace a path in the AG

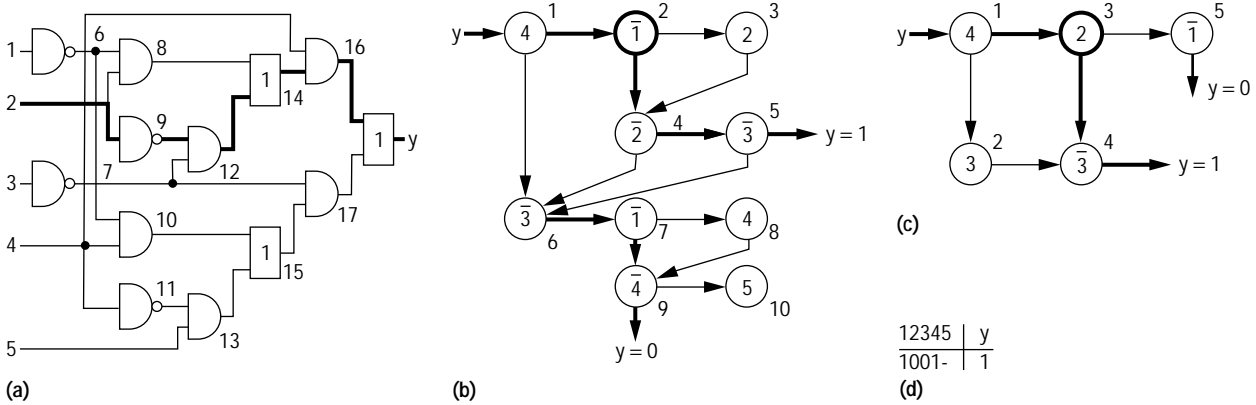


Figure 2. Combinational circuit (a), corresponding structural AG (b), and functional AG (c) with example test pattern (d).

activated by values of $x \in X$ and find this path's terminal node. The constant at this node is the value of y .

Structural AGs. Consider a digital system as a network of components, each of which executes one or more Boolean functions. Consequently, we can represent a digital system by a set of corresponding AGs. For a gate level description, the number of AGs equals the number of gates in the circuit. However, we can decompose a given gate level circuit into subcircuits and represent subcircuit functions by AGs, thus obtaining a compressed AG description of the circuit.

Example 1. Figures 2b and 2c show two AG representations of a combinational circuit (Figure 2a). The labels $x(m) \equiv i$ inside nodes denote input variables i of the circuit. For simplicity, Figures 2b and 2c omit activating values of variables on edges (by convention, the activation value for leaving a node to the right is 1; downward, 0). It also omits terminal nodes with constants 0 and 1. (By a similar convention, leaving the AG node to the right corresponds to $y = 1$; from the bottom, $y = 0$.) To create AGs that satisfy these conventions, we may use inverted variables as node labels.

To generate structural AGs, consider

the following procedure for superposition of graphs analogous to superposition of functions. If label variable $x(m)$ of node m in G is a function represented by another AG, $G_{x(m)}$, we can replace node m with graph $G_{x(m)}$.¹⁰ By starting from the AG of the output gate in a gate level AG description and iteratively using this superposition procedure, we can compress the circuit's AG representation. (Each substitution reduces the model by one node and one AG.) It is evident that we may not compress the model beyond fan-out points. Therefore, to maximize AG model compression, we should only generate AGs by superposition for treelike subcircuits. That is, we stop the superposition procedure at nodes labeled by fan-out variables that correspond to the original circuit's fan-out nodes. AGs created by superposition are structural AGs.

Structural AGs have an important property:¹⁰ Each node m in a structural AG G that describes treelike subnetwork S_G of circuit S represents a signal path L_m in S_G . The one-to-one correspondence between nodes m and paths L_m results from superposition. Let us show it recursively. For structural AG G (as just described), all label variables $x(m)$ in G correspond to inputs of S_G , and L_m denotes the path in S_G from input $x(m)$ to the output of S_G . Suppose variable $x(m)$ is an output variable of component $K_{x(m)}$ in S . Superposition of

node m in G with graph $G_{x(m)}$ means including $K_{x(m)}$ into tree S_G and extending path L_m by inputs $x(m^*)$ of $K_{x(m)}$ for all new nodes m^* in G labeled $x(m^*)$.

The nodes of a structural AG correspond one to one with the paths in the associated subnetwork. Thus it follows that the number of all nodes in the set of structural AGs representing a given circuit equals the number of paths in all the circuit's treelike subnetworks.

Using structural AGs, it becomes possible to rise from the gate level description of a digital circuit to a higher level of compressed structural description and accurately represent gate level stuck-at faults. In fact, all gate level faults along signal path L_m will collapse through superposition to the two representative faults of label $x(m)$ at node m . Hence, we can simulate structural, stuck-at faults in the path of a gate level circuit by simulating faults at a node in the structural AG.

Example 2. Figure 2b depicts an example structural AG for the circuit in Figure 2a. Node $m = 4$ (labeled by the variable $x(4) \equiv \bar{2}$) represents the lower path in the original circuit (indicated by heavyweight lines). The path extends from 2_2 (which denotes the lower branch at input 2) to circuit output y : $L_4 = (2_2, 9, 12, 14, 16, y)$. Node 3 in the structural AG (labeled $x(3) \equiv 2$) represents

the path from upper branch 2_1 of input 2 to output y : $L_3 = (2, 8, 14, 16, y)$.

Functional AGs. Another way to generate logical-level AGs uses implementation-free descriptions of digital devices (Boolean expressions, truth tables, and so on). In this case, we can use the methods developed for binary decision diagram synthesis.^{6,7} Since we do not derive these graphs from the structure and they represent only the circuit's function, we call them functional AGs. Unlike structural AGs, there is no one-to-one correspondence between the AG nodes and paths in a circuit.

Example 3. Figure 2c depicts an example of a functional AG for the given circuit. Node 3 in the functional AG (labeled $x(3) \equiv 2$) does not represent any path of the original circuit. Hence, in general, functional AGs do not represent structural faults inside circuits. However, we may use them to obtain concise functional descriptions.

Dynamic combination of functional and structural AGs can contribute to efficient test generation or fault simulation for large digital circuits. In general, functional AGs afford a more concise description than structural AGs. We can therefore use them to solve justification tasks or generate sensibility conditions for fault propagation between inputs and outputs in black-box components. Structural AGs represent implementation-dependent faults in components or subcircuits. State transition and output functions represent sequential parts in digital circuits. We may in turn represent such parts of circuits with either structural or functional AGs.

ATPG. A promising trend in automatic test pattern generation combines symbolic techniques based on binary decision diagrams and traditional topological algorithms.¹⁴ The AG approach provides a uniform theoretical basis for that methodology. In such a case, we should

base topological algorithms on structural AGs and use functional AGs to support the symbolic technique. The experimental results presented later show that topological ATPG based on structural AGs is about two to five times more efficient than using a gate level topology.

General case of AGs. Consider a digital system as a dynamic system $S = (Z, F)$. Here Z is a set of digital variables z (the numbers $V(z)$ of possible values for $z \in Z$ are finite), and F is a set of digital functions on Z . An AG model represents system S such that for each function $z_k = f_k(Z_k)$, $z_k \in Z$, $f_k \in F$, and $Z_k \subset Z$, there exists an AG, G_k . Depending on the class of digital system (or level of its representation), we may develop various AGs in which nodes have different interpretations and relationships to the system structure. These relationships directly influence the potential for representing high-level (functional) faults of systems using AGs.

In Boolean representations, Boolean variables, expressions, or constants label all AG nodes. In structural AGs, each node represents a signal path in the corresponding logic circuit. In register transfer level (RTL) descriptions, we usually decompose digital systems into control and data parts. State and output variables of the control part serve as addresses and control words, and the variables in the data part serve as data words. High-level data word variables describe RTL component functions in data parts. We can transform RTL functions $z_k = f_k(Z_k)$, where z_k and $z \in Z_k$ are either Boolean or integer variables, into AGs.

For example, either Boolean variables (representing logical conditions) or integer variables (representing current states) label nonterminal nodes in AGs that represent state transition functions of control parts. On the other hand, integer constants (next states or addresses) or expressions for calculating next states label the terminal nodes. In AGs

that represent components or subcircuits of data parts, Boolean variables representing 1-bit control signals or integer variables representing control words or fields of control words label the nonterminal nodes. Terminal nodes carry integer constants, integer data variables, or subexpressions of functions $z_k = f_k(Z_k)$ as labels. These subexpressions represent microoperations or operations.

In general, terminal nodes in AGs labeled by expressions represent data manipulation functions and the corresponding subcircuits of the data part. Terminal nodes labeled by variables represent buses or registers. AG subgraphs consisting of nonterminal nodes represent control functions, whereas nonterminal nodes labeled by Boolean variables represent signal paths in the control circuit (in structural AGs). Nonterminal nodes labeled by integer variables represent decoders.

Example 4. Figure 3b (next page) depicts an example AG for a digital circuit (Figure 3a) with function Out represented at mixed gate and register transfer levels:

```

if  $x_4 = 1$  then
  if  $I = 1$  then  $F(R, In)$ 
  else if  $I = 2$  then  $R$ 
  else if  $I = 3$  then
    if  $(x_1 \text{ AND } x_2) \vee (\neg x_1 \text{ AND } x_3) = 1$  then In
    else 0
  else 0
else 0

```

Here $x_1, x_2, x_3,$ and x_4 denote Boolean variables; In (input data), R (register data), and Out (output data) are integer variables; $F(R, N)$ represents an RTL data manipulation expression or subexpression.

For nodes with Boolean variables, Figure 3b (next page) omits edge values and uses the same convention as Figures 2b and 2c (see Example 1). It also omits terminal nodes with integer

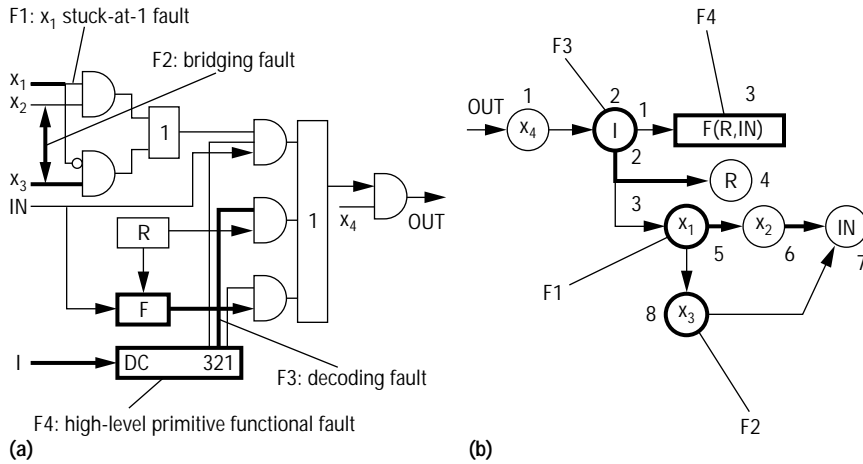


Figure 3. Digital circuit (a) and corresponding mixed-level AG representation (b).

constants 0. The graph allows easy simulation of circuit behavior. For example, in the case of mixed-value pattern 1,1,0,1,3,47 ($x_1, x_2, x_3, x_4, I, In$), Out equals 47, and simulation will trace activated path $L = (1,2,5,6,7)$.

Circuit structure is also easy to see in the AG representation. For example, node 6 (labeled x_2) represents a signal path from input x_2 to output Out of the circuit. The AG collapses stuck-at faults along this path into the two representative faults at node 6. Node 2 (labeled I) represents decoder DC with all the signal paths from DC to Out. The AG collapses all the decoding faults and stuck-at faults in this part of the circuit into the faults at node 2.

Hierarchical descriptions. We may easily produce hierarchical AG-based descriptions, since low-level AGs can be derived and represent implementation details for functions given as labels of high-level AG nodes. This also allows us to easily move from level to level. Two means of descending from higher to lower levels exist.¹²

First, we represent integer variable $I(m)$ in nonterminal node m with $N = |\Gamma(m)|$ output edges as a concatenation of subvariables I_i . In this case, we exchange node m for a decision tree

with nodes labeled by subvariables I_i that have the same number, N , of output edges. For example, we can exchange node m with N output edges labeled by an n -bit variable for a binary tree with N output edges. For another example, we can split instruction word I into subfields I_1, \dots, I_k and represent nodes labeled I using a decision tree with nodes labeled I_1, \dots, I_k .

Second, the concatenation of N 1-bit subexpressions can replace the N -bit expression in a terminal node. In this case, we can exchange the terminal node with a tree that has N terminal nodes labeled by 1-bit expressions. Lower-level graphs that more precisely represent implementation details of corresponding functions can also replace terminal nodes.

Fault models. In AGs, a uniform fault model replaces fault models defined at different digital system representation levels. The uniform model covers the following faulty cases

- the edge of a node is always activated
- the edge of a node is broken
- instead of the given edge, another edge or set of edges of a node is activated

This model leads to exhaustive testing of a node's functionality. Model complexity depends on the level of system representation—that is, on the functionality of a node. Each path in an AG describes the system behavior in a specific operation mode. Faults affecting the behavior relate to nodes along the given path and cause an erroneous change in the path activated by the test. The physical meaning of faults associated with node outputs depends on the relationship between the node and circuit.¹² Depending on the adequacy of the system structure representation, the proposed fault model can cover a wide class of structural and functional faults. We can regard the fault model defined for AGs as a generalization of the classical gate level stuck-at fault model. Fault objectives for the classical case are Boolean variables (or literals); fault objectives for the AG approach are AG nodes.

Let us interpret some well-known low- and high-level fault models using AG notation. We treat the gate level stuck-at fault model at the logical level as a special case of faults in AGs: those at nodes labeled by Boolean variables. (In structural AGs, these faults represent a whole class of faults along corresponding signal paths in circuits.) AGs represent RTL statement faults—such as label, timing or logical condition, register or function decoding, and control faults¹—as faults at nonterminal nodes with corresponding labels, conditions, decoding variables, or control variables. Faults in AGs at terminal nodes with register variables or data manipulation expressions represent RTL statement faults such as data storage, transfer, and manipulation faults.¹ A single class of faults defined for an AG node uniformly represents the fault classes introduced by Thatte and Abraham¹ for microprocessors.

Representing bridging and cross-talk faults in AGs is also possible. However such representations will only be as accurate as the level of detail supported by the AG notation. For example, if

nodes m_1 and m_2 in the AG represent paths L_{m_1} and L_{m_2} in the circuit, then we simulate cross-talk or bridging faults between these paths by values of variables $x(m_1)$ and $x(m_2)$. Since AG nodes represent signal paths in treelike subcircuits, we may also use AGs to simulate and analyze delay faults in these paths.

Example 5. In Figure 3b, the F1 fault of node 5 represents the stuck-at-1 faults of input x_1 in Figure 3a. Since node 6 represents the path from x_2 to Out in the corresponding circuit (Figure 3a), both of the node 6 stuck-at faults represent all the stuck-at faults of this path in the circuit. Choosing different values for x_2 and x_3 activates the F2 bridging fault between input leads x_2 and x_3 . We may observe the fault as a change in one of these values; for example, x_3 . In Figure 3b, then, F2 would affect node 8 (labeled x_3). Moreover, for AGs, since nodes 6 and 8 represent the circuit paths from x_2 and x_3 to Out, the AG represents a whole set of bridging faults between these paths simultaneously.

The F3 faults of node l in the AG (Figure 3b) represent decoder DC's functional faults and stuck-at faults on the paths from DC to Out. F4 faults at node 3 represent faults in block F . As the representation includes no more details of function $F(R, \text{In})$, we require an exhaustive test for it. To use a hierarchical test approach, we exchange node 3 for a structural AG of block F and, instead of an exhaustive test, may derive a shorter test from the structural AG.

In Figure 2b, structural AG fault 4/1 (a stuck-at-1 fault at node 4) represents a class of faults ($2_2/0$, $9/1$, $12/1$, $14/1$, $16/1$, and $y/1$) along the bold path of the corresponding circuit. Testing node 4 in this structural AG is equivalent to testing all the faults in this class of gate level faults. For functional AGs as well as binary decision diagrams, in general, it is impossible to find relationships between faults in the graph and faults in the circuit (except for input faults).

Test pattern design

The AG model is well suited to solving a set of test design tasks: test generation (fault propagation, line justification, implication), test analysis (simulation, fault cover calculation, multivalued simulation), testability calculation, and so on. All these tasks reduce to using a few, standard path-tracing procedures on AGs. Traditionally, we had to create different component model libraries to solve these tasks. As an example, Figure 1 depicts models of AND gates used for different test design tasks. Using the AG approach, we need only one AG library.

Simulation (logical level). Test pattern simulation on AGs is equivalent to tracing paths on graphs according to variable values for a given test pattern. As a result of path tracing in G_y , we calculate the y that equals the the label function value at the terminal node. For example (Figure 2b) for test pattern 1001- (1,2,3,4,5), we trace the path through nodes 1, 2, 4, and 5 in the structural AG (or nodes 1, 3, and 4 in the functional AG) that yields $y = 1$.

As initially presented, we may only analyze faults inside a circuit using structural AGs (and not arbitrary binary decision diagrams). Now, let $l(m)$ be the activated path in the AG from the root node to node m ; $l(m,1)$ or $l(m,0)$ be the activated path from node m to the terminal node labeled by constants 1 or 0; and m^1 or m^0 be the successor of node m for $x(m) = 1$ or $x(m) = 0$. Use the notation $l(m) = 1$ or $l(m,e) = 1$ if a path $l(m)$ or $l(m,e)$ exists where $e \in \{0,1\}$; otherwise, $l(m) = 0$ or $l(m,e) = 0$.

We may regard fault analysis as the process of calculating the values of Boolean derivatives: $dy/dx = 1$ indicates that a fault at x is detected at y . For AGs, $dy/dx(m) = 1$ is equivalent to either

$$l(m) \wedge l(m^1,1) \wedge l(m^0,0) = 1 \quad (1)$$

$$l(m) \wedge l(m^1,0) \wedge l(m^0,1) = 1 \quad (2)$$

In other words, $dy/dx(m) = 1$ is equivalent

to the existence of three simultaneously activated paths: $l(m)$, $l(m^1,1)$ or $l(m^1,0)$, and $l(m^0,0)$ or $l(m^0,1)$. Let us denote $dy/dx(m)$ for simplicity by dy/dm . For example, for the pattern in Figure 2d, $dy/d4$ equals 1 for the structural AG (Figure 2b), since the following three paths are activated: $l(4) = (1,2,4)$, $l(5,1) = (5,1)$, $l(6,0) = (6,7,9,0)$, fulfilling the condition of Equation 1. (In this notation, boldface numbers are constants rather than node numbers.) To calculate the faults detected by a given test pattern, the simulation procedure traces activated path l in the structural AG from the root node to a terminal node. We then calculate Boolean derivative values for each node on this path.

Example 6. The test pattern in Figure 2d activates structural AG path $l = (1,2,4,5)$. For node 2, we have $dy/d2 = 0$, because the pattern does not fulfill either Equation 1 or 2: $l(m) = l(2) = 1$, $l(m^1,1) = l(3,1) = 1$, and $l(m^0,0) = l(4,0) = 0$. For other nodes m in this path, we have $dy/dm = 1$ (each of these nodes fulfills Equation 1). Hence, the test pattern detects 1/0, 4/0, and 5/0 faults on the structural AG. These faults represent the following fault classes: ($4_1/0$, $16/0$, $y/0$), ($2_2/1$, $9/0$, $12/0$, $14/0$, $16/0$, $y/0$), and ($7_1/0$, $12/0$, $14/0$, $16/0$, $y/0$). The intersection of the three classes is the set of gate level faults detected by the pattern. Here, the subscript at fan-out variables denotes the number of the fan-out branch, with branches enumerated from the top (branch 1) to the bottom.

In the same way, for the functional AG, we detected 3/1 and 4/0 faults, which correspond to faults 2/1, 3/1, and 7/0 on the circuit inputs. As the functional AG represents only the circuit function, it is not possible to detect internal faults by simulating test patterns on it. On the other hand, faults at fan-out inputs in the circuit manifest themselves as multiple faults on the structural AG. For example, circuit fault 4/1 is equivalent to multiple fault (1/1, 8/1, 9/0).

Thus, functional AGs are useful for simulating faults at fan-out nodes and for propagating fault effects through subcircuits. Structural AGs are appropriate for representing, collapsing, and simulating gate level faults inside corresponding subcircuits. We have implemented fault analysis methods such as deductive analysis and parallel, critical-path tracing¹² based on AGs. Methods and algorithms based on AGs also exist and for fault diagnosis.¹⁵

Pattern generation (logical level).

Line justification is a subtask of test pattern generation and a reversal of the simulation procedure. To justify line y to value D , we must solve a corresponding equation $y=F(X)=D$. Using the AG model, line justification becomes a graph (or path) activation task. To solve $y=D$ in graph G_y , we must activate G_y to value D . That is, we create a pattern that activates a path in the graph from the root node to a terminal node with a label function (or constant) equaling D .

Path activation procedures form the basis for test pattern generation on AGs. To generate a test for node m , we must solve one of the conditions of Equations 1 and 2 (that is, simultaneously activate three nonoverlapping paths).

Example 7. In Figure 2b, to test faults at node 4, we activate paths $l(m) = (1,2,4)$, $l(m_1,1) = (5,1)$, and $l(m_0,0) = (6,7,9,0)$. This yields test pattern 1D01-(1,2,3,4,5), where $D = 0$ for stuck-at-1 faults, and $D = 1$ for stuck-at-0 faults. We activated the paths and generated the pattern without backtracking. For the structural AG in Figure 2b, we find patterns for testing nodes 1 to 6 and 10 also without backtracking. When determining the pattern for fault 7/1, we carry out the following path activation procedure (path tracing) and only backtrack once. In the path-tracing notation, the numbers in parentheses indicate the node, node variable, and chosen variable value for a step. If the value of the variable is already

set, we indicate only the node. Symbol \emptyset indicates an inconsistency between the constant reached by the path and the constant desired at a terminal node.

- To detect 7/1, we set label variable $\neg 1$ to 0.
- Then, for $l(m) = l(7)$, we trace $(1,4,0) \rightarrow (6, \neg 3,1) \rightarrow 7$, and include node 1 in the backtrack list.
- For $l(m^1,1) = l(8,1)$, we trace $8 \rightarrow 9 \rightarrow (10,5,1) \rightarrow 1$. For $l(m^0,0) = l(9,0)$, we trace $9 \rightarrow 10 \rightarrow 1\emptyset$.
- Because no solution exists, we backtrack to node 1 and for $l(m) = l(7)$ trace $(1,4,1) \rightarrow 2 \rightarrow (4, \neg 2,0) \rightarrow (6, \neg 3,1) \rightarrow 7$. We include node 4 in the backtrack list.
- For $l(m^1,1) = l(8,1)$, we trace $8 \rightarrow 1$. For $l(m^0,0) = l(9,0)$, we trace $9 \rightarrow 0$; which is consistent. The test pattern is then 1101- (1,2,3,4,5).

The number of variables to be set during the test generation procedure determines the search space in which backtracking occurs. For structural AGs, we use fewer variables than in the gate level case; hence, the search space and amount of backtracking also decrease.

Fault propagation. This process is analogous to that for test generation, but instead of structural AGs, we may use more compact functional AGs. For example, to propagate faults at input 2 through the circuit in Figure 2a, we generate a test pattern for node 3 by activating paths $l(m) = (1,3)$, $l(m_0,1) = (4,1)$, and $l(m_1,0) = (5,0)$. The functional AG in Figure 2c does not require backtracking.

Test generation on higher levels.

Path activation principles also form the basis for test generation in the general case of AGs. Consider an AG, G , representing function $z_k = f_k(Z_k)$ in which labels of nonterminal nodes are Boolean or integer variables. Either integer variables or subexpressions of $z_k = f_k(Z_k)$ label terminal nodes.

Conformity test. Let $l(m^i, m^{T,i})$ denote the activated path from node m^i to terminal node $m^{T,i}$. To generate a test for nonterminal node m , we must

1. find a symbolic test pattern by simultaneously activating nonoverlapping paths $l(m)$ and $l(m^i, m^{T,i})$ for all values $i \in V[z(m)]$ of variable $z(m)$, and
2. find data values by solving one of the following equations (depending on the technology used) for each bit of data words:

$$\forall i, j \in V[z(m)] (j \neq i):$$

$$\neg z(m^{T,i}) \wedge z(m^{T,j}) = 1 \quad \text{or} \quad (3)$$

$$z(m^{T,i}) \wedge \neg z(m^{T,j}) = 1 \quad (4)$$

Fulfilling these conditions guarantees that deviation from the path activated by the test pattern produces another path. This path has a new terminal node with a label function value that differs from what we expect from the test pattern. If node m represents a decoder, the test produced by solving Equations 3 and 4 will detect functional faults in the decoder and stuck-at faults on the decoder's control leads. In general, solving Equation 3 or 4 may require several sets of data operands for each value of i . We test node m using the symbolic test pattern generated by substituting the symbolic value of $z(m)$ by all $i \in V[z(m)]$. We also repeat this test for all data that are solutions of Equations 3 or 4. Such testing, called conformity testing, aims to detect control faults.

Scan test. To generate a test for terminal node m labeled by a data variable or subexpression of function $z_k = f_k(Z_k)$, we must again create a symbolic test pattern. We do this by activating a path from the root node to terminal node m (step 1). Here, values of data $z \in Z_k$ are symbols. Step 2 generates test values for $z \in Z_k$ based on function $f_k(Z_k)$ or on a lower-level implementation of $f_k(Z_k)$. We repeat the symbolic test pattern of step

1 for all operand values found in step 2. Such testing, called scan testing, aims to detect faults in the data part.

Example 8. To test node $m = 2$ in Figure 3b, we activate paths $l(m) = (1,2)$, $l(m^1, m^{T.1}) = (3)$, $l(m^2, m^{T.2}) = (4)$, $l(m^3, m^{T.3}) = (5,6,7)$. Solving Equation 3 for $F(R, In)$, R , and In of terminal nodes, for each bit of data words yields the pattern $1, 1, 1, R^*, In^* (x_1, x_2, x_4, R, In)$. Solving the equation yields R^* and In^* . We must repeat this pattern for all values of $I = 1, 2, 3$. This test detects functional faults in the DC decoder and stuck-at faults on paths between it and Out.

Other literature^{11,12} discusses the details of fault propagation, line justification, and test generation at various system representation levels.

Implementation results

On the mathematical basis of AGs, my colleagues and I developed computer-aided tools for solving different test design tasks. Test design software for digital networks at the gate level¹⁶ consists of tools for test generation, fault coverage analysis, multivalued simulation (for hazard and dynamic test analysis), fault detection probability, and testability analysis. The fault classes considered include stuck-at and transition faults (delay and stuck open). Our software requires an IBM PC or PC-compatible computer, runs on MS-DOS (version 3.3 or higher), and also executes in an MS Windows environment. The system supports designs from various CAD environments, including those of Synopsys, Cadence, OrCAD, and ViewLogic (or packages such as AsyI+ and Dixi-CAD).

Table 1 lists a comparison of test generation efficiency for ISCAS85 benchmarks using gate and macro level AGs. (These benchmarks are from the 1985 International Symposium on Circuits and Systems.) Because of fault collapsing and model compressing, the number of target faults decreases 1.4 to 1.8 times, and test generation time decreases

Table 1. Comparison of test generation efficiency for gate and macro level AGs.

Circuit	Level	No. of target faults	CPU time (s)	No. of patterns	Fault coverage (%)	
					Gate level	Macro level
c499	Macro	1,202	131	106	99.6	99.3
	Gate	2,194	662	109	99.5	—
c880	Macro	994	46	112	98.6	98.6
	Gate	1,550	119	101	98.1	—
c1355	Macro	1,618	278	105	99.6	99.5
	Gate	2,194	953	107	99.5	—
c1908	Macro	1,732	219	169	99.0	98.5
	Gate	2,788	743	160	99.0	—

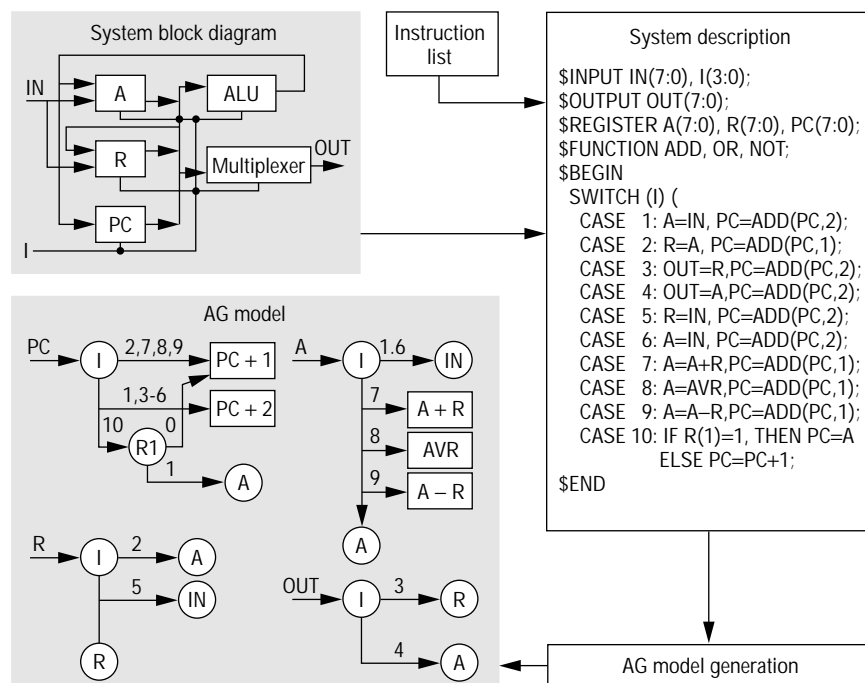


Figure 4. AG generation for a hypothetical microprocessor.

es 2.6 to 5.1 times. These results are for a compressed AG model in which structural AGs represent treelike subcircuits, and we obtain them using an IBM PC 486, 66-MHz platform.

AG-based ATPG for functional testing of microprocessors (Figure 4) consists of an AG synthesizer, symbolic test pattern generator, and test program compiler (Figure 5, next page). Starting with an

RTL description of the instruction list, the synthesizer creates an AG model of the processor. The ATPG system allows test engineers to manually write symbolic subroutine templates (in an automatic test equipment language) to apply test patterns.

The system will exchange symbols in these subroutines for the real test data it generates. Each symbolic test pattern

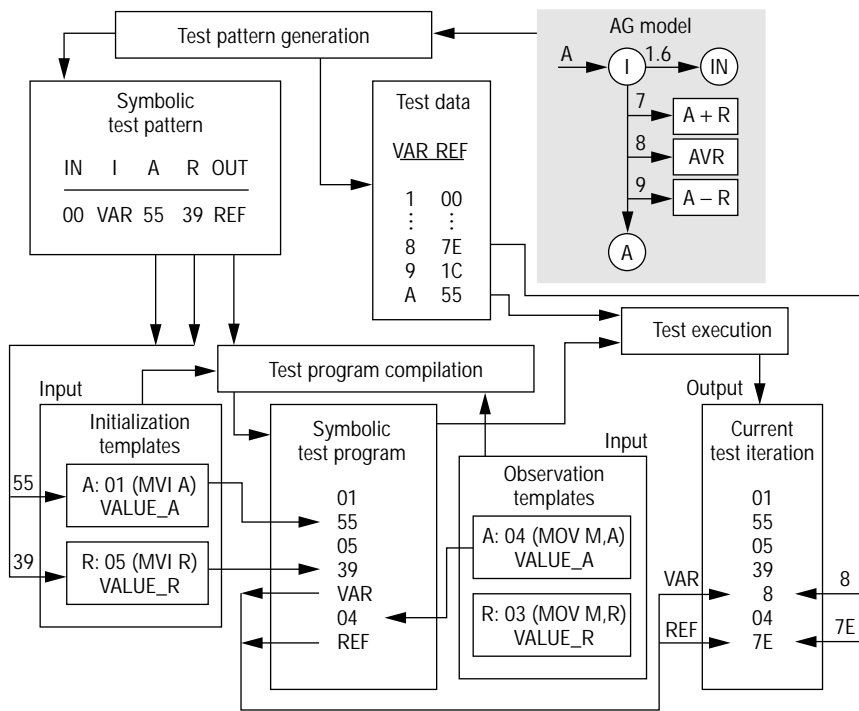


Figure 5. Test pattern generation for part of a hypothetical microprocessor.

Table 2. Structural characteristics of benchmark circuits.

Circuit	No. of gates	No. of inputs	No. of outputs	No. of fan-outs	No. of target faults
4	603	42	5	879	2,728
8	1,195	74	9	1,711	5,360
16	2,379	138	17	3,375	10,624
32	4,747	266	33	6,703	21,152

tests an AG node, and we divide test patterns into conformity and scanning patterns. Conformity tests apply to internal AG nodes that represent the control part of the system, whereas scanning tests apply to terminal AG nodes that represent the data part. Figure 5 gives an example of conformity test generation for node *I* in graph *A*. (This tests the instruction decoder in the data manipulation block by observing register *A*).

Our experience using this ATPG system showed that high-level test design for Intel 8080, 8086, and 8286 micro-

processors takes about one man-month (including the time to describe the processor). Automated test generation for the Intel 8080 took about 2 CPU minutes on an IBM PS/2. The 573-pattern test included data arrays with a total length of 3,412 words. Due to our technique's compact data representation, memory space was 67.5 times less than that required for conventional representations.

This ATPG covers the fault classes proposed by Thatte and Abraham.¹ We were unable to evaluate test quality for gate level fault detection because of the

absence of gate level descriptions.

We also investigated the adequacy of high-level fault models defined for AGs and the potential for high-quality gate level tests using only high-level descriptions. Experiments for a restricted class of digital systems used benchmarks based on a family of 4-, 8-, 16-, and 32-bit simplified RISC processors. We implemented only arithmetical (based on adders) and logical operations (a total of 8 instructions or 512 working modes) and examined only the RISC processors' combinational parts. Table 2 summarizes the structural characteristics for four benchmark circuits.

We created two AG models for each circuit: compressed structural AGs for treelike subcircuits (for structural, gate level test generation) and high-level AGs (for functional, high-level test generation). Table 3 lists the results of test generation experiments. The number of functional test patterns was independent of the processor's word length since we based the arithmetical part of the ALU on a ripple-carry adder. Both structural and functional test generation cases failed to detect only two redundant faults in the control part.


TESTING AND DIAGNOSIS of digital electronic systems face many problems that result mainly from system complexity. One solution is to hierarchically test systems, which reduces the complexity of the task. Hierarchical testing, however, lacks a general theory for diagnosing systems by uniform methods on various levels. AGs provide a way of representing diagnostic information uniformly for different system levels. They serve as a basis for a general theory of test design and fault diagnosis for mixed-level digital systems.

We can simultaneously regard AGs as a procedural notation (a program), a data structure (decision tree) to be processed, a representation of diagnostic knowledge, or as a compact way to

Table 3. Functional (high-level) and structural (gate level) test generation results.

Circuit	Level	No. of test patterns	Fault coverage (%)	Test generation time (s)	No. of target faults
4	High	126	99.86	0.1	2,728
8		126	99.93	0.4	5,360
16		126	99.96	1.2	10,624
32		126	99.98	4.3	21,152
4	Gate	111	99.93	31.1	1,522
8		140	99.97	109.8	2,970
16		169	99.98	440.0	5,866
32		274	99.99	2,584.6	11,657

represent all possible test modes for the system. The various interpretation possibilities allow the use of AGs in many applications related to digital design, test synthesis, fault diagnosis, test knowledge compression, and test processing.

A promising trend in ATPG is joining symbolic techniques based on BDDs and traditional topological algorithms. AGs offer a uniform theoretical basis for such an approach. Combining symbolic and topological techniques with high-level functional approaches while using a common mathematical basis would increase ATPG efficiency. 

Acknowledgments

The Estonian Science Foundation (under Grant 1433) and the European Community (under Copernicus JEP 9624 and ESPRIT III basic research JEP 6575) supported this work. I thank my colleagues E. Ivask, G. Jervan, A. Markus, P. Paomets, and J. Raik, who participated in developing the software and conducting experiments.

References

1. S.M. Thatte and I.A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. Computers*, Vol. 29, 1980, pp. 429-441.
2. A.G. Gupta and J.R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices," *Proc. ACM/IEEE 22nd Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp. 720-726.

3. W. Geiselhardt, W. Mohrs, and U. Moeller, "FUNTEST-Functional Test Generation for VLSI-Circuit and Systems," *Microelectronics Reliability*, Vol. 29, No. 3, Mar. 1989, pp. 357-364.
4. D. Bhattacharya and J.P. Hayes, "A Hierarchical Test Generation Methodology for Digital Circuits," *JETTA: Theory and Application*, Vol. 1, 1990, pp. 103-123.
5. J. Lee and J.H. Patel, "Hierarchical Test Generation Under Intensive Global Functional Constraints," *Proc. 29th ACM/IEEE Design Automation Conf.*, IEEE CS Press, 1992, pp. 261-266.
6. C.Y. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell System Technology J.*, Vol. 38, No. 7, July 1959, pp. 985-999.
7. S.B. Akers, "Binary Decision Diagrams," *IEEE Trans. Computers*, Vol. 27, No. 6, July 1978, pp. 509-516.
8. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, Vol. C-35, No. 8, Aug. 1986, pp. 667-690.
9. B. Becker and R. Drechsler, "How Many Decomposition Types Do We Need?" *Proc. European Design and Test Conf.*, IEEE CS Press, 1995, pp. 438-443.
10. R. Ubar, "Test Generation for Digital Circuits Using Alternative Graphs," (in Russian), *Proc. Tallinn Technical Univ.*, No. 409, Tallinn Technical Univ., Tallinn, Estonia, 1976, pp. 75-81.
11. R. Ubar, "Test Pattern Generation for Digital Systems on the Vector AG-Model," *Proc. 13th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1983, pp. 347-351.

12. R. Ubar, "Alternative Graphs and Technical Diagnosis of Digital Devices," (in Russian), *Electronic Technique*, Vol. 8, No. 5, May 1988, pp. 33-57.
13. *Fehler in automaten*, [Faults in Automata], D. Bochmann and R. Ubar, eds., VEB Verlag Technik, Berlin, 1989.
14. F. Corno et al., "Improving Topological ATPG with Symbolic Techniques," *Proc. IEEE VLSI Test Symp.*, IEEE CS Press, 1995, pp. 338-343.
15. R. Ubar, "Fault Diagnosis in VLSI Devices," *Proc. Estonian Acad. of Sciences, Engineering*, No. 1, Estonian Acad. of Sciences, Tallinn, Estonia, 1995, pp. 51-67.
16. R. Ubar et al., "A PC-Based CAD System for Training Digital Test," *Proc. Fifth Eurochip Workshop on VLSI Design Training*, CMP/Eurochip, Grenoble, 1994, pp. 152-157.



Raimund Ubar is a professor of computer engineering and head of the Electronics Competence Centre at Tallinn Technical University (TTU) in Estonia. His research interests include computer diagnostics, test pattern generation for digital systems, fault simulation, design for testability, and built-in self-test. Ubar received his MS in control engineering from TTU and PhD in computer engineering from Moscow Technical University. He is a member of the IEEE, Gesellschaft der Informatik (Information Society, Germany), European Test Technology Technical Committee, and Estonian Academy of Sciences. He chairs the Estonian Science Foundation.

Address questions or comments about this article to the author at Tallinn Technical University, Computer Engineering Dept., Ehitajate tee 5, EE-0026 Tallinn, Estonia; raiub@pld.ttu.ee.