VLSI Design, 2002 Vol. 14 (1), pp. 107–122

Taylor & Francis
Taylor & Francis Group

Term Trees in Application to an Effective and Efficient ATPG for AND–EXOR and AND–OR Circuits

LECH JÓŹWIAK[a], ALEKSANDER ŚLUSARCZYK[a,*] and MAREK PERKOWSKI[b,†]

[a]*Faculty of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513 EH 10.25, 5600 MB Eindhoven, The Netherlands;* [b]*Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207, USA*

(Received 20 January 2000; In final form 4 October 2000)

A compact data representation, in which the typically required operations are performed rapidly, and effective and efficient algorithms that work on these representations are the essential elements of a successful CAD tool. The objective of this paper is to present a new data representation—term trees (TTs)—and to discuss its application for an effective and efficient structural automatic test-pattern generation (ATPG). Term trees are decision diagrams similar to BDDs that are particularly suitable for structure representation of AND–OR and AND–EXOR circuits. In the paper, a flexible algorithm for minimum term-tree construction is discussed and an effective and efficient algorithm for ATPG for AND–EXOR and AND–OR circuits is proposed.

The term trees can be used for many other purposes in logic design and in other areas—for all purposes where compact representation and efficient manipulation of term sets is important.

The presented experimental results show that term trees are indeed a compact data representation allowing fast manipulations. They form a good base for algorithms considering the function's and circuit's term structures.

Keywords: Decision diagrams; Term trees; Circuit structure representation; Automatic test-pattern generation; Structural fault model

INTRODUCTION

Binary decision diagrams (BDDs) have been recognized as efficient means to model and manipulate Boolean function [1,2], and are used for design, verification and testing of digital circuits [3–6]. In particular, they are applied in functional testability analysis and functional automatic test-pattern generation (ATPG) [2,7–9].

However, exhaustive functional testing is virtually impossible. Application of the functional fault models is also questionable, because it is very difficult to guarantee a good correlation between the actual most probable physical defects and the functional faults. Therefore, structural fault models are commonly applied. They model the probable physical defects as faults of the logic level circuit structure. The aim of structural test-pattern generation then is to construct a compact set of test patterns, which are able to discover faults in a given structural fault model. The exponentially increasing complexity of digital circuits and their rapidly growing usage in various highly demanding applications has resulted in a growing demand for testing and ATPG. Here, a hundred percent fault coverage tends to be more important than a strictly minimal test set, because a lower coverage means that some faults of substantial probability remain untested, while a near-minimal test set only means that the testing time will be a bit longer. It is also important for a structural test-pattern generator to be efficient. To achieve effective and efficient ATPG, it is very important to have compact data representations in which the typical operations related to ATPG are fast, and to have effective and efficient algorithms that work on these representations.

Although BDDs can be used for compact modeling of many Boolean functions, they have too low a modeling power to represent the circuit structures accurately, and therefore cannot be directly used for the structural ATPG.

We have proposed an extension of BDDs to OR-BDDs [10,11]. This extension enables compact modeling of logic level circuit structures for AND–OR and AND–EXOR circuits and, as a result, modeling of structural faults required for structural ATPG. In further research [12], we

*Corresponding author. Tel.: +31-40-247-3645. E-mail: lech@ics.ele.tue.nl
†Tel. +1-503-725-5411. E-mail: mperkows@ee.pdx.edu

ISSN 1065-514X print/ISSN 1563-5171 online © 2002 Taylor & Francis Ltd
DOI: 10.1080/10655140290009837

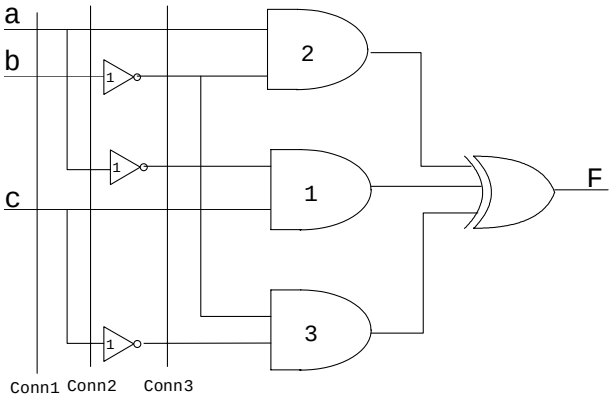

FIGURE 1 An example circuit.

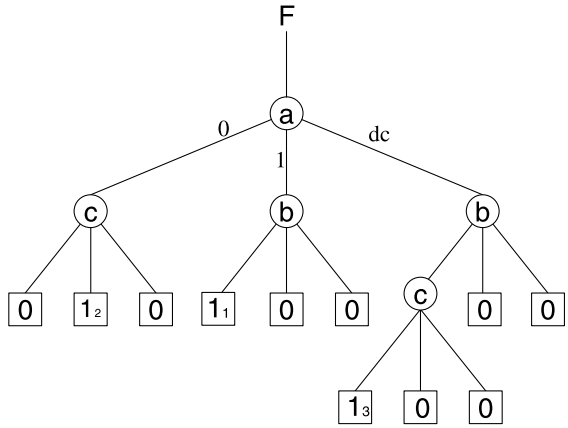

FIGURE 2 Term tree of the circuit in Fig. 1.

developed term trees, a data representation similar to OR-BDDs with the same expressive power, but more regular. Term trees are analogous to SOPTDDs and ESOPTDDs presented in Refs. [5,13]. While paths of a BDD represent disjoint terms (disjoint cubes) of a certain Boolean function, paths of a term tree represent *terms* (cubes) of a function. Since a term may cover a number of disjoint terms and each of the disjoint terms may involve more literals than the term, term trees represent many Boolean functions more compactly than BDDs. Similarly to BDDs, they can be used for modeling and manipulation of Boolean functions for various purposes. They can, however, also be used for the compact modeling of the AND–OR and AND–EXOR circuit structures, and as a consequence, for the compact representation of the structural fault models.

This paper considers application of term trees to an effective and efficient ATPG for the two-level AND–EXOR and AND–OR circuits. It focuses on the efficient construction of test patterns and minimal (or near-minimal) test sets, using the term trees as a data representation. In the scope of the research reported in the paper, we designed and implemented a term-tree-based ATPG algorithm with the following features: 100% coverage of all non-redundant faults of the fault model, a minimal (or near-minimal) test set, detection of circuit redundancy that disables testing for some faults, and a practical ATPG time and memory usage.

TERM TREES AND THEIR FAULT MODEL

Term Trees and their Correspondence to the AND–OR and AND–EXOR Circuit Structure

Term Trees (TTs), also known as "don't care"-BDDs (DCBDDs) [12,14] arose as a modification of OR-BDDs [10,11]. They have the same expressive power as OR-BDDs, but a more regular structure. While each decision node of the OR-BDD can have any number of outgoing edges, each decision node of the TT has exactly three outgoing edges. The TTs are therefore more suitable for implementation and manipulation in computer programs.

A *Term Tree* (TT) is a tree with two types of nodes: the leaf nodes and the internal nodes (decision nodes). The *leaf nodes* are labeled with the values "zero" or "one". The *internal nodes* (*decision nodes*) are labeled with the variable names. Each decision node has three outgoing branches: *zero-branch*, *one-branch* and "*don't care*"-*branch*. A path from the root to a leaf node may not contain two decision nodes labeled with the same variable name.

A collection of TTs can be used to represent a multiple-output Boolean function or a multiple-output AND–OR or AND–EXOR circuit structure as described in Ref. [14], and briefly explained below and shown in Fig. 2 for the case of a two-level AND–EXOR circuit presented in Fig. 1. In general, some TTs from a certain TT collection can share some common parts, but in the particular case considered in this paper, they are separate. For each output of the circuit, the corresponding function and the circuit structure are represented in a separate TT. The labels in the decision nodes represent the input variable names of the corresponding output function. The set of terms that is represented by a TT consists of the excitation terms of its one-leaves. The *excitation term* of a certain one-leaf can be found by traversing the path from this leaf towards the root of the TT or vice versa. The term contains literal x' (not x) for each traversed node that is labeled with variable x and succeeded by a zero-branch, literal x for each node that is labeled x and succeeded by a one-branch, and does not contain variable x for node x succeeded by a "don't care"-branch.

To determine the output value produced by the circuit for a certain input pattern, one has to trace from the root down through the tree. At each node one chooses the zero-branch (left branch) if the variable corresponding to the node has value of "zero" and the one-branch (middle branch) if the value of the variable is "one". The right branch represents a "don't-care" value of the corresponding variable. These branches are always followed for the nodes that are reached. In general, this process means more than one exit node is reached. To compute the

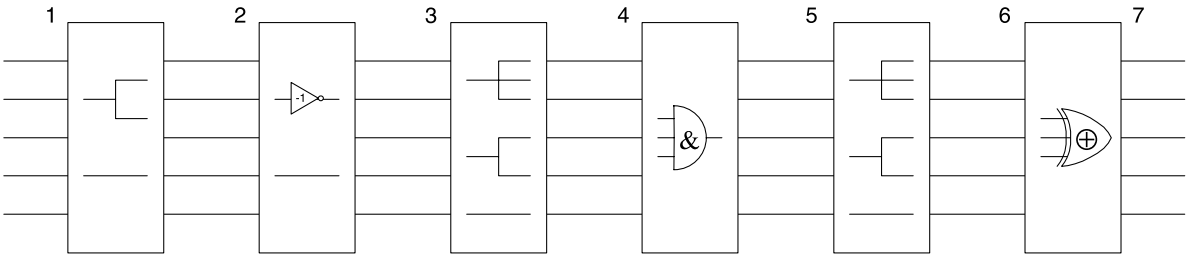

FIGURE 3 General representation of a two-level logical circuit.

circuit's output value, the values of all the exit nodes reached are EXOR'ed (in the case of an AND–EXOR circuit) or OR'ed (in the case of an AND–OR circuit). The multi-level circuits can be represented with TTs using nesting (a decision variable can represent a sub-function of a certain sub-circuit). Since TTs enable modeling of the circuit structures for AND–EXOR and AND–OR circuits, they also enable representation of the structural fault models for these circuits.

Term trees can be ordered and reduced in the same way as BDDs. However, the ordered TTs tend to be larger than the non-ordered ones. Moreover, to represent the original terms of a function, reduced TTs must preserve the paths leading to the original 1-nodes. The reduced TTs are not actually true trees, but DAGs. They can be smaller than the non-reduced TTs, but they represent the original terms of a function only implicitly, by the paths leading to the 1-node, instead of representing them explicitly by 1-nodes with different labels for different terms. This can result in extra computation effort and/or extra data structures in cases where explicit term information is required.

Fault Model

The most widely used logic-level fault model is the single stuck-at-value fault model. Therefore, and also for simplicity's sake, we will use this model to illustrate the application of TTs to ATPG. In this model, an input or output of a logic gate is considered stuck at a certain logical value. Two types of faults are distinguished: the stuck-at-zero (sa0) and the stuck-at-one (sa1). The model assumes that only a single fault can occur in a circuit at a certain time. Some faults can be eliminated from further consideration using fault collapsing. In this section, we will analyze dependencies between stuck-at faults that can occur in the different parts of a circuit, and we will derive the collapsing relations and conditions.

For this purpose, the general model of a two-level AND–EXOR circuit presented in Fig. 3 is assumed. Almost the same model and the fault collapsing for AND–OR circuits are considered in Refs. [8,14]. In the model of Fig. 3, the primary inputs (connection 1) are branched out in the first stage and fed to the second stage to produce the position or negation of an input variable. The signals are further branched out to the inputs of the AND-gates. Finally, the outputs of the AND-gates are fed to the inputs of the EXOR-gates, which compute the primary output values of the circuit.

In the AND–EXOR circuit model all single stuck-at-value faults can be modeled as faults at the interconnections between the circuit stages.

In fault collapsing, some faults can be eliminated from the model and can be discarded from further consideration. In general, stuck-at-0 (sa0) and stuck-at-1 (sa1) faults can occur for all connections 1–7. The second stage however only propagates or negates a signal. Therefore, the faults at connection 2 collapse under the faults at connection 3.

The faults that remain to be examined are sa0 and sa1 at the following locations:

- The EXOR output (connection 7—fault type EO).
- The EXOR inputs (connection 6—fault EI).
- The AND output (connection 5—fault AO).
- The AND input (connection 4—fault AI).
- The fan out points (connection 2 and 3—fault FO).
- The primary inputs (connection 1—fault I).

The detection method for each of the above fault types is given below.

EO Faults

EO faults will be tested if each output is made "0" and "1" at least once.

EI Faults

Sa0 at a particular EXOR input will be excited if "1" is offered to this particular input by the connected AND-gate. Propagation of the fault to the corresponding output is guaranteed. Because the occurrence of at most one stuck-at-value fault at a time is assumed, more than one EXOR input can be tested at a time. All EXOR inputs where "1" can be produced at the same time can be simultaneously tested for sa0 faults. For sa1 the same reasoning applies. The testing for sa0 and sa1 faults can be performed simultaneously as long as the appropriate test patterns can be produced simultaneously.

AO Faults

This fault type will be excited under the same conditions as EI fault. Propagation of the fault is always guaranteed. As long as every AND output is tested for sa0 and sa1 and the output of every EXOR is examined, every EXOR input will also be tested for sa0 and sa1, so EI fault collapses under this fault type.

AI Faults

Sa0 at a particular input of each AND-gate is excited by "1" at this particular input. For propagation, all other inputs of the considered AND-gate must be also "1". Since all inputs of the AND-gate have to be "1" at the same time, all inputs will be tested for sa0 at the same time. The excitation conditions for this fault type are the same as the excitation conditions for the corresponding AO fault (and EI fault). For this reason, sa0 EI and AO faults collapse under this fault type.

Sa1 at a particular input of each AND-gate is excited by "0" at this particular input. Propagation is only possible if all other inputs for the considered AND-gate are "1". This implies that only one input of each particular AND-gate can be tested at a time. However, more than one input can be tested simultaneously when the inputs belong to different AND-gates. When the excitation conditions for this fault type are met, the excitation conditions for sa1 at the AND output are also met. Therefore, the conditions for testing for sa1 at the EXOR input are also met. Since the fault symptoms for the EI and AO faults are always propagated, testing for fault type AI covers testing for fault types EI and AO.

FO Faults

Each function F, implemented by a single output AND–EXOR circuit, can be expressed as follows:

$$F = G_{\bar{a}}(\bar{a})\oplus G_a(a)\oplus H$$

where a is a variable occurring in F, $\bar{a}$ is the negation of the variable a, $G_{\bar{a}}(\bar{a})$ is a subfunction containing all cubes in which $\bar{a}$ occurs, $G_{\mathrm{a}}(a)$ contains all cubes in which the position of a occurs and H is the subfunction of F containing all the remaining cubes. Only two cases have to be distinguished for this fault type, namely a-sa0 and a-sa1. For $\bar{a}$-sa0 and $\bar{a}$-sa1 a similar line of reasoning applies as that of a-sa0 and a-sa1, because it concerns variables in cubes and not position or negation of variables.

For the a-sa0 fault, the faulty function F_0 becomes

$$F_0 = G_{\bar{a}}(\bar{a})\oplus G_a(0)\oplus H$$

where $G_a(0) = 0$. To detect this fault, an odd number of cubes from $G_{\mathrm{a}}(a)$ have to be selected to be evaluated to "1" for $a = 1$, so that $G_a(a) = 1$ will hold for the correct circuit. Otherwise, the fault will not be propagated. All the other cubes from $G_{\mathrm{a}}(a)$ should evaluate to "0". The cubes from $G_{\bar{a}}(\bar{a})$ and H may evaluate to any arbitrary value. This corresponds to testing all inputs of an odd number of AND-gates for sa0. Thus, to test this type of fault the same excitation conditions have to be met as those of the AI fault. The same propagation conditions as for the AI fault apply, but an additional condition states that an odd number (with respect to the variable) of AND-gates concerned must propagate to ensure the propagation of the fault symptom. Under this extra condition, the faults of type a-sa0 collapse under the AI faults.

The second fault, which can occur at connection 3 is sa1. Now, the faulty function F_1 becomes

$$F_1 = G_{\bar{a}}(\bar{a})\oplus G_a(1)\oplus H$$

To excite this fault, a has to be set to "0". For propagation, the number of cubes in $G_{\mathrm{a}}(a)$ that can evaluate to "1" if a is stuck-at-one must be odd. The cubes from $G_{\bar{a}}(\bar{a})$ and H may evaluate to any arbitrary value. The faults of type a-sa1 collapse under the AI faults, provided the number of cubes evaluating to "1" is odd.

I Faults

Assume fault a-sa0 at a certain input a. For a-sa1 a strictly analogous line of reasoning is applicable. Assignment of $a = 1$ is necessary to excite the a-sa0 fault. This means that

$$F = G_{\bar{a}}(0)\oplus G_a(1)\oplus H$$

while the faulty function F being F_2 becomes

$$F_2 = G_{\bar{a}}(1)\oplus G_a(0)\oplus H$$

Since

$$G_{\bar{a}}(0) = G_a(0) = 0.$$

fault detection is possible if

$$G_{\bar{a}}(1) \neq G_a(1)$$

is satisfied. This condition can only be met, if, for at least one output function, no redundancy for this input occurs. Take, for instance, a simple two input–output function:

$$Z_1 = ab\oplus\bar{a}b \quad \text{and} \quad Z_2 = ab$$

The fault a-sa0 cannot be detected in Z_1, because it contains redundancy for this variable. The propagation for a-sa0 has to be done via Z_2.

The condition $G_{\bar{a}}(1) \neq G_a(1)$ can be met by forcing an odd number of cubes of either $G_{\bar{a}}(\bar{a})$ or $G_{\mathrm{a}}(a)$ and an even number of cubes of the other subfunction to evaluate to "1", while the other cubes are forced to "0". H can be chosen to evaluate to an arbitrary value. The a-sa0 fault will collapse under the AI fault, under the additional condition that it will propagate through an odd number of

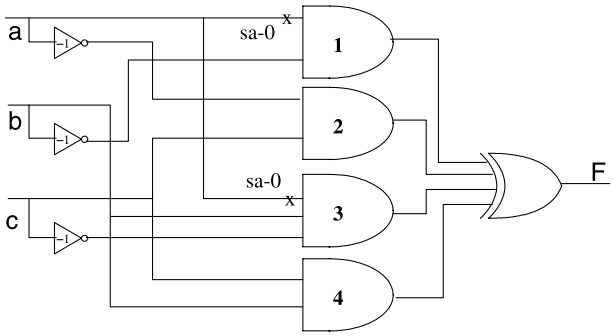

FIGURE 4 A subset of the sa0-faults in a circuit.

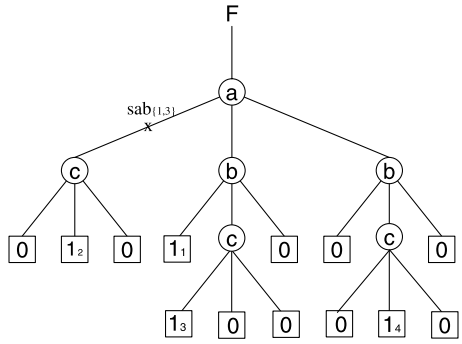

FIGURE 5 The corresponding sab faults in the term tree of the circuit in Fig. 4.

gates (i.e. if a and $\bar{a}$ is replaced with "1" in the considered cubes).

The sa1 fault will collapse under the AI fault, under the same additional condition concerning the number of cubes.

In sum: the faults of type EI and AO need not be considered during ATPG, because they collapse under other faults. The faults of type FO and I collapse under type AI, under the above-mentioned extra conditions. In our ATPG method, these fault types are not considered initially. At a later stage, a check is made to see if the extra conditions for the fault collapsing are actually met. This is explained in "Generation and reduction of the cover matrix" section.

This Leaves Us the Following Fault Types

Sa0 and sa1 at the EXOR outputs (EO), and sa0 and sa1 at the AND inputs (AI). In practice, the EO faults are tested automatically during the testing of the AI faults, because each EXOR output has to switch to logical "zero" and to logical "one" only once to test for the EO fault. However, it is not absolutely certain that this always will be the case. In the "Generation and reduction of the cover matrix" section, we will show how testing of all EO faults is guaranteed.

Similarly, as shown in Refs [11,14], after fault collapsing, only the following two types of faults remain to be tested for AND–OR circuits:

- Sa0 faults at the OR inputs.
- Sa1 faults at the AND inputs.

For each of the above types of faults, an analogy in the TT can be found.

Fault Model Representation With TTs

As discussed in "Term trees and their correspondence to the AND–OR and AND–EXOR circuit structure" section, direct correspondence between the circuit structures of the AND–OR and AND–EXOR circuits and TTs enables representation of the structural fault models with the TTs. To represent the circuit's stuck-at-value faults, the concept of a stuck-at-branch (sab) fault is introduced in the term tree. A sab fault is defined in relation to a particular subset of the TT's 1-nodes (subset of the function's cubes), and means that a side-branch will be taken on the path to the subset of the TT's 1-nodes instead of selecting the correct branch and following the correct path. This will lead to erroneous leaf-nodes, and may result in a faulty output value.

A sa0 fault at a certain input of a particular AND-gate of a given circuit is mapped to a sab fault in the corresponding term tree. The sa0 fault will manifest itself at a side branch of the path that corresponds to the cube associated with the AND-gate to which the input belongs. The particular branch where the sab fault appears is the branch stemming from the node matching the variable that determines the value of the input where the sa0 fault occurs. The sab fault appears at the zero-branch if the input is determined by the position of the variable, and it appears in the one-branch if the input is determined by the negation of the variable

Figure 4 shows some of the sa0 faults at the inputs of AND-gates. In the case of the sa0 fault at the input of the first gate, the output of the AND-gate will constantly produce "zero", no matter what patterns are presented to the inputs of the circuit, since the value on one of the inputs is always "zero". When considering the first cube, it is obvious that the influence of variable a is nil. The first cube constantly sees value of "zero" for variable a. Therefore, in the TT it looks like variable a constantly has value "zero" for this particular cube. This can be modeled

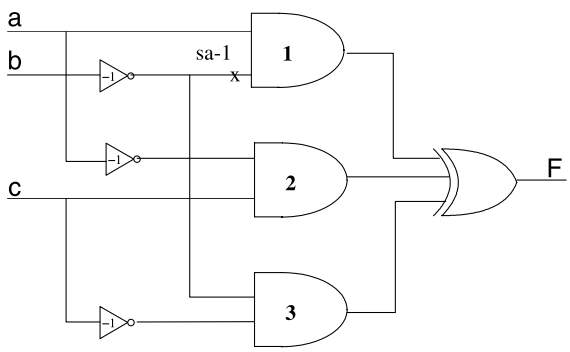

FIGURE 6 A sa-1 fault in a circuit.

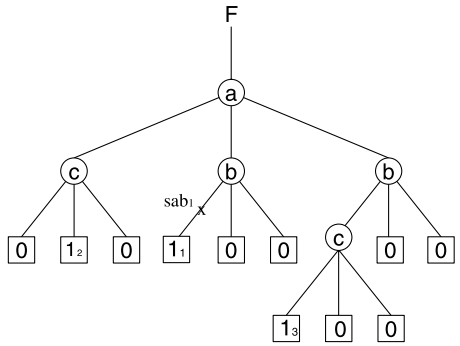

FIGURE 7 The corresponding sab fault in the term tree of the circuit in Fig. 6.

by a stuck-at-branch fault in the zero-branch of the node matching variable *a* in the path representing the first cube. A similar reasoning applies to the sa-0 fault at the input of the third gate. Since both the considered faults appear at the AND-gates' inputs (determined by position of variable *a*), they both can be modeled by the same sab fault. This is shown in Fig. 5. An index of the sab fault represents the cubes for which the fault manifests itself.

Similarly, the occurrence of a sa1 fault at a certain input of a particular AND-gate (Fig. 6) introduces a corresponding stuck-at-branch fault in the term tree. However, in contrast to the sa0 fault, which forces the circuit to leave the path leading to a one-exit node, the sa1 fault forces the circuit to stay on the path (at the node of the variable where the stuck-at-value fault occurs). This is modeled by a sab fault in one-branch (if the position of the variable is fed to the input of the AND-gate) or zero-branch (in the case of negation of the variable). The term tree with the corresponding sab fault is shown in Fig. 7. For the AND–OR circuits, the sa0 faults at the OR inputs must be modeled in their corresponding TTs too. A sa0 fault at the OR-input corresponds to a sa0 fault at the appropriate 1-leaf node in the TT [11,14].

For simplicity's sake, we only considered the stuck-at-value fault model. However, other structural fault models can be introduced for TTs in an analogous way, because there is a direct correspondence between the AND–OR or AND–EXOR circuit structure and the corresponding TT.

The direct correspondence between the circuit structure and the diagram is partially lost in the case of reduced TTs. The original terms (AND-gates) of a circuit are no longer explicitly represented in the diagram by different 1-nodes for different ANDs, but only implicitly by the paths leading to the only 1-node of the corresponding reduced TT. Therefore, the faults at the ANDs' outputs cannot be explicitly represented in the reduced TTs. This way the uniform fault representation is lost, because an additional data structure is required to explicitly represent these faults. In this paper, only the non-reduced and non-ordered TTs will be considered.

Usage of the TT representation for ATPG instead of the netlist representation has several advantages. First, the size of a TT (the number of its decision- and 1-nodes) usually is smaller than the size of the corresponding netlist modeled by the TT (the number of input nets of all its gates). The number of the TT's 1-nodes is equal to the number of the EXOR-gate (or OR-gate) inputs in the corresponding netlist. The lower bound of the number of the TT's decision nodes is equal to the number of the circuit's primary input variables, and the upper bound is equal to the number of inputs to all AND-gates. In the TT, this upper bound is only reached in the case that there are no common literals for any two terms of the function. Compactness of the TT representation results in reduction of the number of faults to be covered, because some subsets of faults in the netlist are represented as single faults in the TT. In Ref. [15], we showed for a number of circuits that the reduction factor was between 30 and 56%. This fact together with easy computation of excitation and propagation conditions in TTs result in a shorter test-pattern generation time. Another advantage is quick detection of circuit redundancy [11].

TERM-TREE CONSTRUCTION METHOD

A certain circuit structure can be represented by many various term trees, with different numbers of nodes [11,14]. In order to have a compact structure representation and to speed up the ATPG algorithms, a term tree with a small number of nodes should be used. This results in the following minimization problem: *given a set of terms, find a term tree that represents this set of terms and has the minimal number of decision nodes*. For the non-ordered TTs, this can be achieved by a different ordering of the variables in each sub-tree and may result in smaller non-ordered TTs than the minimum ordered TTs.

As some subsets of terms generally can have common literals, the TT's paths representing the terms of a certain subset can share some common sub-paths. The upper bound of the TT's complexity therefore is the complexity of the netlist represented by the tree. This upper bound is only reached in the case that terms of the modeled netlist have no common input literals. In most practical cases, the netlist is much more complex than its TT representation. Since a certain term may cover a number (in some cases a large number) of smaller disjoint terms (i.e. involving more literals), TTs represent many Boolean functions more compactly than BDDs [11,14].

The algorithm used to solve the term-tree minimization problem is based on the AND/OR graph search methods of artificial intelligence [16]. Usually, such algorithms are described by means of a search graph and a separate heuristic control mechanism that expands the graph. A description in the form of a hierarchical system of concurrent processes, however, is more elegant and simplifies implementation with an object-oriented programming language, as each process can be directly implemented as an object. A node of a search graph now corresponds with a process that communicates with its parent process and its child processes. Each process has

TABLE I Summary of term tree construction results for 20 AND–OR benchmark circuits

Name	#i	#o	#t	$w = 0$	$w = 0.5$	$w = 1$
9sym	9	1	86	241/0	219/1	208/1034
b12	15	9	43	109/1	104/1	101/2
Bw	5	28	22	428/0	391/1	376/2
Con1	7	2	9	18/0	18/0	17/0
duke2	22	29	86	1102/4	774/3	-/-
ex1010	10	10	284	2659/8	2385/22	-/-
ex5p	8	63	74	2609/9	2434/9	-/-
inc	7	9	30	141/1	134/0	130/0
misex1	8	7	12	95/0	93/0	90/1
misex2	25	18	28	166/0	160/0	160/1
misex3c	14	14	197	853/4	722/3	-/-
rd53	5	3	31	68/0	68/1	68/0
rd73	7	3	127	301/1	301/34	301/365
rd84	8	4	255	606/1	606/69	-/-
sao2	10	4	58	235/1	183/0	-/-
sqrt8	8	4	38	73/0	73/0	71/2
squar5	5	8	25	67/0	65/0	63/0
Table3	14	14	175	3363/15	2491/13	-/-
Table5	17	15	158	3124/20	2522/16	-/-
xor5	5	1	16	31/0	31/0	31/0

The columns #i, #o and #t denote the number of inputs, outputs and terms, respectively. In each entry *x*/*y* from the last three columns, *x* is the total number of decision nodes in the term trees, and *y* is the number of seconds needed to compute these term trees. An entry -/- indicates that the maximal memory resources (about 200 Mb) were not sufficient to complete the construction algorithm.

the task of solving a sub-problem. The communication between a parent and its child only occurs when one party wants to receive a message and the other party wants to send a message.

We distinguish the three following types of processes:

- *Root process*: it receives a term set T from the environment, and returns a minimal term tree representing T to the environment.
- *Support process*: it solves the problem of constructing a minimal term tree representing a given term set T.
- *Term process*: it solves the problem of constructing a minimal term tree representing a given term set T and having a root labeled with a given variable x from the support of T.

Each process can solve its problem by creating child processes and delegating tasks to them. Most processes start with sending a cost estimate of their solution to their parent. The parent then interprets this information and chooses whether or not the child may take the next step towards solving its problem. If the next step may be taken, the parent process sends an activation message to its child and waits for a message from the child. If the child requires more steps to come to a solution, it sends a new cost estimate and waits for the next activation message. Otherwise, it sends the solution to its parent. When a process has returned its solution to its parent, it automatically destroys itself and all of its descendants.

A process that delegates tasks to the child processes decomposes its problem into sub-problems that have to be solved by the child processes. This decomposition can be performed in two dimensions: decomposition of the problem (used by the term processes) or decomposition of the solution space (used by the support processes). In the first case, each child solves a sub-problem and the results of all children are combined into a solution to the main problem. In the second case, all children search different parts of the solution space of the main problem and the solution of one of the children is chosen as the solution to the main problem. The total system uses distributed hierarchical control: each process is controlled by its parent and controls its own children. The heuristic rules that control the system are incorporated in the processes. Processes contributing to partial solutions that seem to have a high chance of becoming optimal are given the chance to take action and initiate child processes assisting them in their task, while the processes that seem to be worse remain in a waiting state. Thus, the ability of a process to perform actions and to create child processes depends on the choices made by its superiors in the hierarchy.

The term-tree construction method as outlined above, has been implemented in C++ on a sequential computer, but its implementation on a parallel computer would be easy, because the method consists of a hierarchical system of concurrent processes. The algorithm was tested on a Pentium 133 MHz personal computer running Windows95 for circuits from the IWLS'93 benchmark suite [17]. We used ESPRESSO [18] to synthesize the two-level logic circuits, and then constructed the term-tree representations of the resulting circuits by using the method described above. Table I shows the summary of the results for a number of benchmark circuits.

Note that we used our term-tree construction algorithm for different weight factors, ranging from $w = 0$ (pessimistic heuristics, fast search, and sub-optimal results) to $w = 1$ (optimistic heuristics, slow search, and strictly optimal results). The results clearly show the flexibility, efficiency and effectiveness of our algorithm: we can trade off the run-time of the algorithm against the solution quality, we can handle large term sets in less than a second and the algorithm constructs the optimal or near-optimal TTs.

ATPG ALGORITHM

We developed a term-tree-based ATPG algorithm with the following features:

- 100% coverage of all non-redundant faults of the fault model.
- A (near) minimal test set.
- Detection of circuit redundancy that disables testing for some faults.
- A practical test-pattern generation time and memory usage.

The successive steps of the ATPG algorithm are presented and explained below:

1. Read the circuit specification.

2. Construct the minimal (or near-minimal) term trees.
3. Compute the excitation patterns from the term trees.
4. Compute the test patterns from the excitation patterns.
5. Construct the fault cover matrix.
6. Solve the covering problem.
7. Output the results.

The near-minimal term trees are constructed from the original circuit specification as described in "Term trees and their fault model" section.

The excitation patterns of particular one- and zero-leaves of the constructed term trees are easily determined for each tree by walking from its leaves to its root.

The sa0 AI faults appear as the sab faults at the side branches of the TT's paths leading to particular one-exit nodes. (Note that the sab faults do not appear in the *don't care* branches). This means that the occurrence of such a sa0 fault diverts the circuit from the TT's path representing the function's product term where this fault occurs. To excite and propagate a particular sa0 (sab) fault, the path to the corresponding one-exit node has to be activated. The correct circuit reaches the one-exit node. Any sab fault anywhere along the activated path prevents the circuit from reaching the one-exit node. The sab faults at the side-branches along the path leading to a certain one-exit node correspond to the sa0 faults at the inputs of the product term, which is represented by the one-exit node. Each test pattern that reaches the one-exit node tests simultaneously all sab (sa0) faults of the particular product term. Thus, the sa0 faults at all inputs of one particular AND-gate are tested simultaneously. No other test patterns than the ones activating the paths leading to the TT's one-exit nodes cover the sab (sa0) faults, and therefore this set of patterns is complete as far as the sa0 AI faults are concerned. This means that the test patterns for the sa0 AI faults are the excitation patterns for the TT's one-nodes. These patterns can be derived directly and easily from the term tree, as described in "Term trees and their fault model" section. The number of the test patterns for the sa0 AI faults equals the number of the AND-gates in the circuit. (Note that the patterns contain *don't care* values for the variables that do not appear in the considered product term).

The sa1 AI faults appear as the sab faults at the zero- and one-branches in the paths leading to the corresponding one-exit nodes. To excite a particular sa1 (sab) fault "zero" must be offered to the input that is tested. To ensure propagation of the fault symptom, all other inputs connected to the considered AND-gate have to be "one". All patterns satisfying these conditions are the patterns with Hamming-distance-one from the excitation pattern of the corresponding one-exit node. All these patterns correspond to certain side branches of the path leading to a certain one-exit node. If a sa1 fault occurs, the circuit will not reach the zero-node through a certain side branch, but will reach the one-exit node instead. The number of test patterns for the sa1 faults equals the sum over all product terms (AND-gates) of all variables that determine the value of the term.

For the AND–OR circuits, the sa0 faults at the AND outputs (OR inputs) must also be tested. They correspond to the stuck-at-zero faults of the TT's one-leaves.

The test patterns for the stuck-at-zero faults of the one-leaves are computed from the one-leaves' excitation patterns by a procedure that ensures a Hamming distance of at least one between all test patterns of different one-leaves of a certain term tree. If such Hamming distance is guaranteed, then no test pattern of a certain one-leaf node can excite another one-leaf node of the term tree considered. This guarantees the propagation of the fault symptom to the circuit output corresponding to the term tree.

At this point, the test patterns are found for all faults that remain after fault collapsing. We also have explicit information about which patterns cover a certain fault, implicitly describing which faults are covered by a certain pattern. From this, we compute explicit information about which faults are covered by a certain pattern and we construct a cover matrix, by using a recursive procedure with a minimized computation effort. This procedure also "combines" the patterns that include sub-patterns covering multiple faults, through extraction of such sub-patterns from the original patterns. In the case of the AND–EXOR circuits, the matrix is expanded by adding some extra rows to ensure that the listed patterns meet the requirement to collapse the FO and I-faults (see "Term trees and their fault model" section).

For larger problem instances, we use some heuristics to limit the number of test patterns in the resulting cover matrix. These heuristics use the concepts of preprocessing and exclude the least promising patterns from further consideration. Their decisions follow from the selection criteria based on the following:

- Potentiality of the test patterns.
- Essentiality of the test patterns.
- Effectiveness of the test patterns.
- Required ATPG time.

The *potentiality* is determined by a number of "don't cares" in the pattern, and it is the measure of the number of extra faults the pattern may cover after assigning its "don't cares". The more "don't cares", the more freedom the pattern has to combine itself with some other patterns that cover some extra faults. Thus, the more likely it is that this pattern can detect more faults after an appropriate "don't care" assignment.

The *essentiality* is a measure of the likelihood that the pattern will be an element of the minimal test set. It is determined by the number of patterns that cover any fault covered by this pattern. If that number is one, then the pattern is essential, i.e. it is sure to be in the minimal test set and cannot be deleted because otherwise the main objective (100% fault coverage) will not be guaranteed. If the number is two or greater, then the pattern can be

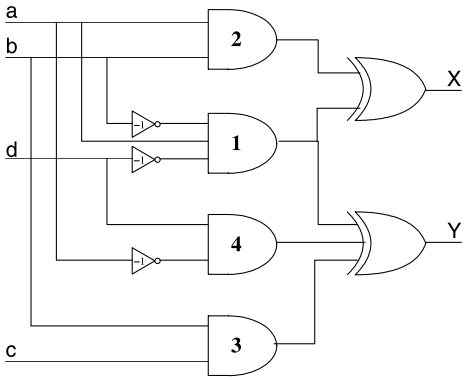

FIGURE 8 An example circuit.

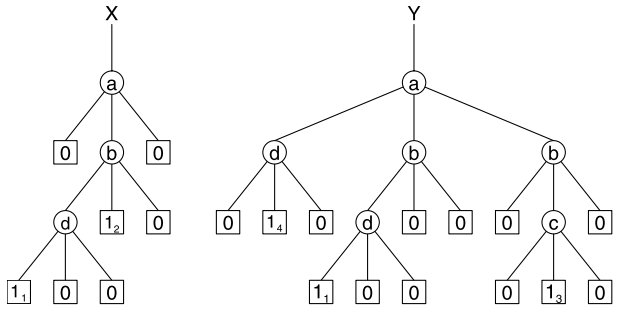

FIGURE 9 The term trees of the circuit in Fig. 8.

deleted, but only the patterns with the lowest essentiality are actually deleted.

The *effectiveness* of patterns is measured by considering how many faults are covered by a certain pattern, and how difficult these faults are to cover. The lower the effectiveness, the sooner the pattern will be discarded.

The *ATPG time* can be influenced by loosening or tightening the pattern-selection criteria based on the previous three parameters. If the criteria for deleting patterns are very loose, then the resulting cover table will be small, and the algorithm will be fast. Strict criteria result in a larger cover table and a longer ATPG time. In this way, a strongly non-linear trade-off is realized between the quality of the resulting test set and the ATPG time. Consequently, the ATPG time can be shortened considerably without any substantial influence on the quality of the test set.

The cover matrix is further reduced by deleting all dominated patterns (i.e. patterns that cover less faults than some other ones), and all but one of the equivalent patterns (i.e. patterns that cover precisely the same faults). The essential patterns (i.e. such patterns that each of them covers some faults that are being covered exclusively by this pattern) are directly added to the test set and removed from the cover matrix, along with all the faults they cover. These preprocessing methods are alternately and recursively applied until no more reductions are possible.

Now the hard core of the coverage problem remains to be solved. We apply a beam-search algorithm to solve the reduced cover problem. The beam-search is a variation of the breadth-first search where only a limited set of the most promising alternatives is explored in parallel. For each step it uses some heuristic rules and the preprocessing described above to prevent the search tree from expanding too much [19]. It first considers the near-essential patterns (i.e. patterns that cover faults that are being covered by only a few patterns) in the order of their essentiality (patterns that cover faults covered by less patterns have priority). It is therefore capable of preserving among its sub-solutions the strictly minimal character of the constructed test set until the sub-solutions become more advanced, while preventing the search tree from expanding too much. This capability ensures that at least some of the constructed sub-solutions are subsets of the minimal test sets, until the construction process becomes more advanced. If there are no more near-essential patterns, then the choices of the beam algorithm are based on the potentiality and effectiveness of the patterns. In the first search phase, all constructed sub-solutions are preserved until a certain limit is reached. Hopefully, at the time when some sub-solutions have to be discarded, the potential of the sub-solutions to lead to the optimal (or near-optimal) solution will be estimated well enough. This estimation is based on the information included in the already constructed sub-solutions and in the reduced matrix, which is still remaining.

To further improve the effectiveness and efficiency of the ATPG, the algorithm described above can be executed in a parallel-processing environment. All known parallelization methods [20,21,33] can be used in relation to our ATPG algorithm, including fault partitioning [21,22], search-space partitioning [23], algorithmic partitioning [23], and topological partitioning [24]. Moreover, the fault partitioning and the search-space partitioning do not require any substantial changes to the algorithm itself, because it works the same way on a fault sub-list as on the original fault list and its beam search actually decomposes the search space into sub-spaces that can be searched in parallel.

EXAMPLE

To illustrate and better explain the ATPG algorithm, its test-pattern generation process will be discussed for an example circuit in this section. We will demonstrate the following:

How the patterns that meet the excitation and propagation conditions are derived.

How the cover matrix is built.

How the cover matrix is reduced by removing the dominated patterns and equivalent patterns and by selecting the essential patterns.

TABLE II Sa0 excitations

(Group of) Fault(s)	Pattern
1X1	10-0
1X2	11–
1Y1	10-0
1Y3	-11-
1Y4	0–1

How the algorithm builds the solution tree.

How the complete solution is found, which covers all non-redundant faults of the fault model.

Finding Excitation and Propagation Patterns

In Fig. 8, an example circuit is given. The corresponding term trees are shown in Fig. 9. From these term trees the test patterns are derived for all faults from the fault model that remained after fault collapsing ("Fault model" section). Observe that term 1 has a one-exit node in the term trees for both outputs *X* and *Y*. This term therefore is observable through both outputs. In the path to this one-exit node, the variables *a*, *b* and *d* appear. The path is formed by a one-branch, a zero-branch, and another zero-branch of the nodes, respectively, corresponding to these variables. Therefore, the pattern 10-0 excites cube 1. This pattern covers all sa0 faults at term 1. In a strictly analogous way, the excitation patterns for sa0 faults for all other terms of the circuit can be found. In Table II, the patterns that cover all sa0 faults of the circuit are listed.

The following identifier convention is used in Table II:

The first identifier indicates the excitation type: 1 for one-node excitation and 2 for zero-node (recall that a particular one-node excitation covers all the sa0 faults at the inputs of the corresponding AND-gate, while a zero-node excitation is only capable of testing a single sa1 fault for the corresponding AND-gate).

The meaning of the second identifier depends on the value of the first identifier: for 1 it is the name of the output variable, at which the fault can be observed; for 2 it is the name of the input variable, at which the fault occurs.

The third identifier indicates the term number (AND-gate number) of the cube (gate), at which the error occurs.

The patterns for the sa1 fault testing can be derived from the patterns for the sa0 fault testing. For instance, to detect a sa1 fault in variable *a* at term 1 we need a pattern that will excite term 1 (the value of *a* is the opposite of its value in the term's 1 excitation pattern). In Table II, we can find that the pattern 10-0 will excite term 1. By inverting the value for the variable *a* we obtain a pattern that detects the sa1 fault in *a* at term 1. This is also the weakest condition for testing of this fault. By doing this for every cube and every variable in each cube, we obtain all patterns covering all sa1 faults. These patterns are listed in Table III.

TABLE III Sa1 excitations

Fault	One-exit node to be tested	Test pattern
2a1	10–0	00–0
2a2	$\overline{1}$1–	01–
2a4	$\overline{0}$–1	1–1
2b1	$\overline{1}$0-0	11–0
2b2	1$\overline{1}$–	10–
2b3	–$\overline{1}$1–	–01–
2c3	–$\overline{1}$1–	–10–
2d1	10–0	10–1
2d4	0–$\underline{\overline{1}}$	0–0

Generation and Reduction of the Cover Matrix

Information on test patterns and faults covered by them is used to construct the initial cover matrix (Tables II and III). The cover matrix for the example circuit is presented in Table IV. After the cover matrix is constructed, its reduction process starts. We have to remove the equivalent and dominated patterns, and store the essential patterns from the cover matrix. In Table IV, for instance, pattern 0000 is dominated by pattern 0010. For type AI (AND input sa-faults) testing domination is unconditionally true, but FO (fan-out point) and I (input) faults only collapse under certain additional conditions. These additional conditions state that a fault of type FO or I collapses only if the variable (either position or negation of a variable) that is tested changes the output of an odd number of cubes seen from the perspective of one output. Therefore, we need to determine what patterns meet the conditions to collapse faults FO and I. With the guidelines from "Term trees and their fault model" section, we can determine if a certain FO or I fault is detected by a pattern that detects the AI fault. This results in the cover matrix from Table V.

TABLE IV Cover matrix

	0000000011111111
	0000111100001111
	0011001100110011
	0101010101010101
1X1	..... X . X .....
1X2	..... X X X X
1Y1	..... X . X .....
1Y3	..... X X ..... X X
1Y4	. X . X . X . X .....
2a1	X . X .....
2a2	..... X X X X .....
2a4	..... X . X . X . X
2b1	..... X . X
2b2	..... X X X X .....
2b3	..... X X ..... X X .....
2c3	..... X X ..... X X .....
2d1	..... X . X .....
2d4	X . X . X . .....

TABLE V Cover matrix with type FO and I coverage

	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1X1	X X
1X2	X X X X
1Y1	X X
1Y3	X X X X
1Y4	X X X X
2a1	X X
2a2	X X X X
2a4	X X X X
2b1	X X
2b2	X X X X
2b3	X X X X
2c3	X X X X
2d1	X X
2d4	X X X X
3a1	X X X X X
3a2	X X X X X
3b1	X X X X X
3b2	X X X X X
3c1	X X X X
3c2	X X X X
3d1	X X X X
3d2	X X X X
4a1	X X X X
4a2	X X X X
4b1	X X
4b2	X X
4d1	X X
4d2	X X
5a1	X X X X X X X X
5a2	X X X X X X X X
5b1	X X X X X
5b2	X X X X X
5c1	X X X X
5c2	X X X X
5d1	X X X X X X
5d2	X X X X X X
6X	X X X X
6Y	X X X X X
7X	X X X X X X X X
7Y	X X X X X X X

The faults of type EO are also added to the matrix. The newly added faults are assigned new identifiers. If the first identifier has value "3", we deal with a type FO fault in the position of the corresponding variable. Value "4" denotes a FO fault in the negation of the corresponding variable and value "5" denotes an I fault. An sa0 EO fault is identified by "6" and sa1 EO fault by "7". The second identifier gives the name of the corresponding variable.

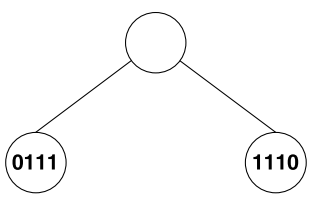

FIGURE 10 The first level of the solution tree.

TABLE VI Cover matrix after reduction

	0 0 0 0 0 1 1 1 1 1 1
	0 1 1 1 1 0 0 1 1 1 1
	1 0 0 1 1 0 1 0 0 1 1
	1 0 1 0 1 0 0 0 1 0 1
1X1	X X
1X2	X X X X
1Y1	X X
1Y3	X X X X
1Y4	X X X
2a2	X X X X
2b1	X X
2c3	X X X X
3a2	X X X X X X X
3b2	X X X X X X
3c1	X X X X
3c2	X X X X
3d2	X X X
4a2	X X X
4b1	X X
4b2	X X
4d2	X X
5b2	X X X X X
5c1	X X X X
5c2	X X X X
6X	X X X X X X
6Y	X X X X X X

For the FO faults, the third identifier denotes the sa1 fault (1) or sa0 (2).

An example of a pattern that does not detect a particular I-type fault in spite of detecting the AI faults is 1010. It does not detect faults of type I in variable *b* (fault 5b), because as seen through both outputs, two cubes always propagate a change in variable *b*. The same applies to pattern 1110.

Now the coverage of the EO, FO and I faults is also known and we can safely reduce the cover matrix. It is obvious that the following patterns are dominated patterns: 0000 (dominated by 0010), 0001 (dominated by 0011, and 0101), and 1001 (dominated by 1011). These patterns can be removed from the cover matrix. Equivalent patterns are not present in this particular matrix.

The cover matrix can be further reduced by placing the essential patterns in the table containing the test patterns and removing all faults covered by these patterns from the cover matrix. In this example, the essential patterns are 0010 (covers fault 2a1 as the only pattern) and 1011 (covers fault 2d1 as the only pattern). These patterns are stored in the table containing the test patterns. All faults that are covered by these two patterns are removed from the cover matrix. The resulting cover matrix is shown in Table VI.

This process of the alternate removal of the equivalent and dominated patterns and transfer of the essential patterns from the cover matrix to the table of test patterns is performed repeatedly until no such patterns can be found anymore. For our example circuit, the cover matrix from Table VII and the table of test patterns as shown in Table VIII results from this process.

Since the cover matrix form Table VII cannot be reduced further by preprocessing techniques, the beam-search

TABLE VII Cover matrix after the third reduction

	0	0	1	1	1
	1	1	1	1	1
	0	1	0	1	1
	1	1	0	0	1
1X2			X	X	X
1Y3		X		X	X
1Y4	X	X			
2a2	X	X			
2b1			X	X	
2c3	X		X		
3b2		X	X	X	X
3c1	X		X		
3c2		X		X	X
3d2	X	X			
4a2	X	X			
4b1			X	X	
5b2		X	X		X
5c1	X		X		
5c2		X		X	X

TABLE VIII Test patterns

Pattern
0010
1011
1000

algorithm is used to decide which of the test patterns remaining in the reduced cover matrix have to be selected and stored in the table of test patterns.

The Hard Core of the Cover Problem

The First Level One Sub-solutions

In Table VII, we can see that the excitations 1Y4, 2a2, 2b1, 2c3, 3c1, 3d2, 4a2, 4b1, and 5c1 are all covered by two patterns. Among the patterns covering these faults (all the patterns except 1111), pattern 0111 covers the most faults (9) and therefore is selected as a part of the sub-solution. Assuming that the beam width (number of sub-solutions considered at each level) is two, we select yet another pattern of high efficiency—1100. Each of the two patterns forms a sub-solution of the hard-core cover problem. In

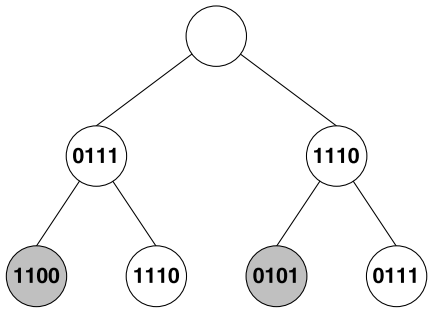

FIGURE 11 The solution tree.

TABLE IX Cover matrix for the sub-solution 1

	0	1	1	1
	1	1	1	1
	0	0	1	1
	1	0	0	1
1X2		X	X	X
2b1		X	X	
2c3	X	X		
3c1	X	X		
4b1		X	X	
5c1	X	X		

Fig. 10, the first level of the solution tree is shown. Since in the first sub-solution pattern 0111 is selected as a test pattern, it is removed from the cover matrix together with all faults covered by it. The resulting cover matrix is shown in Table IX. In the second sub-solution, pattern 1100 is chosen as the test pattern. This pattern is removed from the cover matrix together with all faults covered by it. The resulting cover matrix is shown in Table X. For both sub-solutions *solve covering problem* routine is called again to find the next level in the solution tree.

The Second Level Sub-solutions

First, the algorithm is executed again for the cover matrix from Table IX. Again two patterns can be chosen as potential test patterns, but this time the algorithm discovers that the most efficient pattern 1100 covers all faults of the cover matrix in Table IX. Thus, the first complete solution is found.

The algorithm is also executed again for the cover matrix from Table X. Again, two patterns would be chosen as potential test patterns, however this time the algorithm discovers that the most efficient pattern 0111 covers all faults in the cover matrix in Table X. Thus, another complete solution (see Fig. 11) has been found.

The Final Solution

After finding the set of complete solutions, the algorithm stops. It should be noted that both complete solutions constructed at level 2 represent the same test pattern set: {0111,1100}. There, therefore, is no need to make any additional choices, which, in the case of different solutions, would be made arbitrarily. The complete set

TABLE X Cover matrix for the sub-solution 2

	0	0	1	1
	1	1	1	1
	0	1	1	1
	1	1	0	1
1Y3		X	X	X
1Y4	X	X		
2a2	X	X		
3c2		X	X	X
3d2	X	X		
4a2	X	X		
5c2		X	X	X

TABLE XI The final solution

Patterns
0010
0111
1000
1011
1100

of the test patterns representing the final solution is given in Table XI.

ATPG RESULTS

The implemented ATPG program has all required features. It guarantees 100% coverage of all non-redundant single stuck-at faults, because it constructs and keeps the test patterns for all faults from the fault model and it works until all faults are covered. The results of the ATPG program of more than 30 circuits, mostly from the IWLS'93 logic synthesis benchmark set [17], were simulated to check whether the software implementation is correct. The tested circuits were minimized using ESPRESSO [18] for AND–OR implementations and EXORCISM [25] for AND–EXOR implementation. All tested circuits, but one indeed yielded 100% coverage. The only exception was the AND–OR circuit implementation of a05 that has redundancy. For this circuit, the ATPG program covered all non-redundant faults and reported the redundancy.

To check if the second objective, a minimal (or near-minimal) test set, was accomplished, the solutions from the ATPG program were compared for a number of circuits to the exactly minimal solutions computed by an implicit exhaustive algorithm. In most cases the ATPG program found the strictly optimal solutions, and in the other cases the best near-optimal solutions, i.e. the ones containing only one test pattern more than minimum.

The last and less important aspect is the ATPG time and memory usage. The cost of CPU time for test generation is negligible compared to the costs of the actual test application or the costs that result from approving a faulty product. The ATPG time and memory usage should, however, be practical. Our algorithm controls the usage of these two resources and it is able to limit their usage to be practical without an excessive increase of the test set. The ATPG time for small circuits was in the order of seconds and in the order of hours for large ones on a slow computer of only 2 MIPS. This is certainly a practical time. It can, however, be at least 10 times shorter on a fast PC or workstation, and it can further be much reduced by execution of the ATPG program in a parallel processing environment or by narrowing the search.

The comparison with ATPG results of SIS system [26] was also performed. However, in the case of the AND–EXOR circuits a problem of representation of large EXOR-gates was encountered for SIS. In BLIF format accepted by SIS, the representation of an EXOR-gate with more than 12 inputs leads to input files that cannot be handled by SIS in a practical space of time. Therefore, the output EXOR-gates were replaced by the two-level EXOR networks, with the 8-input EXOR-gates at the first level and an EXOR-gate of the appropriate size at the second level. This solution obviously introduces few new points (the inputs of the second level EXOR-gate), which are tested by SIS for stuck-at faults. This fact makes the comparison more difficult. To make the comparison as objective as possible, we present two numbers in the case of the problematic benchmarks—lower and upper bound of the SIS pattern set cardinality. The upper bound is the number of patterns actually generated by SIS—including the patterns testing the additional points between the EXOR levels. The lower bound is calculated using an assumption that all the additional points are tested by a

TABLE XIII The number of optimal and near optimal solutions for some circuits

Circuit	Optimal	Optimal + 1	Optimal + 2	Total
Sse	432	5600	40,000	3,000,000
ex4	320	6700	62,000	44,000,000
mark1	1	19	55	179

TABLE XII Results of the ATPG program

			OR						XOR					
Circuit	#i	#o	#t	#n	#p	t(s)	SISp	SISt	#t	#n	#p	t(s)	SISp	SISt
a04	9	8	74	1061	99	9	134	2	80	1586	173	26	161–194**	40
a05	7	6	18	231	25*	2	33	<1	20	381	52	5	54	16
clip	9	5	117	956	148	5	173	5	63	731	109	15	113–128**	12
dk16	7	8	67	932	91	4	102	2	62	896	77	12	68–87**	4
ex4	10	13	21	433	37	2	43	1	19	436	34	7	35	2
mark1	9	18	24	546	43	3	45	1	21	537	41	12	45	1:02
opus	9	10	21	487	55	3	59	1	18	424	47	5	49	5
rd53	5	3	31	207	32	1	32	<1	14	111	15	2	17	<1
rd84	8	4	256	1822	256	14	256	1:20	63	514	62	14	71–80**	3
squar5	5	8	25	191	20	2	21	<1	18	164	15	3	16	<1
sse	11	11	34	587	74	9	81	1	27	524	60	12	62–68**	4

#i, #o, #t, #n and #p, respectively denote the number of inputs, outputs, terms, term tree nodes, and test patterns. SISp and SISt denote number of patterns and time of SIS atpg function. (*) indicates redundancy (**) lower-upper bound of the number of patterns

pattern that was introduced exclusively for this point. In most cases this is superfluous, as the additional points (EXOR-gate inputs) are "easily" testable, and often are tested by the patterns generated for other points.

As one can see from Table XII, our program outperforms SIS (as far as the number of patterns is concerned) in all cases where the comparison is made between exactly the same circuits. The only exceptions are circuits rd53 and rd84 in AND–OR implementation, which, due to their structure, require the full set of input vectors to be applied as the test set. The test sets for rd53 and rd84 are the same for our program and SIS. For AND–EXOR implementation, in only two cases (a04 and clip) SIS may possibly outperform our ATPG program, because the pattern set cardinality for our program is slightly higher than the lower bound of the SIS pattern set cardinality. It is, however, very probable that in these two cases SIS generates more test patterns for the compared test points. Run-times of the programs favor SIS, but mostly in the cases of low complexity. For some larger input files (e.g. AND–OR rd84, AND–EXOR mark1), SIS seems to have problems with efficiency.

Table XIII shows how many minimal and near minimal test sets exist for some example circuits. It can be seen that, for larger circuits, such as mark1, for which extremely large numbers of various test sets are possible, the number of minimal or near minimal solutions is very small. Thus, only a good search algorithm will be capable of finding a minimal or near-minimal test set for such circuits in a reasonable time. Therefore, it can be concluded that the realized ATPG program uses an adequate search algorithm.

The results presented in Table XII also provide some arguments to the discussion on which circuits, AND–OR or AND–EXOR, are better testable [27–32]. We can conclude that the number of deterministic test patterns in the minimal test set strictly depends on a particular function. For most of the functions that we checked, their minimal two-level AND–EXOR implementations required less deterministic test patterns than their minimal two-level AND–OR implementations. However, for a number of functions, AND–OR implementations required less test patterns, and for some of them (e.g. a04 and a05), the AND–OR implementations required much less patterns.

CONCLUSIONS

In this paper, we introduced term trees (TTs): decision trees similar to BDDs, but often enabling more compact modeling of Boolean functions, and moreover, enabling a compact representation of AND–EXOR and AND–OR circuit structures and their structural fault models. We showed how TTs can be used for an effective and efficient test-pattern generation and discussed results of the TT-based ATPG program.

The high effectiveness and efficiency of the ATPG program results from an appropriate composition of the following factors:

Usage of the term trees, which compactly represent faults from the fault model and enable easy construction of test patterns.

Intelligent construction, instead of generation, of the test patterns.

Usage of test patterns with "don't cares" instead of test vectors without "don't cares", which is equivalent to imposing minimum requirements on test patterns for a certain fault.

Combination of test patterns for various faults by an appropriate assignment of the pattern's "don't cares".

Usage of an efficient data structure for the cover matrix.

Usage of the preprocessing methods in the construction and processing of the cover matrix.

Application of an effective and efficient beam-search algorithm to find the minimal test sets from the cover matrix.

We also discussed an effective, efficient and flexible term-tree minimization algorithm and benchmark results produced by the C++ program that implements the algorithm. Execution of the presented algorithms in a parallel processing environment, to further improve their effectiveness and efficiency, does not require any substantial changes in the algorithms.

Naturally, we demonstrated only one of many possible applications of the term trees, a certain way of using them for ATPG and a particular minimization method. We are conscious that term trees can be used for many more purposes in logic design and in other areas. For example, they can be used as data representation in hazard-free synthesis of asynchronous sequential circuits. In general, they can be applied everywhere, where the compact representation and efficient manipulation of term sets is important. We know that other, maybe better, ATPG and term-tree minimization algorithms are possible. We demonstrated, however, that term trees constitute a good data representation, presented some new insights into the nature of aspects that are important for an effective and efficient ATPG, and we have shown a way to construct effective and efficient constructive search algorithms for ATPG. Many of the presented concepts are independent of the particular classes of circuits or the fault model considered in this paper, and can be used in ATPG algorithms for another classes of circuits and another fault models (e.g. intelligent construction of test patterns instead of generation; usage of test patterns with "don't

cares" instead of test vectors; combination of test patterns for various faults; usage of preprocessing methods not only by processing, but also by construction of the coverage matrix; application of the beam-search algorithm that considers the elements of the coverage matrix in a proper order and bases its decisions on the essentiality, potentiality and effectiveness of the covering elements). Many of them can be used to solve any coverage problems and other, similar problems. In this way, we provide researchers and developers of CAD tools with a collection of concepts that can be applied not only for AND–EXOR or AND–OR circuits and solving some ATPG problems, but in a much broader context.

Acknowledgements

The authors are indebted to the following students of Lech Jóźwiak: W.F.A. Bosch, M.J. Groen, M.A.J. Kolsteren and M.J.M. Looijmans for their contribution in development of algorithms and programs and in performing experiments.

References

[1] Akers, S.B. (1978) "Binary decision diagrams", *IEEE Trans. Comput.* **27**(6), 509–516.
[2] Bryant, R.E. (1986) "Graph-based algorithms for Boolean function minimization", *IEEE Trans. Comput.* **C-35**(8), 677–691.
[3] deMicheli, G. (1994) *Synthesis and Optimization of Digital Circuits* (McGraw-Hill, New York).
[4] Sasao, T. (1992) *Logic Synthesis and Optimization* (Kluwer Academic Publishers, Dordrecht).
[5] Sasao, T. and Fujita, M. (1996) *Representations of Discrete Functions* (Kluwer Academic Publishers, Dordrecht).
[6] Drechsler, R. and Becker, B. (1998) *Binary Decision Diagrams* (Kluwer Academic Publishers, Dordrecht).
[7] Chang, H.P., Rogers, W.A. and Abraham, J.A. (1986) "Structured functional level test generation using binary decision diagrams", *IEEE Int. Test Conf.*, 97–104.
[8] Naim, B.H. and Kaminska B. (1991) "Hierarchical functional level testability analysis," *IEEE European Test Conference, Munich*, pp. 327–332.
[9] Ubar, R., Kuchcinski, K. and Peng, Z. (1990) "Test Generation for Digital Systems at Functional Level." Research report, Institutionen för datavetenskap, Linköping, April.
[10] Jóźwiak, L. and Mijland, H.P.J. (1992) "OR-BDDs: modelling logical structures for test generation," *6th Workshop on New Direction for Testing*, Montreal, Canada, May 21–22.
[11] Jóźwiak, L. and Mijland, H.P.J. (1992) "On the use of OR-BDDs for test generation", *Microprocessing and Microprogramming* **35**(1–5).
[12] Kolsteren, M.A.J. (1993) "Heuristic Algorithms for the Construction of OR-BDDs," trainee report, supervisor: Dr L. Jóźwiak, Section of Digital Information Systems, Faculty of Electrical Engineering, Eindhoven University of Technology, January.
[13] Sasao, T. (1997) "Ternary decision diagrams—survey", *ISMVL'97*, 241–250.
[14] Jóźwiak, L. (1997) "On the use of term trees for effective and efficient test pattern generation," *EUROMICRO'97*, Budapest, Hungary, September 1–4, pp. 87–95.
[15] Mijland H.P.J. (1992) "On the use of OR-BDDs for logical structure modeling an fault detection," Eindhoven University of Technology Technical Report, ISBN 90-5282-206-9.
[16] Nilsson, N.J. (1971) *Problem-Solving Methods in Artificial Intelligence* (McGraw-Hill, New York).
[17] McElvain, K. *IWLS'93 Benchmark Set: Version 4.0*, Distributed as a part of IWLS'93 benchmark set.
[18] Brayton, R., Hachtel, G., McMullen, C. and Sangiovanni-Vincentelli, A. (1994) *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, Dordrecht).
[19] Jóźwiak, L. (1992) "An efficient heuristic method for state assignment of large sequential machines", *J. Circuits, Syst. Comput.* **2**(1), 1–26.
[20] Klenke, R.H., Williams, R.D. and Aylor, J.H. (1990) "Parallel processing techniques for automatic test pattern generation", *IEEE Trans. Comput.* **39**, 71–84.
[21] Wolf, J.M., Kaufman, L.M., Klenke, R.H., Aylor, J.H. and Waxman, R. (1996) "An analysis of fault partitioned parallel test generation", *IEEE Trans. CAD* **15**(5), 517–534.
[22] Patil, S. and Banerjee, P. (1989) "Fault partitioning issues in an integrated parallel test generation/fault simulation environment", *IEEE Int. Test Conf.*, 718–726.
[23] Motohara, A., Nishimura, K., Fujiwara, H. and Shirakawa, I. (1986) "A parallel scheme for test-pattern generation", *IEEE ICCAD*, 156–159.
[24] Klenke, R.H., Williams, R.D. and Aylor, J.H. (1993) "Parallelization methods for circuit partitioning based parallel automatic test pattern generation", *IEEE VLSI Test Symp.*, 102–107.
[25] Song, N. and Perkowski, M. (1996) "Minimization of exclusive sum of products expressions for multi-output multiple-valued input, incompletely specified functions", *IEEE Trans. CAD* **15**(4), 385–395.
[26] Sentovich, E., Singh, K, Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R. and Sangiovanni-Vincentelli, A. (1992) *SIS: A system for sequential circuit synthesis*, Memorandum No. UCB/ERL M92/41, University of California, Berkeley.
[27] Sarabi, A. and Perkowski, M.A. (1992) "Fast exact and quasi-minimal minimization of highly testable fixed-polarity AND/XOR canonical networks", *Design Automation Conf.*, 30–35.
[28] Saul, J. (1992) "Logic synthesis for arithmetic circuits using the Reed–Muller representation", *Eur. Conf. Design Automation*, 109–113.
[29] Brand, D. and Sasao, T. (1993) "Minimization of AND–EXOR expressions using rewrite rules", *IEEE Trans. Comput.* **42**, 568–576.
[30] Escobar, M. and Somenzi, F. (1995) "Synthesis of AND–EXOR expressions via satisfiability," *IFIP WG 10.5 Workshop on Application of the Reed–Muller Expansion in Circuit Design*, pp. 80–87.
[31] Hirayama, T. and Nishitani, Y. (1995) "A simplification algorithm of AND/EXOR expressions for multiple-output functions," *IFIP WG 10.5 Workshop on Application of the Reed–Muller Expansion in Circuit Design* pp. 88–93.
[32] Drechsler, R., Hengster, H., Schafer, H., Hartman, J. and Becker, B. (1997) "Testability of 2-Level AND/EXOR circuits", *ED&TC'97*, 548–553.
[33] Inoue, T., Fuji, T. and Fujiwara, H. (1994) "On the performance analysis of parallel processing for test generation", *IEEE Asian Test Symp.*, 69–74.

Authors' Biographies

Lech Jóźwiak received his M.Sc. and Ph.D. degrees in Electronics from the Warsaw University of Technology, Poland, in 1976 and 1982, respectively. From 1976 to 1979, he worked at the Faculty of Electronics of this University. From 1979 to 1986, he was the chief of the R&D team in the Research Institute of Computers in Warsaw. From 1986 till now, he is an associate professor in the Faculty of Electronics, Eindhoven University of Technology, The Netherlands. His research interests include design theory and technology, electronic design automation, circuit and system theory and technology, decision theory and artificial intelligence in design, hardware/software codesign, custom computing machines, re-configurable computing, quality-driven

design, dependable computing, testing, and design validation. He is an author of about 100 journal and conference papers and of some book chapters. He is an advisor to the industry and to the Ministry of Economy. He is a member of IEEE and Board of Directors Member of EUROMICRO.

Aleksander Ślusarczyk received his M.Sc. degree (*cum laude*) in Computer Science from Department of Electronics of the Warsaw University of Technology in 1998. During the academic year 1996/97, he took part in the MSc in Computer and Communication Networks program at the University of Wales in Swansea. Since 1998, he works as a scientific assistant and Ph.D. candidate in the Section of Information and Communication Systems (ICS) at the Eindhoven University of Technology. His present research interests include digital circuit theory, methods and tools for automatic logic synthesis (in particular sequential synthesis targeting FPGA implementation) and application of artificial intelligence methods in design.

Marek Perkowski has his Ph.D. in Automatic Control from the Technical University of Warsaw, Warsaw, Poland. He served on the faculties of Technical University of Warsaw and University of Minnesota. Currently he is professor of Electrical and Computer Engineering at Portland State University, Portland, Oregon. He has been a visiting professor at the University of Montpellier, France, and Technical University of Eindhoven, The Netherlands. He was also a summer professor and consultant at Wright Laboratories of U.S. Air Force, GTE, Intel, Cypress, Sharp, and other high technology and EDA companies. Dr Perkowski is the general chair of International Symposium on Multiple-Valued Logic, 2000, the Vice-Chair for Technical Activities of IEEE Technical Committee on MVL, and the Chair of Fourth Oregon Symposium on Logic, Design, and Learning, 2001. His recent research interests include spectral decision diagrams, functional decomposition, intelligent robotics, walking robots, robot theatre, axiomatic morality, and all applications of logic synthesis outside circuit design.