A Complement-Based Fast Algorithm to Generate Universal Test Sets for Multi-Output Functions

Beyin Chen and Chung Len Lee, *Senior Member, IEEE*

***Abstract*—In this paper, a fast universal test set (UTS) generation algorithm for multi-output functions is presented.**

The algorithm first generates the UTS for single-output functions by directly Shannon-expanding and complementing the function. This significantly reduces the time complexity and the usage of temporary memory. Also, it stores tests in test cubes to save the size of memory for test storing. Two–six orders of magnitude in computation efficiency improvement and 1–1800 fold for memory saving over the conventional method are achieved. It then merges the generated test cubes for each single-output function into a set of mutually disjoint test cubes to be the UTS for a multi-output function by employing a new compaction technique. The size of UTS thus obtained is 1–20 times smaller than that of UTS without compaction.

I. Introduction

BASED ON THE unate function theory [1], a universal test set (UTS) for combinational circuits was proposed by Akers [2] and Reddy [3]. It was shown that the UTS can be generated from the functional description and can detect all single and multiple stuck-at faults of the circuit implementation which satisfies "unate gate network" [3] property for the function. Also, the UTS can be paired with a universal initialization set to detect every detectable stuck-open fault in a "restricted CMOS circuit" [4]. However, the procedure to generate the UTS involves a process to enumerate the truth table of the function and it has an exponential complexity. Also, even the computation can be speeded up, the size of UTS grows exponentially with the number of binate input variables. This makes the storage of test patterns a problem. Moreover, for a multi-output function, the compaction of compatible tests among the UTSs of the single-output functions is also a problem which has never been considered by researchers.

This paper first presents a fast algorithm, **FUTS**, to generate UTS for single-output functions. The algorithm adopts a method to find UTS by Shannon—expanding and complementing the function, hence completely eliminates the truth table enumeration process. Also, the method treats the test information in terms of "test cubes" instead of "test patterns," and this solves the storage problem for the UTS. Experimental results show that the algorithm achieves improvements of 10^2–10^6 times in the computation efficiency and 1–1800 times in the storage saving of test sets over the conventional method.

Manuscript received February 18, 1993. This work was supported by The National Science Council, Republic of China, under Grant NSC82-0404-E009-183. This paper was recommended by Associate Editor K.-T. Cheng.

The authors are with the Department of Electronics Engineering & Institute of Electronics, National Chiao Tung University, Hsinchu, Taiwan, Republic of China.

IEEE Log Number 9214057.

Then this paper presents a compaction technique to merge the generated test cubes into a set of *mutually disjoint* test cubes to be the UTS for multi-output functions. Experimental results show that the size of UTS generated with the technique is 1–20 times smaller than that of UTS without compaction.

II. Denotations

In this section, some terms and denotations which are used in this paper are first given.

A logic function F has n input variables $x_1, x_2, \ldots, x_n$ and is represented in the *sum-of-products* form: $F = P_1 + P_2 + \cdots + P_k$, where P_j is represented by a **cube** c_j [5].

The set of k cubes defined as above is said to be a cover of F, denoted as **cover(F)**. For the cover(F), it can be minimized to be prime and irredundant [5], which is denoted as **mini_cover(F)**.

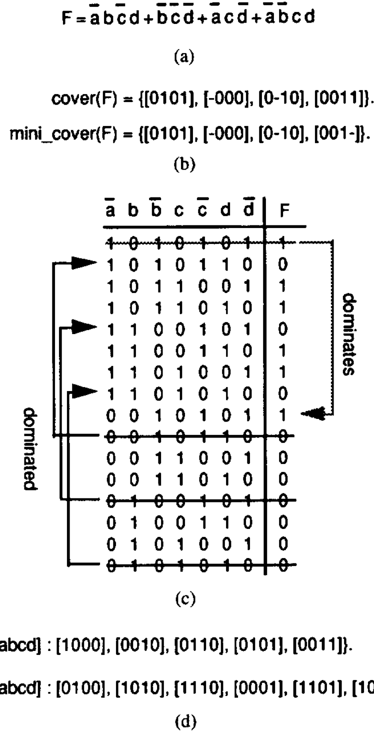
The **expanded truth table** of a logic function is the truth table of the input literals of the input variables. In the expanded truth table, an input vertex X **dominates** the input vertex Y if and only if the entry of X is 1 where the corresponding entry of Y is 1. For example, if $X = 1101$ and $Y = 1001$ then X dominates Y. A **minimal true vertex** of a logic function is the input vertex that does not dominate any other true vertex except itself. **A maximal false vertex** of a logic function is the input vertex that is not dominated by any other false vertex except itself. The **UTS** of a logic function F is the union of the minimal true vertex set and the maximal false vertex set of the expanded truth table of F [2], [3]. It is denoted as: $\mathbf{UTS(F) = V_{mint}(F) + V_{maxf}(F)}$.

An example as shown in Fig. 1 is to demonstrate the above, where the sum-of-products form, cubical representations, and the expanded truth table of F are shown in Fig. 1 (a), (b), and (c) respectively. The $V_{\text{mint}}(F)$ and $V_{\text{maxf}}(F)$ obtained, which constitute the UTS of F, are shown in Fig. 1 (d).

In the **Shannon expansion** of the function F, i.e., $F = x_i.F_{x_i} + \overline{x_i}.F_{\overline{x_i}}, F_{x_i}$ and $F_{\overline{x_i}}$ are the **one-cofactor** and the **zero-cofactor** of F with respect to the **splitting variable** x_i respectively. A **cofactor is strict_unate** if it is independent of all the binate input variables of F. A cofactor is **tautology** or **nil** if it is always logic true or false respectively for all its input combinations. It is seen that tautology and nil are two special cases of strict_unate cofactors.

The UTS generated by the method of this paper is represented by test_cubes instead of the conventional test patterns. A **test_cube** is a subset of UTS represented by a cube. A test_cube is said to be **true** with respect to an output F if it makes the output logic 1. It is **false** if it makes the output logic

0278-0070/94$04.00 © 1994 IEEE



$V_{mint}(F)$ = {[abcd] : [1000], [0010], [0110], [0101], [0011]}.

$V_{maxf}(F)$ = {[abcd] : [0100], [1010], [1110], [0001], [1101], [1011], [0111]}.

(d)

Fig. 1. An example to demonstrate the conventional method to generate the universal test set. (a) Sum-of-products form of function F. (b) Cubical representation of F. (c) Expanded truth table of F. (d) The minimal true vertex set and maximal false vertex set of F.

0. The test_cover of F, denoted as **test_cover(F)**, contains all the true and false test_cubes of F, and is the UTS(F). For example, the UTS(F) in Fig. 1 contains 5 true tests and 7 false tests, which can be represented by 4 true test_cubes = {[1000], [001-], [0110], [0101]} and 6 false test_cubes = {[0100], [101-], [1110], [0001], [1101], [0111]}. Note that in the above, test_cube (001-) represents test patterns (0010) and (0011).

III. A Fast Algorithm to Generate UTS for Single-Output Functions

To present the algorithm **FUTS**, a theorem and some propositions and lemmas are first presented.

Proposition 1: For an input variable x_i of function F, if it is positively (negatively) unate for F, then it is negatively (positively) unate for $\overline{F}$, which is the complement of F. If x_i is binate for F, then it is still binate for $\overline{F}$.

Applying Proposition 1 to the definition of $V_{\text{maxf}}(F)$ and $V_{\text{mint}}(\overline{F})$, the following proposition holds.

Proposition 2: For a function F, its maximal false vertex set is equal to the minimal true vertex set of its complement $\overline{F}$, i.e., $V_{\text{maxf}}(F) = V_{\text{mint}}(\overline{F})$.

Theorem 1: $UTS(F) = x_i[UTS(F_{x_i})] + \overline{x_i}[UTS(F_{\overline{x_i}})]$, where x_i is a binate input variable of F.

Proof: For two row vectors X and Y in the expanded truth table of F, if there exists a binate input variable for which the values of X and Y are different, then X and Y never dominate each other. So, to find UTS(F), we can divide the expanded truth table into two for x_i to be 0 and 1 and then find their UTS's respectively. Based on this and the definition of Shannon expansion, this theorem holds. □

With Theorem 1, a binate function F can be recursively expanded into two cofactor functions, and this recursion can be continued until all the cofactor functions are strict_unate. The UTS of F can be obtained directly by using Theorem 1 by combining the UTS's of the cofactor functions.

For a cofactor function C which is strict_unate, Lemmas 1, 2, and 3 can be used to obtain its test_cubes.

Lemma 1: If C is tautology, then UTS(C) contains a single true test_cube t and it can be obtained as follows:

for (i := 1 to n)
if (x_i is not split) then
$t^i = 0$ if x_i is positively unate w.r.t. F;
$t^i = 1$ if x_i is negatively unate w.r.t. F;
t^i = - if x_i is binate w.r.t. F,

where t^i is an entry of the true test_cube $t = [t^1, \ldots, t^n]$.

Lemma 2: If C is nil, then UTS(C) contains a single false test_cube f and it can be obtained as follows :

for (i :=1 to n)
if (x_i is not split) then
$f^i = 1$ if x_i is positively unate w.r.t. F;
$f^i = 0$ if x_i is negatively unate w.r.t. F;
f^i = - if x_i is binate w.r.t. F,

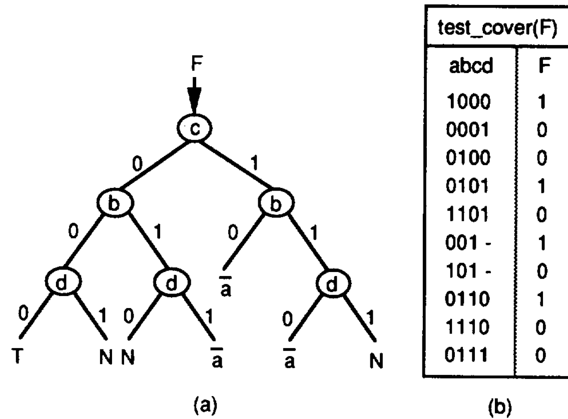
where f^i is an entry of the false test_cube $f = [f^1, \cdots, f^n]$.

Lemma 3: If C is strict_unate and is neither tautology nor nil, and mini_cover(C) and mini_cover($\overline{C}$) have k cubes $c_1, \ldots, c_k$ and m cubes $b_1, \ldots, b_m$ respectively, then UTS(C) contains k true test_cubes $t_1, \ldots, t_k$ and m false test_cubes $f_1, \cdots, f_m$ which can be obtained as follows :

for (j :=1 to k)
for (i := 1 to n)
if (x_i is not split) then
$t_j^i = c_j^i$ if c_j^i = 1 or 0;
$t_j^i = 0$ if $c_j^i = 2$ and x_i is positively unate w.r.t. F;
$t_j^i = 1$ if $c_j^i = 2$ and x_i is negatively unate w.r.t. F;
t_j^i = - if $c_j^i = 2$ and x_i is binate w.r.t. F,
where t_j^i is an entry of the true test_cube $t_j = [t_j^1, \ldots, t_j^n]$; and
for (j := 1 to m)
for (i := 1 to n)
if (x_i is not split) then
$f_j^i = b_j^i$ if b_j^i = 1 or 0;
$f_j^i = 1$ if $b_j^i = 2$ and x_i is positively unate w.r.t. F;
$f_j^i = 0$ if $b_j^i = 2$ and x_i is negatively unate w.r.t. F;
f_j^i = - if $b_j^i = 2$ and x_i is binate w.r.t. F,
where f_j^i is an entry of the false test_cube $f_j = [f_j^1, \cdots, f_j^n]$.

The proofs of Lemmas 1, 2, and 3 can be done by applying Propositions 1 and 2 and Theorem 1.

For the example function of Fig. 1, the Shannon expansion process for F to find its test_cubes can be represented by a binary tree as shown in Fig. 2(a), where T and N are cofactors which are tautology and nil respectively. After applying Lemmas 1, 2, and 3 to the co factors and merging the test_cubes by setting the splitting variables to either '0' or '1' according to the path values in the binary tree, the



test_cover(F)	
abcd	F
1000	1
0001	0
0100	0
0101	1
1101	0
001 -	1
101 -	0
0110	1
1110	0
0111	0

Fig. 2. The demonstration of the algorithm FUTS using the example of Fig. 1. (a) Shannon expansion process for F to find its test_cubes. (b) The test_cover of F which constitutes the UTS of F.

test_cubes of F are generated and shown in Fig. 2(b) which is the test_cover of F.

As demonstrated with this example, the expanded truth table enumeration and the comparing operations are completely eliminated. Also, for the conventional method, 12 tests as shown in Fig. 1 were generated, while here, only 10 test_cubes are generated. In general, the size of memory for test storing is largely reduced by using this method.

Finally, it is specially mentioned that to generate the test_cover of an output function using FUTS, the intersection of any two test_cubes in the test_cover is empty.

The details of the algorithm FUTS are presented as follows:

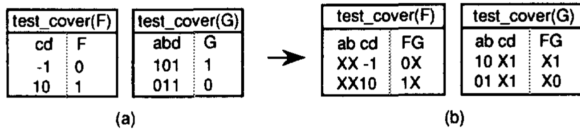
Algorithm FUTS(F);
INPUT: F /* a set of cubes. */
OUTPUT: test_cubes /* a set of test_cubes that
  constitutes the UTS of F. */
{
Make F a prime cover.
Scan the prime cover, check for every input variable
  to determine whether it is positively,
    negatively unate, or binate.
test_cubes := UTS_GEN (F);
}
Procedure UTS_GEN(cofactor);
{
if (cofactor == nil) **then**
/* Lemma 2 is applied. */
    test_cubes := NIL_CASE( );
else if (there is a row of all '-'s in cover(cofactor))
/* Lemma 1 is applied. */
    test_cubes := TAUTOLOGY_CASE( );
else if (cofactor is strict_unate)
/* Lemma 3 is applied. */
    test_cubes := UNATE_CASE(cofactor);
else
/* Theorem 1 is applied. */
    x_j := SPLIT_SELECT(cofactor);
/* In general, a most binate variable of cover(cofactor)
  is selected. */
    one-cofactor := ONE_COF(cofactor, x_j);
    zero-cofactor := ZERO_COF(cofactor, x_j);
/* Compute the one-cofactor and the zero-cofactor
  w.r.t. x_j respectively. */
    test_cubes := MERGE (UTS_GEN(one-cofactor),
    UTS_GEN(zero-cofactor), x_j);
/* The variable x_j in test_cubes generated from
  UTS_GEN(one-cofactor) is set to 1. */
/* The variable x_j in test_cubes generated from
  UTS_GEN(zero-cofactor) is set to 0. */
return (test_cubes);
}
Procedure UNATE_CASE(cofactor);
{
mini_cover := UNATE_SIMPLIFY (cofactor);
Generate k true test_cubes corresponding to the k cubes
    in mini_cover; /* Lemma 3 */
comp_ cover := UNATE_COMPLEMENT (mini_ cover);
Generate m false test_cubes corresponding to the m cubes
    in comp_cover; /* Lemma 3 */
return (the k true test_cubes + the m false test_cubes);
}

Note that in the above Procedure UNATE_CASE( ), UNATE_SIMPLIFY( ) is to make a unate cover prime and irredundant and UNATE_COMPLEMENT( ) is to compute the complement of a unate cover. There are published procedures [5] which are simple and fast to do the above. In addition, if UNATE_COMPLEMENT( ) is used to complement a prime and irredundant cover, e.g., mini_cover(C), the returned cover, $\overline{C}$, is prime and irredundant too. So, comp_cover is prime and irredundant and Lemma 3 can be applied.

IV. UTS Compaction for Multi-Output Functions

To generate tests from a multi-output function, all the outputs of the circuit of the realized function are assumed to be independent of one another and every output circuit satisfies the property of "unate gate network" as proposed in [3]. The multi-output UTS can be easily obtained by finding the UTS for every single-output function by using the UTS generation algorithm FUTS and then combining the UTS for every single-output function. However, since the test_covers generated for the single-output functions have compatible test patterns of one another, it needs to compact the tests among test_covers. Since the number of patterns of the generated UTS may increase exponentially to the number of the function inputs, it is impossible to compact them in a reasonable time if the conventional compaction method, which can only compact test patterns, is used. In the following, some denotations are given and then a procedure is proposed to compact tests directly from test_cubes.

For a multi-output function, any of its single-output is usually dependent on a subset of primary inputs. The inputs that do not belong to an output function are **independent inputs** (or **don't care inputs**) with respect to the output function and are represented by X (**don't care**) in the generated test_cubes. For example, consider a function which has two outputs F and G. Assume that their test_cubes are as



(a)

test_cover(F)	
cd	F
-1	0
10	1

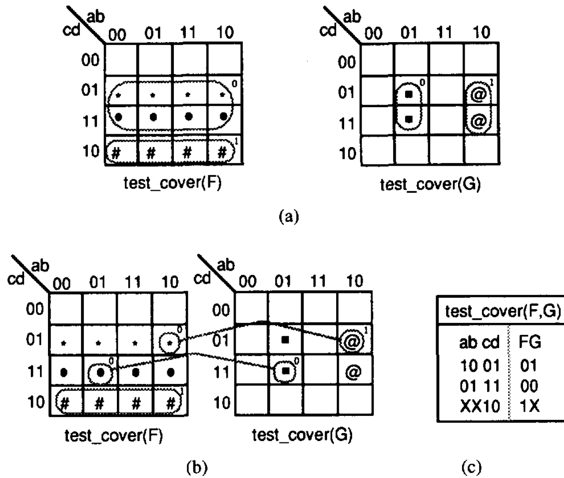
test_cover(G)	
abd	G
101	1
011	0

(b)

test_cover(F)	
ab cd	FG
XX -1	0X
XX10	1X

test_cover(G)	
ab cd	FG
10 X1	X1
01 X1	X0

Fig. 3. An example to demonstrate the test_cover representation for multi-output functions.



test_cover(F,G)	
ab cd	FG
10 01	01
01 11	00
XX10	1X

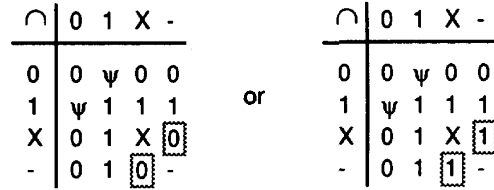
Fig. 4. (a) Karnaugh map representation for test_covers of F and G of Fig. 3(b). (b) An optimum compaction performed from the Karnaugh map of (a). (c) The cubical representation of the compaction result of (b).

shown in Fig. 3(a). Since F and G are independent of $\{a, b\}$ and c respectively, test_cover(F) and test_cover(G) can be re-expressed as shown in Fig. 3(b). The columns correspond to inputs a, b in test_cover(F) and input c in test_cover(G) are all X's. Note that the meaning of the symbol '–' is the same as mentioned in the previous sections. For example, the false test_cube $(XX-1)$ of F as shown in Fig. 3(b) contains two **test patterns** $(XX01)$ and $(XX11)$.

The example of Fig. 3 is used to demonstrate the problem to compact tests between two test_covers. Fig. 4(a) shows the Karnaugh map representation of the test_covers of F and G of Fig. 3(b), where each type of symbols of the entries of the maps represents a test. For example, test_cover(F) contains three tests, so there are three types of symbols. Also since there are two independent inputs, i.e., a and b, for output function F, there are four choices for a, b for the three tests respectively. The problem to find the test_cover(F, G) is to find a minimal set of input patterns to cover all types of symbols in the Karnaugh maps. In this example, there exist two optimum solutions for test_cover(F,G), one of which is shown in Fig. 4(b) and its cubical representation is shown in Fig. 4(c). It is noted that don't cares (X's) are preserved for further compaction if there are more outputs.

Since the test compaction is a problem equivalent to the minimum coloring problem, which is NP-complete, in the following, two operations for test_cubes are defined and a heuristic compaction method is proposed to solve the problem.

It is the order that the test_covers are to be compacted affects the compaction efficiency. A weight: **Essentiality(TC)** for a test_cover TC, is first defined to guide the ordering to compact the test_covers.



∩	0	1	X	-
0	0	ψ	0	0
1	ψ	1	1	1
X	0	1	X	0
-	0	1	0	-

or

∩	0	1	X	-
0	0	ψ	0	0
1	ψ	1	1	1
X	0	1	X	1
-	0	1	1	-

Fig. 5. The operation rules for test_cube intersection.

$$\text{Essentiality(TC)} = 1 - \frac{\text{The number of don't care inputs of TC}}{\text{The number of total inputs}}$$

The value of an essentiality is 1 if the test_cover is a function of all inputs, and is 0 if the test_cover is independent of all the inputs. The larger of the essentiality of the test_cover is, the earlier of the test_cover should be selected to be compacted.

A procedure, **TEPACT**, is developed to compact test_covers. TEPACT selects two test_covers at one time, according to the weights of the test covers, to do compaction. To do compaction, it invokes a **cover compaction** operation to compact every test_cube of a test_cover with the test cubes of the other test_cover in a one by one order. The **cube compaction** operation involves two steps:

Step 1. Perform the intersection operation to obtain a compatible cube of the two cubes.

Step 2. Delete the compatible cube from the two cubes respectively.

The **intersection** operation (∩) of two cubes is to find their compatible cube. The operation rules are summarized in Fig. 5. A ψ generated during the intersection process for two cubes means that the two cubes are not compatible and no compatible cube is generated. In the table, it is noted that $X \cap$- is defined to be either 0 or 1.

To do the operation of Step 2, the **disjoint sharp** operation (*) which was defined in [6] is used. The operation can be used to delete the compatible cube of a cube from itself and make the resultant cubes mutually disjoint. It may generate more than one cubes. For example: $(X--0)*(1000) = \{(X1-0), (X010)\}$.

The procedure of TEPACT is:

Procedure TEPACT(test_covers)

/* INPUT: the test_covers generated by FUTS. OUTPUT : test_cover_H */

Step 1. Compute Essentiality for every test_cover. Determine the order of the test_covers to be compacted according the Essentiality value;

Step 2. Get two test_covers, test_cover_F and test_cover_G;

Step 3. Perform cover compaction : test_cover_H := cover_compaction(test_cover_F,test_cover_G);

Step 4. If there is test_cover uncompacted, then let test_cover_F := test_cover_H and test_cover_G := next test_cover, go to Step 3.

Otherwise, return test_cover_H and exit from this procedure.

We use the example of Fig. 3 to demonstrate TEPACT.

test_cover(G)	test_cover(F)	compatible test_cover(F,G)	Comments
10 X1 X1 → 01 X1 X0	XX -1 0X XX10 1X	∅	Initial State.
01 X1 X0 →	XX10 1X XX11 0X	1001 01	Result of the first time of cube compaction.
∅	XX10 1X	1001 01 0111 00	Result of the second time of cube compaction.

(a)

test_cover_H = compatible test_cover(F,G) + test_cover(G) + test_cover(F)

= {(1001 01), (0111 00), (XX10 1X)}

(b)

Fig. 6. The demonstration of the process of TEPACT using the example of Fig. 3. (a) Test_cover compaction process. (b) The resultant test_cover after the compaction process of (a).

First, since the essentiality of G is larger, the test_cover(G) is selected to compact with the test_cover(F). Initially, an empty cover, **compatible test_cover(F,G)** is created to store the compatible cubes of test_covers F and G, as shown in the first row of Fig. 6(a). The cube $(10X1, X1)$ is intersected with $(XX-1, 0X)$ and the resultant cube $(1001, 01)$ is put into compatible test_cover(F,G). $(1001, 01)$ is then disjoint sharped with $(10X1, X1)$ and $(XX-1, 0X)$ respectively. The results are $\emptyset$ (empty set) and $(XX11, 0X)$ respectively. The old cubes $(10X1, X1)$ and $(XX-1, 0X)$ are then removed and the new cube $(XX11, 0X)$ is appended to test_cover(F). This result of cube compaction is shown in the second row of Fig. 6(a). The same process is repeatedly performed until one of the two test_covers is empty. The resultant test_cover_H after this compaction process is shown in Fig. 6(b). It is seen that the number of tests is reduced from five to three and this is the optimum compaction result which is shown in Fig. 4(c).

With the above, the procedure, **MOUTS**, to generate the UTS for a multi-output function is summarized as follows:

Procedure MOUTS /* To generate the UTS for a multi-output function */

Step 1. Extract all the single-output functions from the original multi-output function. Find their independent input sets respectively.

Step 2. Generate the test_cover for every single-output function using the UTS generation algorithm FUTS.

Step 3. Perform procedure TEPACT for the generated test_covers to obtain the UTS of the multi-output function.

V. Experimental Results

A. Results on FUTS

The above algorithm and the conventional method to find UTS have been implemented in C language to run on a sun4/SPARC2 workstation. They are denoted as FUTS and CUTS respectively and were applied to run on 18 benchmark functions [5], which are represented as sum-of-products forms, to generate UTS. Since these benchmark functions are multiple-output functions, for each of them, we randomly select and extract an output as a single-output function. Table I gives the results obtained for FUTS and CUTS. In the table, column 2 is the number of input variables, column 3 is the number of binate input variables. Column 4 shows the number of test_cubes obtained with FUTS, column 5 shows the size of UTS, and column 6, which is obtained by dividing column 5 by column 4, is the memory saving of using "test_cubes" to store the test patterns. Column 7 and column 8 are the CPU times spent by FUTS and CUTS to obtain UTS respectively, and column 9 is the ratio of improvement of FUTS over CUTS obtained by dividing column 8 by column 7. It can be seen that 10^2–10^6 fold improvement is obtained for these 18 functions. It is to be specially mentioned that, for the functions: rckl, x6dn, in4, whose input numbers are larger than 30, more than 2^{30} bytes ($= 1000$ Mbytes) of temporary memory are needed for CUTS for truth table enumeration! Yet, for FUTS, for the largest benchmark function, in4, only 1 s of CPU time was spent to obtain its UTS and 4K-byte memory was used to store its test_cubes.

TABLE I
The Run Results on 18 Benchmark Functions [5] By FUTS and CUTS

Function			# of test cube	Size of UTS	Storage Saving Ratios	Running Time (sec.)		Time Saving Ratios
name	#in	#bi				FUTS	CUTS	
bc0	18	15	247	46362	188	0.13	176760	1325700
bca	15	2	41	41	1	0.00	12	*
bcb	15	13	178	8256	46	0.08	8711	104532
bcc	15	11	417	2317	6	0.23	1993	8541
bcd	15	7	30	144	5	0.00	118	*
chkn	16	3	24	26	1	0.00	36	*
gary	11	10	51	1034	20	0.02	37	2220
in0	15	13	153	8348	55	0.12	4671	40037
in1	14	12	81	4164	51	0.10	1270	12700
in2	18	8	81	732	9	0.05	4288	85760
in4	31	20	1148	2150080	1873	0.90	O.O.M.	*
in5	16	8	166	591	4	0.08	875	10500
rckl	32	0	33	33	1	0.08	O.O.M.	*
vg2	18	10	588	3748	6	0.42	15990	38376
x1dn	14	3	66	74	1	0.03	16	480
x6dn	34	13	321	24944	78	0.30	O.O.M.	*
x7dn	11	5	109	182	2	0.03	4	120
x9dn	14	3	66	74	1	0.02	31	1860

Storage Saving Ratios : test-storing saving ratios.
O.O.M. : out of memory. * : the ratio can not be calculated.

To demonstrate the efficiency of the generated UTS, the functions in Table I are synthesized by the multi-level logic synthesizer misII [7], and then the fault coverages for these circuits are simulated for both the generated UTS's in Table I and the same number of randomly generated patterns. The simulation results are shown in Table II. In the table, column 2 is the number of gates synthesized by misII. Column 3 is the number of detectable faults (the number in parenthesis is the number of redundant faults.). Column 4 is the number of simulated patterns. Column 5 and column 6 show the fault coverages obtained by simulating the UTS's and the randomly generated patterns respectively. The fault coverages in both the columns are obtained by dividing the number of detected faults by the number of detectable faults. We can see that the generated UTS's reach 100% fault coverages for all the circuits. But for random patterns, the fault coverages may be low to 3%.

TABLE II
THE FAULT SIMULATION RESULTS FOR THE GENERATED UTS'S IN TABLE I AND THE SAME NUMBER OF RANDOM PATTERNS

Fun.	#gate	#fault	#pattern	Fault Coverages UTS	random
bc0	120	600	46362	100.00%	98.00%
bca	15	68	41	100.00%	4.41%
bcb	65	346	8256	100.00%	79.48%
bcc	175	906	2317	100.00%	62.47%
bcd	17	92	144	100.00%	3.26%
chkn	15	78	26	100.00%	24.36%
gary	30	(+2)152	1034	100.00%	100.00%
in0	77	(+4)384	8348	100.00%	98.63%
in1	60	(+1)283	4164	100.00%	100.00%
in2	27	138	732	100.00%	78.26%
in4	144	532	2150080	100.00%	98.31%
in5	45	228	591	100.00%	45.61%
rckl	19	102	33	100.00%	19.61%
vg2	43	198	3748	100.00%	100.00%
x1dn	15	78	74	100.00%	6.41%
x6dn	91	(+1)421	24944	100.00%	100.00%
x7dn	24	104	182	100.00%	99.04%
x9dn	17	82	74	100.00%	30.49%
Ave.				100.00%	63.80%

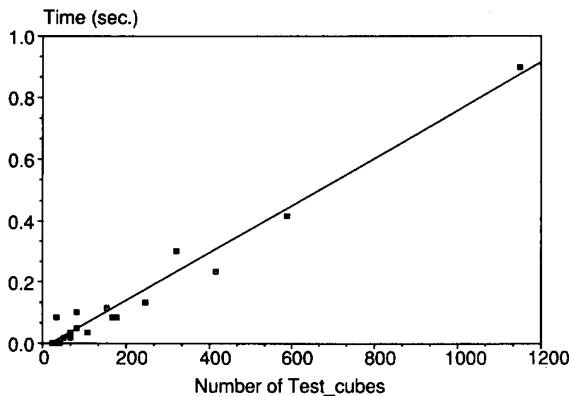
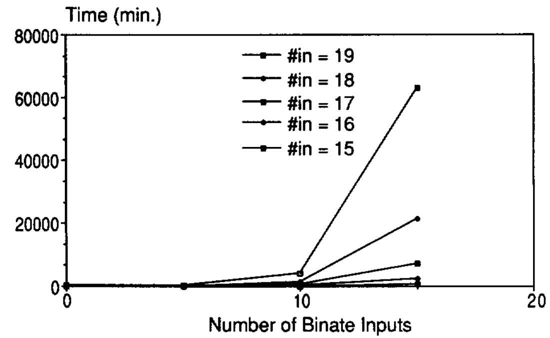


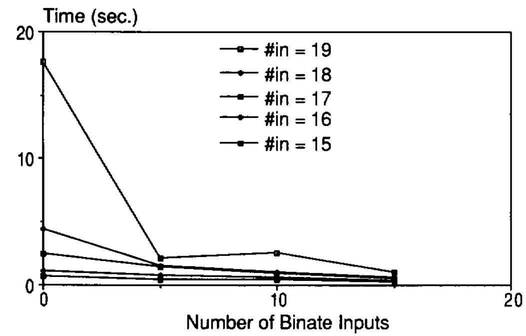
Fig. 7. The plots of computation times with respect to the numbers of "test_cubes" of the 18 benchmark functions run by FUTS in Table I.

For the algorithm CUTS to find UTS, it is the number of the operation of "comparing two vertices to determine whether they dominate each other," which costs the computation time. The number of "comparing" operations is approximately proportional to the square of the number of vertices which need to be "compared." So, for CUTS, the time complexity is $O(2^n)$–$O(2^{2n})$ for an n-input function. Yet, for the algorithm FUTS, since no comparing operation is involved, the time complexity is approximately linear with respect to the number of "test_cubes". To demonstrate the relationship between CPU time spent by FUTS and the number of test_cubes, Fig. 7 is plotted for the 18 functions in Table I. It is seen that a linear curve is obtained.

In addition, since the size of UTS grows exponentially with the number of binate inputs, the number of "comparing" operations for CUTS to generate UTS increases rapidly with the number of binate inputs. For FUTS, on the contrary, the computation time decreases with the number of binate inputs since it Shannon-expands the function with respect to the binate input variables. This effectively applies a "divide-and-conquer" strategy to solve the problem. To demonstrate this, CUTS and FUTS were applied to several functions, which were composed of randomly generated but equal numbers of product terms, to generate their UTS's. The computation times to generate UTS's are plotted in terms of the numbers of binate inputs in Fig. 8(a) and (b) for CUTS and FUTS respectively. In Fig. 8, the numbers of input variables for these functions were from 15 to 19 and the numbers of binate inputs were varied from 0 to 5, 10, and 15. In Fig. 8(a) and (b), two facts can be observed : First, the computation times for CUTS are much much larger than that of FUTS, and, second, the computation times for CUTS increase exponentially with the numbers of binate inputs while for FUTS the computation times decrease with the numbers of binate inputs.



(a)



(b)

Fig. 8. The plots of computation times with respect to the numbers of binate inputs for the cases : input numbers = 15, 16, 17, 18, and 19 obtained by (a) CUTS and (b) FUTS respectively.

B. Results on MOUTS

The procedure MOUTS has also been implemented in C language to run on a sun4/SPARC2 workstation. It was applied to run on the benchmark functions of [5]. The results are compiled in Table III. In the table, the numbers of inputs and outputs of each function are also listed. For comparison, the

TABLE III
THE COMPARISON OF UTS SIZE AND CPU TIMES FOR TEST GENERATION PROCEDURES WITH AND WITHOUT TEPACT FOR 56 BENCHMARK FUNCTIONS [5]

Function Characteristics			Test Length			CPU Time (Sec.)		
name	#in	#out	without TEPACT	with TEPACT	reduction ratio	with TEPACT	without TEPACT	increase percent.
add6	12	7	5618	4096	1.37	0.69	0.67	3%
adr4	8	5	378	256	1.48	0.09	0.07	29%
alu1	12	8	46	8	5.75	0.02	0.01	100%
alu2	10	8	2695	1024	2.63	0.1	0.08	25%
alu3	10	8	2407	1024	2.35	0.07	0.05	40%
apla	10	12	360	143	2.52	0.09	0.07	29%
bc0	26	11	156320	124529	1.26	1.98	1.38	43%
bca	26	46	410217	32992	12.43	8.05	1.98	307%
bcb	26	39	329085	16733	19.67	3.7	1.57	136%
bcc	26	45	309750	16698	18.55	9.22	1.72	436%
bcd	26	38	153711	18669	8.23	2.77	1.10	152%
chkn	29	7	75920	66355	1.14	2.17	1.65	32%
co14	14	1	16384	16384	1.00	0.04	0.02	100%
cps	24	109	446153	217639	2.05	17.85	2.85	526%
dc1	4	7	68	16	4.25	0.03	0.01	200%
dc2	8	7	422	128	3.30	0.05	0.03	67%
dist	8	5	1055	256	4.12	0.15	0.13	15%
dk17	10	11	159	65	2.45	0.03	0.02	50%
dk27	9	9	44	14	3.14	0.02	0.01	100%
dk48	15	17	132	48	2.75	0.04	0.03	33%
exep	30	63	9846	8552	1.15	0.66	0.28	136%
f51m	8	8	510	256	1.99	0.03	0.02	50%
gary	15	11	56156	20924	2.68	1.2	0.37	224%
in0	15	11	56156	20924	2.68	1.16	0.33	252%
in1	16	17	227333	22712	10.01	2.81	1.13	149%
in2	19	10	22380	17977	1.24	1	0.50	100%
in3	35	29	2699	2119	1.27	0.53	0.45	18%
in4	32	20	2163416	2160568	1.00	2.05	1.52	35%
in5	24	14	21844	19248	1.13	0.59	0.47	26%
in6	33	23	3245	2083	1.56	0.3	0.22	36%
in7	26	10	4076	2073	1.97	0.17	0.15	13%
jbp	36	57	5131	1941	2.64	1.05	0.62	69%
misg	56	23	126	41	3.07	0.19	0.18	6%
mish	94	43	181	24	7.54	0.11	0.10	10%
mlp4	8	8	1127	256	4.40	0.11	0.10	10%
opa	17	69	3458	866	3.99	0.71	0.53	34%
radd	8	5	378	256	1.48	0.03	0.02	50%
rckl	32	7	230	191	1.20	0.34	0.33	3%
rd53	5	3	79	32	2.47	0.03	0.02	50%
rd73	7	3	326	128	2.55	0.14	0.13	8%
risc	8	31	210	50	4.20	0.04	0.03	33%
root	8	5	969	256	3.79	0.08	0.07	14%
sqn	7	3	260	128	2.03	0.03	0.02	50%
sqr6	6	12	294	64	4.59	0.04	0.03	33%
ti	47	72	218901	208981	1.05	6.8	1.67	307%
tial	14	8	38790	16384	2.37	2.93	2.28	29%
vg2	25	8	137048	134848	1.02	0.65	0.57	14%
wim	4	7	41	12	3.42	0.02	0.01	100%
x1dn	27	6	137032	134884	1.02	0.58	0.50	16%
x2dn	82	56	131408	131111	1.00	0.58	0.33	76%
x6dn	39	5	115748	114714	1.01	0.74	0.67	10%
x7dn	66	15	2735322	1555481	1.76	6.8	4.55	49%
x9dn	27	7	268140	134884	1.99	0.68	0.60	13%
z4	7	4	198	128	1.55	0.06	0.05	20%
Z5xp1	7	10	518	128	4.05	0.05	0.03	67%
Z9sym	9	1	512	512	1.00	0.11	0.10	10%
Ave.					3.43			81%

results run by the procedure which includes TEPACT and not TEPACT are listed. The fault coverages for each implemented circuit of the function covered by the UTSs generated by the procedure for two cases are all 100%. It can be seen that the average reduction ratio on test lengths of UTSs generated with TEPACT to those generated without TEPACT is 3.4. For some functions, the reduction ratio is even as high as 20. The increased CPU times to generate the UTS for each function with TEPACT is generally small. For these 56 functions, the average CPU time increase is 81%. Also, it is seen that due to the fast algorithm FUTS, the CPU times are very small for each function.

VI. CONCLUSION

In this paper, a fast universal test set (UTS) generation for multi-output functions is presented.

First, this paper presents a fast algorithm to generate the UTS for single-output functions. The algorithm generates UTS from the algebraic representation directly to obtain test_cube set instead of enumerating the expanded truth table to obtain test pattern set. This significantly reduces the CPU time and the requirement of temporary memory and memory storage for tests. Experimental results show that the algorithm achieves an improvement of 10^2–10^6 fold in the CPU time to compute

UTS and a saving of 1–1800 fold in memory storage to store UTS over the conventional method.

Then this paper presents a fast procedure to generate UTS for multi-output functions. It has been shown that the compaction technique used in the procedure achieved a reduction of 1–20 times in test lengths for the benchmark functions. For the most time-cost function: cps, which has 24 inputs and 109 outputs, only 18 s of sun4/SPARC2 CPU time is needed to obtain its UTS. From the experimental results, we could claim that the algorithms in this paper make the UTS generation practical for multi-output functions.

REFERENCES

[1] R. McNaughton, "Unate truth functions," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 1–6, Mar. 1961.

[2] S. B. Akers, "Universal test sets for logic networks," *IEEE Trans. Computers*, vol. C-22, pp. 835–839, Sept. 1973.

[3] S. M. Reddy, "Complete test sets for logic functions," *IEEE Trans. Computers*, vol. C-22, pp. 1016–1020, Nov. 1973.

[4] G. Gupta and N. K. Jha, "A universal test set for CMOS circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, pp. 590–597, May 1988.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Hingham, MA: Kluwer Academic, 1984.

[6] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. Develop.*, vol. 18, pp. 443–458, Sept. 1974.

[7] R. K. Brayton, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multilevel logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

Beyin Chen received the B. S. degree in electronics engineering in 1987 from the National Chiao Tung University, Taiwan, Republic of China, where presently she is working towards the Ph. D. degree.

Her research interests include VLSI testing and logic verification.

Chung Len Lee (S'70–M'75–SM'92) received the B. S. degree from National Taiwan University, Taiwan, Republic of China, and the M. S. and Ph. D. degrees from Carnegie-Mellon University, Pittsburgh, PA, all in electrical engineering, in 1968, 1971, and 1975, respectively.

He joined the Department of Electronics Engineering, National Chiao Tung University in 1975, where presently he is a Professor. His teaching and research have been in the areas of integrated circuits and testing. He has supervised more than 90 M. S. and Ph. D. students to complete their thesis and has published more than 140 papers in technical journals and conferences in the above areas.

Presently, Dr. Lee is a member of the editorial board of *Journal of Electronic Testing, Theory and Application* (Kluwer) and the IEEE Asian Test Technology Committee.