A Complement-Based Fast Algorithm to Generate Universal Test Sets for Combinational Function Blocks

Beyin Chen and Chung Len Lee
Department of Electronics Engineering and Institute of Electronics
National Chiao Tung University
Hsin-Chu, Taiwan, 30050

Abstract
In this paper, a fast algorithm to generate the universal test sets (UTS) for combinational function blocks is presented. The algorithm generates the UTS directly by Shannon-expanding and complementing the function, instead of the conventional truth table enumerating [3]. This significantly reduces the time complexity and the memory requirement. Experimental results show that this algorithm achieves an improvement of 2-6 orders of magnitude in computational efficiency over that of Ref[3]. This makes the UTS generation be practical for combinational function blocks.

1. Introduction
Presently, millions of transistors can be put into a single chip and this makes the circuit of IC become large and complex. As a result, the testing problem for IC becomes more complicated and difficult. Many ATPG systems have been proposed. However, since the variety of circuit technologies currently available can result in many different implementations of the same logical function, it would certainly be convenient to have a single test set suitable for all of these implementations. Also, it is very often for a user to encounter a manufacturer who does not wish to disclose his particular implementation. Therefore, a set of functional test patterns would be necessary and convenient.

Theoretically, for the functional testing, the complete test set for a combinational circuit is the whole truth table which constitutes the exhaustive test. Since this is not practical, it is hoped that a test set which has a quite good fault coverage but a smaller size can be found.

Uni-ate function theory was proposed in 1961 [1], and it was used to generate functional test patterns for combinational circuits [2,3,4]. It had been shown that for any "unate gate network", there is an equivalent "restricted gate network" for which a complete test set for any number of stuck-at faults, namely, the universal test set (UTS), can be generated and the generated UTS is independent of the circuit implementation [3,4]. However, the procedure to generate the UTS for a combinational function block involves a process to enumerate the truth table of the function, and it is computationally impractical. Also, even the computation can be speeded up, the size of UTS grows exponentially with the number of binary input variables. This makes the storage of test patterns be a problem.

This paper presents a fast algorithm, FUTS, to generate UTS for combinational functional blocks. The algorithm adopts a method to find UTS by Shannon-expanding and complementing the function, hence eliminates completely 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 computational efficiency and 1-1800 times in storage saving of test set over that of Ref[3]. Also, only 1 second of CPU time was spent for the largest benchmark function listed in Ref[5] to obtain its UTS and only 4K-byte memory is used to store the test cubes. Hence this algorithm makes the UTS generation be practical for combinational function blocks.

2. Denotations
In this section, some denotations which are needed to describe the algorithm FUTS are given.
A logic function F has n input variables x1, x2, ..., xn and is represented as the sum-of-products form: F = p1 + p2 + ... + pk. The product term p; is represented by a cube cj which is specified by the row vector cj = [c^1j, ..., c^n], where

\[ c^i_j = \begin{cases} 0 & \text{if } x_i \text{ appears complemented in } p_j; \\ 1 & \text{if } x_i \text{ appears not complemented in } p_j; \\ 2 & \text{if } x_i \text{ does not appear in } p_j; \end{cases} \]

for i = 1, ..., n.

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 [3,4]. It is denoted as:

\[ \text{UTS}(F) = \text{V}_{\text{mt}}(F) + \text{V}_{\text{mf}}(F). \]
An example as shown in Fig. 1 is to demonstrate the above. Fig. 1 (a) shows the sum-of-products form for the logic function F whose cover and mini_cover cubical representations are shown in Fig. 1 (b). Since variable \( a \) is negatively unate and variables \( b, c \) and \( d \) are binate, the expanded truth table of \( F \) is as shown in Fig. 1 (c). After comparing and deleting vertices from the table, \( V_{mT}(F) \) and \( V_{mF}(F) \) are obtained as shown in Fig. 1 (d) which constitutes the UTS of \( F \).

\[ F = ab\overline{cd} + \overline{b}c\overline{d} + \overline{a}c\overline{d} + \overline{a}bc \]

(a) Sum-of-products form of function \( F \).

\[
\begin{array}{ccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{F} \\
0 & 0 & 0 & 0 & 1 \downarrow \text{deleted} \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

(b) Cubical representations of \( F \).

\[
\begin{array}{ccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{F} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

(c) Expanded truth table of \( F \).

\[
\begin{array}{ccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{F} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

(d) Minimal true vertex set and maximal false vertex set of \( F \).

**Fig. 1 Example.**

In the Shannon expansion of the function \( F \), i.e. \( F = x_1 \cdot F_x + \overline{x}_1 \cdot F_{\overline{x}} \cdot F_x \cdot F_{\overline{x}} \) and \( F_x \) and \( F_{\overline{x}} \) are the one-cofactor and the zero-cofactor of \( F \) with respect to the splitting variable \( x_i \) respectively. A cofactor is **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 unate cofactors.

To save the storage space for UTS, the sets of cubes obtained from the algorithm FUTS to store \( V_{mT}(F) \) and \( V_{mF}(F) \) are said to be the **true test-cube** set and the **false test-cube** set respectively.

### 3. FUTS: Fast Universal Test Generation

To present the algorithm, some propositions and theorems are first presented.

**Proposition 1:** For an input of function \( F \), if it is positively (negatively) unate for \( F \), then it is negatively (positively) unate for \( F \)’s complement \( \overline{F} \).

Applying Proposition 1 to the definition of \( V_{mF}(F) \) and \( V_{mT}(\overline{F}) \), the following proposition holds.

**Proposition 2:** For a function \( F \), its maximal false vertex set is equal to the minimal true vertex set of \( \overline{F} \), i.e., \( V_{mF}(F) = V_{mT}(\overline{F}) \).

For an unate function \( F \), the following theorem can be used to generate its minimal true vertices.

**Theorem 1:** If \( F \) is an unate function (i.e. all its inputs are unate) and its minimized form \( \text{mini-cover}(F) \) has \( k \) cubes \( c_1, \ldots, c_k \), then \( V_{mT}(F) \) contains exactly \( k \) vectors \( v_1, \ldots, v_k \), and they can be obtained as follows:

- For \( j := 1 \) to \( k \)
  - For \( i := 1 \) to \( n \)
    - \( v_j := c_j \) if \( c_j \) is 1 or 0;
    - \( v_j := 0 \) if \( c_j \) is 2 and \( x_i \) is positively unate w.r.t. \( F \);
    - \( v_j := 1 \) if \( c_j \) is 2 and \( x_i \) is negatively unate w.r.t. \( F \);
- Where \( v_j \) is an entry of the vertex \( v_j = [v_1, \ldots, v_n] \);

**Proof:** Without loss of generality, let \( F = ab + cd \). Then its expanded truth table is

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{F} \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

The product term \( ab \) contributes four true vectors as shown in above dotted block. Among them, only the row vector \( [1100] \) has the chance to be minimal. Similarly, for product term \( cd \), only the row vector \( [0011] \) has the chance to be minimal. It can be seen that in a prime cube \( c_j \) of \( F \), there exists at most one vector \( v_j \) has the chance to be minimal for the whole expanded table. So, if \( F \) has \( k \) prime cubes, then there are at most \( k \) minimal true vertices, say \( v_1, v_2, \ldots, v_k \) and they just satisfy the assigning rules as described in this theorem. Next, we must prove that there are at least \( k \) vectors that are minimal, i.e. for all \( v_1, v_2, \ldots, v_k \), they do not dominate each other. To prove it,
the counter-proof technique is used. Suppose that, for a row vector \( v_j \), there exists another vector \( v_j \) that is dominated by \( v_i \). Since \( F \) is unate, the prime cube \( c_j \) which corresponds to \( v_j \) must contains the prime cube \( c_j \) which corresponds to \( v_i \). A contradiction arises. So, for \( v_1, v_2, \ldots, v_k \), they do not dominate each other. For example, \( v_1 = [1100] \) and \( v_2 = [1000] \), then \( c_1 = [11-] \) and \( c_2 = [1-1-] \). We can see that \( c_i \) is not a prime cube.

Q.E.D.

For a general function which has some of its inputs binate, the above theorem cannot be used. In this case, the following theorem can be used to simplify the problem.

**Theorem 2** :
Let \( x_i \) be a binate input variable of \( F \), then \( UTS(F) = x_i \cdot UTS(F_{x_i}) + \overline{x_i} \cdot UTS(F_{\overline{x_i}}) \ldots (1) \)

**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 divided the expanded truth table into two for \( x_i \) to be 0 and 1 respectively and then find their UTSs respectively. Based on this and the definition of Shannon expansion, this theorem holds.

Q.E.D.

With Theorem 2, the UTS of \( F \) can be obtained by recursively computing the UTS of the two smaller functions. In FUTS, the recursion continues until the cofactor of which the UTS is to be found is unate. Since in such cases, the minimal true vertices of \( F \) can be obtained directly from the cofactor’s prime cubes similar to the relationships described in Theorem 1. For each recursion, a binate variable of \( F \) is chosen to split the cofactor so that the cofactor becomes successively “more unate”. When the cofactor \( C \) becomes unate, \( \text{UTS}(C) \) and can be obtained to use its test_cubes.

**Lemma 1** : If \( C \) is tautology, then \( \text{UTS}(C) \) contains a single true test_cube and it can be obtained as follows : for \( (i := 1 \) to \( n) \)

\( \text{if} (x_i \text{ is not splited }) \), then

\( t_i = 1 \) if \( x_i \) is positively unate w.r.t. \( F \);

\( t_i = 0 \) if \( x_i \) is negatively unate w.r.t. \( F \);

\( t_i = 2 \) if \( x_i \) is binate w.r.t. \( F \);

where \( t_i \) is an entry of the true test_cube \( t = [t_1, ..., t_n] \).

**Proof** : Without loss of generality, consider a tautology cofactor \( C \) for which there are 4 unsplitted variables \( a, b, c, \) and \( d \) and suppose that \( a \) and \( b \) are binate, variable \( c \) is positively unate, and variable \( d \) is negatively unate with respect to \( F \). Then its expanded truth table is as shown in Fig.2. Since variables \( a \) and \( b \) are binate, the table can be divided into four sub-tables for \((a, b)\) to be \((0,0), (0,1), (1,0), \) and \((1,1)\) respectively and for each sub-table, the UTS can be found respectively as shown in Fig.2. \( \text{UTS}(C) \) is the union of the four vertices above and can be represented by a single test_cube \([abcd] = [-011] \). Q.E.D.

**Lemma 2** : If \( C \) is nil, then \( \text{UTS}(C) \) contains a single false test_cube and it can be obtained as follows : for \( (i := 1 \) to \( n) \)

\( \text{if} (x_i \text{ is not splited }) \), then

\( \beta_i = 1 \) if \( x_i \) is positively unate w.r.t. \( F \);

\( \beta_i = 0 \) if \( x_i \) is negatively unate w.r.t. \( F \);

\( \beta_i = 2 \) if \( x_i \) is binate w.r.t. \( F \);

where \( \beta_i \) is an entry of the false test_cube \( f = [f_1, ..., f_n] \).

**Proof** : With Theorem 2, the UTS of \( F \) can be obtained by recursively computing the UTS of the two smaller functions. In FUTS, the recursion continues until the cofactor of which the UTS is to be found is unate. Since in such cases, the minimal true vertices of \( F \) can be obtained directly from the cofactor’s prime cubes similar to the relationships described in Theorem 1. For each recursion, a binate variable of \( F \) is chosen to split the cofactor so that the cofactor becomes successively “more unate”. When the cofactor \( C \) becomes unate, \( \text{UTS}(C) \) and can be obtained as follows : for \( (j := 1 \) to \( k) \)

\( \text{if} (x_i \text{ is not splited }) \), then

\( \beta_j = c_j \) if \( c_j = 1 \) or \( 0 \);

\( \beta_j = 0 \) if \( c_j = 2 \) and \( x_i \) is positively unate w.r.t. \( F \);

\( \beta_j = 1 \) if \( c_j = 2 \) and \( x_i \) is negatively unate w.r.t. \( F \);

\( \beta_j = 2 \) if \( c_j = 2 \) and \( x_i \) is binate w.r.t. \( F \);

where \( \beta_j \) is an entry of the true test_cube \( t_j = [t_1, ..., t_n] \);

and for \( (j := 1 \) to \( m) \)

\( \text{if} (x_i \text{ is not splited }) \), then

\( \beta_j = c_j \) if \( c_j = 1 \) or \( 0 \);

\( \beta_j = 0 \) if \( c_j = 2 \) and \( x_i \) is positively unate w.r.t. \( F \);

\( \beta_j = 1 \) if \( c_j = 2 \) and \( x_i \) is negatively unate w.r.t. \( F \);

\( \beta_j = 2 \) if \( c_j = 2 \) and \( x_i \) is binate w.r.t. F, where \( \beta_j \) is an entry of the false test_cube \( f_j = [f_1, ..., f_n] \).

**Proof** : The first half of this lemma is the extension of Theorem 1 and can be proved by using Theorem 1 and Theorem 2. The second half of the lemma can be proved by using Proposition 1, Proposition 2 and the first half of this lemma. The detailed proof is omitted.

Q.E.D.
For example of Fig.1, the mini_cover(F) is shown in Fig.3 (a) in matrix form. In FUTS, the Shannon expansion for F to find its test_cubes can be demonstrated with a binary tree in Fig.3 (b), where T and N mean the cofactors which are tautology and nil respectively. After applying Lemmas 1, 2, and 3 to the cofactors 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_cubes of F are generated and are shown in Fig.3 (c). The superscripts in test_cubes mean that the cubes are true (false) test_cubes.

\[
F = \begin{bmatrix}
    a & b & c & d \\
    0 & 1 & 0 & 1 \\
    0 & 0 & 0 & 0 \\
    0 & 1 & 0 & 1 \\
    0 & 0 & 1 & 1 \\
\end{bmatrix}
\]

(a) cover(F) in matrix form.

```
F  
right_child : 1  
left_child : 0  
T N a b c d 
0 1 0 0 0 0 1 1 
(b) Binary recursive tree for test_cubes.
```

\[\{ [1000], [001], [0100], [0110], [1101], [001], [101], [0110], [1110], [0111] \}\]

(c) Test_cubes of F.

Fig.3 Example in Fig.1 is continued to demonstrate the algorithm FUTS.

As demonstrated with this example, the expanded truth table enumeration and the comparing operations are completely eliminated. In addition, using "test_cube" to store the UTS, 2 row vectors of memory are saved.

The details of the algorithm FUTS can be summarized as follows:

### Algorithm FUTS(F);
Input : cover(F).
Output : test_cubes.

{ tests := UTS_GEN (F); }

Procedure UTS_GEN(cofactor);

{ 
  if (cofactor == nil) then
    test_cubes := NIL_CASE();
  else if (there is a row of all 'a's in cover(cofactor))
    test_cubes := TAUTOLOGY_CASE();
  else if (cofactor is unate)
    test_cubes := UNATE_CASE(cofactor);
  else /* Theorem 2 */
    x_j := SPLIT_SELECT(cofactor);

/* A binate variable of F is selected */
onc := ONE_COF(cofactor, x_j);
zero_c := ZERO_COF(cofactor, x_j);
/* Compute one-cofactor and zero-cofactor */
test_cubes := MERGE (UTS_GEN(o), UTS_GEN(z));
/* The variable x_j is in test_cubes generated from */
UTS_GEN(o) is set to 1. */
/* The variable x_j in test_cubes generated from */
UTS_GEN(z) 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; /* The first half of Lemma 3 */
  comp_cover := UNATE_COMPLEMENT (mini_cover);
  Generate m false test_cubes corresponding to the m
  cubes in comp_cover; /* The second half of Lemma 3 */
  return (k true test_cubes + m false test_cubes);
}

Note that in the above Procedure UNATE_CASE( ),
UNATE_SIMPLIFY ( ) is to make an unate cofactor prime
and irredundant and UNATE_COMPLEMENT( ) is to
compute the complement of an unate cofactor. They are
simple and quite fast to be run [5]. In addition, if
UNATE_COMPLEMENT is used to complement a prime
and irredundant cover, e.g. mini_cover(C), then the cover
of C is prime and irredundant too. So, in the Procedure
UNATE_CASE, comp_cover is prime and irredundant and then Lemma 3 can be used.

In logic synthesizers, such as ESPRESSO [5], to
compute a function's complement, the Shannon expansion
is also used to simplify the complement problem. It is
continued recursively until the special cases as described
in FUTS appear. The similar equation as Equation (1) is
as follows.

\[\bar{F} = x_i \cdot \left(\overline{F_{x_i}}\right) + \overline{x_i} \cdot \left(\overline{F_{\overline{x_i}}}\right) \quad (2)\]

So, it is clear that the UTS generator involves a process
of complementing the function of which the UTS is to be
found and it can be implemented by modifying the above
complementer without too much computation overhead.

### 4. Experimental Results

The above algorithm and the method in Ref[3] 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, their outputs were
extracted as single-output functions. Table 1 gives the
results obtained to run 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 saving of memory space by 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: rcl, x6dn, in4, whose input numbers are larger than 30, for CUTS, more than $2^{30}$ bytes (= 1000 M bytes) of temporary memory are needed for truth table enumeration! Yet, for FUTS, the largest benchmark function, in4, only 1 second of CPU time was spent to obtain its UTS and 4K-byte memory was used to store its test-cubes.

To demonstrate the efficiency of the generated UTS, the functions in Table 1 are synthesized by the multi-level logic synthesizer misII, and then the fault coverages for these circuits are simulated for both the generated UTSs in Table 1 and the randomly generated patterns having the same number of patterns as the UTSs. The simulation results are shown in Table 2. 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 UTSs and the randomly generated patterns respectively. The fault coverages in both columns are obtained by dividing the number of detected faults by the number of detectable faults. We can see that the generated UTSs reach 100% fault coverages for all the circuits. But for random patterns, the fault coverages may be very low even to 3%.

For the algorithm CUTS to find UTS, it is the number of the operations 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 $O(2^n)$ or $O(2^n)$ for an n-input function. Yet, for the algorithm FUTS, since no comparing operation is involved, the time complexity is 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. 4 is plotted for the 18 functions of Table 1. 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 UTSs. The computation times to generate UTSs are plotted in terms of the numbers of binate inputs in Fig. 5 (a) and (b) for CUTS and FUTS respectively. In Fig. 5, 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. 5 (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.

5. Conclusions

This work presents a fast algorithm to generate UTS for combinational function blocks. The algorithm generates UTS from the algebraic representation directly instead of enumerating the expanded truth table. This significantly reduces the CPU time and memory requirement. Besides, the algorithm involves a process of complementing the function, which is exactly the same process used in logic minimization for logic synthesis [5]. This means that the algorithm can be incorporated into the logic synthesis process to generate UTS for the synthesized function block during the logic synthesis procedure without taking too much additional computation time. 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 space to store UTS over that of Ref[3]. For a function of 31 inputs, to obtain its UTS only 1 second of sun4/SPARC2 CPU time and 4K-byte memory were needed. From the experimental results, we could claim that the algorithm makes the UTS generation be practical for combinational function blocks.

References

Table 1  The run results on 18 benchmark functions [5] by FUTS and CUTS to generate UTS.

<table>
<thead>
<tr>
<th>Fun.</th>
<th>#in</th>
<th># of test cube</th>
<th>Size of UTS</th>
<th>Mem. Saving Ratios</th>
<th>Running Time (sec.)</th>
<th>Time Saving Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>x7dn</td>
<td>11</td>
<td>9</td>
<td>109</td>
<td>192</td>
<td>2</td>
<td>0.03</td>
</tr>
<tr>
<td>x1dn</td>
<td>14</td>
<td>3</td>
<td>66</td>
<td>74</td>
<td>1</td>
<td>0.03</td>
</tr>
<tr>
<td>x6dn</td>
<td>14</td>
<td>3</td>
<td>66</td>
<td>74</td>
<td>1</td>
<td>0.03</td>
</tr>
<tr>
<td>gary</td>
<td>11</td>
<td>10</td>
<td>51</td>
<td>1034</td>
<td>20</td>
<td>0.02</td>
</tr>
<tr>
<td>bcc</td>
<td>15</td>
<td>11</td>
<td>417</td>
<td>2317</td>
<td>6</td>
<td>0.23</td>
</tr>
<tr>
<td>in5</td>
<td>16</td>
<td>8</td>
<td>166</td>
<td>591</td>
<td>4</td>
<td>0.08</td>
</tr>
<tr>
<td>in1</td>
<td>14</td>
<td>12</td>
<td>81</td>
<td>4164</td>
<td>51</td>
<td>0.10</td>
</tr>
<tr>
<td>v21</td>
<td>18</td>
<td>10</td>
<td>588</td>
<td>3748</td>
<td>6</td>
<td>0.42</td>
</tr>
<tr>
<td>in0</td>
<td>15</td>
<td>13</td>
<td>153</td>
<td>8348</td>
<td>55</td>
<td>0.12</td>
</tr>
<tr>
<td>in2</td>
<td>18</td>
<td>8</td>
<td>81</td>
<td>732</td>
<td>9</td>
<td>0.05</td>
</tr>
<tr>
<td>bcb</td>
<td>15</td>
<td>13</td>
<td>178</td>
<td>8256</td>
<td>46</td>
<td>0.06</td>
</tr>
<tr>
<td>boc</td>
<td>18</td>
<td>15</td>
<td>247</td>
<td>46362</td>
<td>186</td>
<td>0.13</td>
</tr>
<tr>
<td>bca</td>
<td>15</td>
<td>2</td>
<td>41</td>
<td>41</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>chin</td>
<td>16</td>
<td>3</td>
<td>24</td>
<td>26</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>bcd</td>
<td>15</td>
<td>7</td>
<td>30</td>
<td>144</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>rckl</td>
<td>32</td>
<td>9</td>
<td>33</td>
<td>33</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>x6dn</td>
<td>34</td>
<td>13</td>
<td>321</td>
<td>24944</td>
<td>76</td>
<td>0.30</td>
</tr>
<tr>
<td>in4</td>
<td>31</td>
<td>20</td>
<td>11482</td>
<td>51000</td>
<td>1873</td>
<td>0.90</td>
</tr>
</tbody>
</table>

*: out of memory, &,: the ratio can’t be calculated.

Table 2  The fault simulation results for the generated UTS in Table 1 and the randomly generated patterns.

<table>
<thead>
<tr>
<th>Fun.</th>
<th>#gate</th>
<th>#fault</th>
<th>#pattern</th>
<th>Fault Coverages (UTS)</th>
<th>Fault Coverages (random)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcd</td>
<td>17</td>
<td>92</td>
<td>144</td>
<td>100.00% 3.26%</td>
<td></td>
</tr>
<tr>
<td>bca</td>
<td>15</td>
<td>68</td>
<td>41</td>
<td>100.00% 4.41%</td>
<td></td>
</tr>
<tr>
<td>x1dn</td>
<td>15</td>
<td>78</td>
<td>74</td>
<td>100.00% 6.41%</td>
<td></td>
</tr>
<tr>
<td>rckl</td>
<td>19</td>
<td>102</td>
<td>108</td>
<td>100.00% 19.61%</td>
<td></td>
</tr>
<tr>
<td>chin</td>
<td>15</td>
<td>78</td>
<td>26</td>
<td>100.00% 24.36%</td>
<td></td>
</tr>
<tr>
<td>x6dn</td>
<td>17</td>
<td>82</td>
<td>74</td>
<td>100.00% 30.49%</td>
<td></td>
</tr>
<tr>
<td>in5</td>
<td>45</td>
<td>226</td>
<td>591</td>
<td>100.00% 45.61%</td>
<td></td>
</tr>
<tr>
<td>bcc</td>
<td>175</td>
<td>906</td>
<td>2317</td>
<td>100.00% 62.47%</td>
<td></td>
</tr>
<tr>
<td>in2</td>
<td>27</td>
<td>138</td>
<td>732</td>
<td>100.00% 78.29%</td>
<td></td>
</tr>
<tr>
<td>bcb</td>
<td>65</td>
<td>346</td>
<td>8256</td>
<td>100.00% 78.48%</td>
<td></td>
</tr>
<tr>
<td>boc</td>
<td>120</td>
<td>600</td>
<td>46362</td>
<td>100.00% 98.00%</td>
<td></td>
</tr>
<tr>
<td>in0</td>
<td>77</td>
<td>(+4)364</td>
<td>8348</td>
<td>100.00% 98.63%</td>
<td></td>
</tr>
<tr>
<td>x7dn</td>
<td>24</td>
<td>104</td>
<td>182</td>
<td>100.00% 99.04%</td>
<td></td>
</tr>
<tr>
<td>gary</td>
<td>30</td>
<td>(+2)152</td>
<td>1034</td>
<td>100.00% 100.00%</td>
<td></td>
</tr>
<tr>
<td>v21</td>
<td>43</td>
<td>196</td>
<td>3748</td>
<td>100.00% 100.00%</td>
<td></td>
</tr>
<tr>
<td>in4</td>
<td>60</td>
<td>(+1)283</td>
<td>4164</td>
<td>100.00% 100.00%</td>
<td></td>
</tr>
<tr>
<td>x6dn</td>
<td>91</td>
<td>(+1)421</td>
<td>24944</td>
<td>100.00% 100.00%</td>
<td></td>
</tr>
<tr>
<td>Ave.</td>
<td></td>
<td></td>
<td></td>
<td>100.00% 61.77%</td>
<td></td>
</tr>
</tbody>
</table>

Fig 4  The plots of computation times with respect to the numbers of ”test_cubes” of 18 benchmark functions run by FUTS.

Fig 5  The plots of computation times with respect to the numbers of binate inputs for the cases : Input number = 15, 16, 17, 18, 19 obtained by (a) CUTS and (b) FUTS respectively.