高圧縮可能かつコンパクトなテスト生成について

市原　英行[†a)]　　井上　智生[†]

A Test Generation for Compressible and Compact Test Sets

Hideyuki ICHIHARA[†a)] and Tomoo INOUE[†]

あらまし　VLSI のテスト実行時のコストを削減するために，統計型符号を用いたテストデータ圧縮・展開法が提案されている．この手法では，与えられたテスト集合をあらかじめ圧縮しておき，テスト時に展開することを行う．文献 [10] では，テスト集合生成時に統計型符号による圧縮率が高まるようにテスト生成を行う手法が提案されており，高い圧縮率をもつテスト集合が生成できることが報告されている．本論文では，文献 [10] に動的テストコンパクションを併用することで，高い圧縮率を保ったまま，テストベクトル数の小さいテスト集合を生成可能な手法を提案する．ベンチマーク回路に対する実験結果は，提案手法が小さい計算時間で高圧縮かつコンパクトなテスト集合を生成でき，結果としてテストデータ量を小さくできることを示している．

キーワード　VLSI，テストデータ圧縮，統計型符号，テスト生成，テスト圧縮

1. まえがき

VLSI の大規模化・複雑化に伴うテストデータの増加は，テスタメモリの不足だけでなく，テスト実行時間の増加を引き起こす．この問題に対して，大きく分けて二つの手法が提案されている [1]．一つはテストベクトル数の削減を目的としたテストコンパクション法 (test compaction) [2]〜[4] であり，もう一つは，テスト集合をあらかじめ圧縮しておき，テスト実行時に展開するテストデータ圧縮・展開法 (test compression/decompression) [5]〜[14] である．

図 1 にテストデータ圧縮・展開法の概念図を示す．生成したテスト集合 T は，あらかじめテストデータ T_D に可逆圧縮されテスタメモリに蓄えられる．テスト時に，T_D は被テスト VLSI に転送され埋め込まれた展開器でもとのテスト集合 T に展開され，被テスト回路に与えられる．テスト集合を圧縮することで，テストデータを格納するテスタ上のメモリが節約できるだけでなく，テストデータをテスタから被テスト回路まで転送するために必要な時間も削減できる．

テストデータ圧縮・展開法では，テストデータを圧縮するためにデータ符号化手法が用いられる．本論文では，統計型符号を用いた手法に着目する．統計型符号を用いたテストデータ圧縮・展開法はこれまでにいくつか提案されており，コンマ符号を用いた手法 [5]，ハフマン符号を用いた手法 [5]〜[12]，ゴーロム符号を用いた手法 [13], [14] などが挙げられる．これらの手法では，テストベクトルを複数のブロックに分割し，それぞれのブロックを符号語に割り当てる．

統計型符号によるテストデータ圧縮・展開法の効果を高めるために，テスト生成の段階からデータ符号化を考慮したテスト生成法も提案されている [9]〜[11]．統計型符号を用いたテストデータ圧縮・展開法では，圧縮率はテスト集合中におけるブロックの出現頻度の分布によって決まる．一般にテストベクトルは 0 または 1 のどちらを割り当てても故障検出率に影響しない

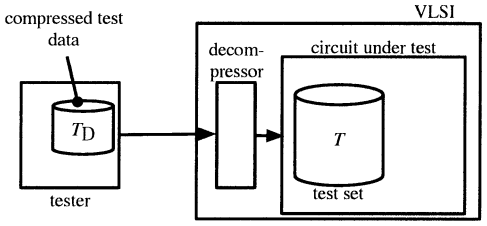

図 1　テストデータ圧縮・展開法
Fig. 1　Test compression/decompression scheme.

† 広島市立大学情報科学部，広島市
Faculty of Information Sciences, Hiroshima City University, Hiroshima-shi, 731–3194 Japan
a) E-mail: ichihara@im.hiroshima-cu.ac.jp

ビット（ドントケアビット）を含むため，これらの手法ではドントケアビットを利用して，圧縮率が高まるようにテスト集合中のブロックの出現頻度を決定する．

文献 [9] では，与えられたテスト集合を変換することにより，テスト集合中のブロックの出現頻度を変化させ，圧縮率を向上させるための手法を提案している．ただし，故障検出率が減少しないことを保証するため，故障シミュレーションを用いている．文献 [11] では，故障シミュレーションに加えてテスト生成アルゴリズムの一部の操作を用いて，与えられたテスト集合中のドントケアビットを判定した後，判定したドンケアビットを 0 または 1 に決定することで，テスト集合を変換している．これらの手法では，故障シミュレーションやテスト生成アルゴリズムの一部の操作を用いているため，比較的大きな処理時間が必要となるものの，変換されたテスト集合は，テストベクトル数はそのままで，高い圧縮率をもつ．

文献 [10] では，文献 [9], [11] のように与えられたテスト集合を変換するのではなく，テスト集合生成時に統計型符号での圧縮率が高まるようにテスト生成を行う手法を提案している．この手法では，一定数のドントケアビットを含むテストベクトルを生成した後，そのテストベクトル中のブロックの出現頻度を調べ，その出現頻度に基づき，一定数のドントケアビットを 0 または 1 に決定する．この操作を，複数のテストベクトルが生成されるごとに繰り返すことで，高圧縮可能なテスト集合を生成する．この手法はテスト生成中に動的にドントケアビットを決定し，ブロックを符号語に割り当てるため，本論文ではこの手法を動的高圧縮可能テスト生成と呼ぶことにする．これに対して，文献 [9], [11] は与えられたテスト集合を静的に変換し符号語を割り当てるため，これらの手法を静的高圧縮可能テスト生成と呼ぶ．

文献 [10] に示された実験結果によれば，動的高圧縮可能テスト生成法は高い圧縮率を示すテスト集合を生成できる．一方で，テスト集合に含まれるテストベクトル数については考慮していないため，テストベクトル数は増加する傾向がある．つまり，この手法によって生成されたテスト集合は，テストベクトル数は大きいものの，高圧縮率のためテストデータ量を小さくできることになる．そこで，圧縮率を高く保ったままテストベクトル数を減らすことができれば，更なるテストデータ量の削減が期待できる．

本論文では，高い圧縮率をもち，かつ，テストベクトル数の小さいテスト集合を生成するためのテスト生成法を提案する．テストベクトル数を低く抑えるために，動的テストコンパクション [2]〜[4] と文献 [10] の動的高圧縮可能テスト生成法の両方を用いる．提案手法では，まず動的テストコンパクションを用いて一つのテストベクトルで多くの故障を検出できるテストベクトルを生成した後，動的高圧縮可能テスト生成に移行して，今度はなるべく圧縮率が高くなるようにドントケアビットを 0 または 1 に決定する．結果として，高い圧縮率をもち，かつ，テストベクトル数の小さいテスト集合が生成できることが期待でき，結果的にテストデータ量を小さくできる．計算機実験では動的テストコンパクションと動的高圧縮可能テスト生成を併用することの効果について調べる．

本論文の構成を以下に示す．**2.** では統計型符号を用いたテスト集合の圧縮手法を簡単に紹介する．**3.** では文献 [10] の動的高圧縮可能テスト生成法について述べ，その問題点について述べる．**4.** では動的テストコンパクションを併用した動的高圧縮可能テスト生成法を提案する．**5.** では提案手法に対する実験結果について述べ，**6.** でまとめを述べる．

2. 統計型符号を用いたテストデータ圧縮・展開法

統計型符号を用いたテストデータ圧縮・展開法では，与えられたテスト集合を符号化するにあたり，テストベクトルを複数のブロックに分割することを行う．図 2 では，それぞれのテストベクトルを 4 ビットのブロックに分割している．テストベクトルを分割することで，展開器の構成を比較的簡単にでき，展開に要する遅れを軽減することができる．ブロックサイズは固定長の場合と可変長の場合がある．例えば，文献 [8] では，多重スキャンチェインの本数をブロックのサイズ（固定長）として与えており，展開後の符号語がそのまま多

t_1:	1111 0101 0011 1111 1011 1110 1101 1011
t_2:	1111 1111 1111 1111 0000 0000 0000 0000
t_3:	1001 0010 0110 0111 1110 1101 0110 1110
t_4:	1111 1111 1110 0110 1111 1001 1111 1011
t_5:	1111 1111 0111 1010 1111 1111 0111 1111
t_6:	0010 1110 1000 0100 1111 1111 1111 1111
t_7:	0110 1011 1011 1011 1101 0111 1011 0111
t_8:	1100 1000 1010 0111 0101 1011 1111 1101
t_9:	1011 0100 1101 1101 1110 1111 1111 1111
t_{10}:	1111 1011 0111 0101 1111 0111 1111 1101
t_{11}:	1100 1101 1001 1110 1110 1101 1011 0110
t_{12}:	0111 0010 1111 1111 0111 1111 1011 1111

図 2　テスト集合（4 ビットブロック化後）
Fig. 2　Test set. (partitioned into 4-bit blocks)

表 1　ブロックの出現頻度とハフマン符号語
Table 1　Block frequency and Huffman codeword.

i	x_i	p_i	Huffman codeword
1	1111	0.3125	11
2	1011	0.1250	100
3	0111	0.1042	010
4	1101	0.0938	000
5	1110	0.0833	1011
6	0110	0.0521	0011
7	0000	0.0417	10101
8	0010	0.0313	01111
9	1001	0.0313	01110
10	0101	0.0313	01101
11	1100	0.0208	01100
12	0100	0.0208	00101
13	1000	0.0208	00100
14	1010	0.0208	101001
15	0011	0.0104	101000
ave. codeword length			3.3125

重スキャン用のスキャンベクトルとなっている．それぞれのブロックに割り当てる符号語は可変長で，符号語の長さはそれぞれのブロックの出現頻度によって決定される．つまり，テスト集合中で出現頻度の高いブロックは短い符号語を，出現頻度の低いブロックは長い符号語を割り当てる．表 1 は，図 2 のテスト集合中に出現するブロック x_i の出現頻度 p_i と，統計型符号の一つの例として，その出現頻度から求めたハフマン符号語を示している．

3. 動的高圧縮可能テスト生成法 [10]

高圧縮可能テスト生成法 [10] は，統計型符号で高圧縮できるテスト集合を生成することを目的としたテスト生成法である．

図 3 にこのテスト生成アルゴリズム HCTG の概要を示す．アルゴリズムは被テスト回路の故障リスト F，ブロックサイズ n，二つのパラメータ N_x と N_r を受け取る．$T = \phi$ の初期化後，故障リスト F から故障 f を選択する（図 3 の 8 行目）．この故障 f に対して，テストパターン t を生成し，テスト集合 T に追加する（9 行目）．ここで，テストパターン t 中のドントケアビットはそのままにしておく．次に，テスト集合 T 中のテストベクトルを n ビットブロックに分割する（10 行目）．このとき，ドントケアビットを含んでいるブロックを X ブロックと呼ぶ．もし，X ブロックの数が与えたパラメータ N_x よりも大きい場合（11 行目），Specify_X_block(T, N_r) を呼び，テスト集合 T 中の X ブロックの数が N_r 以下になるまで，テスト集合 T 中のドントケアビットを 0 または 1 に決定する．特に $N_r = 0$ のときは，テスト集合 T 中に X ブロックがなくなるまでトントケアビットの割当を行う．このとき Specify_X_block(T, N_r) は，テスト集合 T 中のブロックの出現頻度の分布に従い，テスト集合 T の圧縮率が上がるように 0 または 1 の決定を行う（12 行目）．この後，テスト集合 T 中のドントケアビットを含まないテストベクトルに対して故障シミュレーションを行い，検出された故障を故障リストから削除する（13 行目）．これらの処理を故障リストが空になるまで繰り返す（7 行目）．

```
1  HCTG(F, n, N_x, N_r)
2     F: fault list;
3     n: block length;
4     N_x, N_r: number of X blocks;
5  {
6     Set test set T = φ;
7     while (F is not empty){
8        Select a target fault f from F;
9        Generate test vector t for f and add t into T;
10       Divide test vectors in test set T into n-bit blocks;
11       if(the number of X blocks in T is over N_x)
12          Specify_X_block(T, N_r);
13       Perform fault simulation with completely specified
         test vectors in T, and drop the detected faults from
         F;
14    }
15    return T;
16 }
```

図 3　動的高圧縮可能テスト生成アルゴリズム (HCTG)
Fig. 3　Highly compressible test generation algorithm. (HCTG)

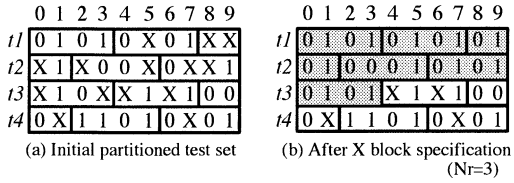

図 4　動的高圧縮可能テスト生成の例
Fig. 4　Example of highly compressible test generation.

表 2　図 4 (a) におけるブロックの出現頻度
Table 2　Block frequency in Fig. 4 (a).

block	frequency
0101	7
0001	6
1101	4
...	...

図 4 と表 2 の例を用いて，このアルゴリズムを説明する．今 HCTG を $n = 4$，$N_x = 7$，$N_r = 3$ として実行したと仮定する．最初は，テスト集合中の X ブロックの数がパラメータ N_x の値より大きくなるまでテスト生成が繰り返される．例では，これにより四つ

のテストベクトル t_1，t_2，t_3，t_4 が生成されたとする．図 4 (a) にブロック化したテストベクトルを示す．四つのテストベクトルにおける X ブロックの数は 8 となり，与えたパラメータ $N_x = 7$ の値よりも大きい．次に，Specify_X_Block が呼ばれる．Specify_X_Block では，各ブロックの出現頻度を数え上げ表 2 のような出現頻度表を作る．なお，出現頻度はそれぞれのブロックの出現頻度により降順でソートされている．また，一つの X ブロックはその X ブロックのドントケアビットを 0 または 1 に置き換えて生じるすべての非 X ブロックとして，数え上げている．例えば，X ブロック X00X は四つの非 X ブロック 0000，0001，1000，1001 として数えている．この出現頻度表に従って，X ブロックの数がパラメータ N_r と等しいか小さくなるようにテスト集合中のドントケアビットを 0 または 1 に決定する．この場合は N_r は 3 であるため，図 4 (b) のようにドントケアビットが決定される．なお，X ブロックの決定はそれぞれのテストベクトルが生成された順に行われており，この例ではベクトル t_1 の 0X01，ベクトル t_1 と t_2 の XXX1，ベクトル t_2 の X00X... という順になる．四つの X ブロック 0X01，XXX1，0XX1，X10X が，最も出現頻度の高いブロック 0101 に置き換えられ，一つの X ブロック X00X が 2 番目に出現頻度が高い 0001 に置き換えられる．これにより，二つのテストベクトル t_1 と t_2 が完全に決定されるため，この二つのベクトルに対して故障シミュレーションを行う．以上の操作を，故障リストが空になるまで繰り返す．

パラメータ N_x は，ブロックの出現傾向を予測するために使用する X ブロックの個数を決定する．一方，パラメータ N_r は，その予測により一度に割り当てる X ブロックの数を決定する．文献 [10] では，実用的な観点から見ると，これらのパラメータの値を小さく設定する方が圧縮率の高いテスト集合が得られ，最終的なテストデータ量を削減できることが報告されている．

4. 高圧縮可能かつコンパクトなテスト生成法

動的高圧縮可能テスト生成法 [10] は，テスト集合の圧縮率の向上だけを考えているため，テストベクトル数が増加する傾向にある．そこで，文献 [10] で提案した動的高圧縮可能テスト生成法に，テストベクトル数を削減するための動的テストコンパクション [2] を組み合わせた手法を提案する．まず，動的テストコンパクションについて説明した後，提案手法を述べる．

```
1  dycomp(F)
2     F: fault list;
3  {
4     while (F is not empty){
5        Select a target fault f from F;
6        Generate test vector t for f and add t into T;
7        while(not-yet-selected faults except f remain in
         F){
8           Select a secondary target fault g from F;
9           Try to extend test vector t to detect g;
10       }
11       Drop detected faults from F;
12    }
13    return T;
14 }
```

図 5　動的テストコンパクション (dycomp)
Fig. 5　Dynamic test compaction. (dycomp)

4.1 動的テストコンパクション

動的テストコンパクション [2]〜[4] は，テスト生成アルゴリズム中に行われる手法で，一つのテストベクトルで検出できる故障数を増やし，結果として生成されるテストベクトル数を削減することを目的としている．図 5 は動的テストコンパクションの基本アルゴリズムを示している．このアルゴリズムでは，選択した対象故障 f のテストベクトル t を生成した後（図 5 の 6 行目），二つ目の対象故障 g を選んで（8 行目），一つ目の対象故障 f に対して生成したテストベクトルで二つ目の対象故障 g を検出できるように，テストベクトル中のドントケアビットを 0 または 1 に決定する（9 行目）．これを最初の対象故障以外のすべての故障に対して行う（7 行目）．

4.2 高圧縮可能かつコンパクトなテスト生成法

動的高圧縮可能テスト生成法と動的テストコンパクションを組み合わせる場合，それぞれの手法がテストベクトル中のドントケアビットを異なる目的で利用していることになるため，その組合せ方法に工夫が必要となる．つまり，高圧縮可能テスト生成法は圧縮率を向上させることが目的であり，テストコンパクションはテストベクトル数を削減することが目的であるため，例えば，テストコンパクションがドントケアビットを利用する割合を増やせば，高圧縮可能テスト生成が利用できるドントケアビットの割合が減ってしまうことになる（これは，図 4 (a) において，1 ベクトル当りのドントケアビット数が減ることに相当する）．

提案手法ではこれら二つの手法がドントケアビットを利用する割合を決めるため，パラメータ N_d を導入する．図 6 に，提案手法のアルゴリズムを示す．この

```
1  HCTG+dycomp(F, n, N_d, N_x, N_r)
2     F: fault list;
3     n: block length;
4     N_d, N_x, N_r: number of X blocks;
5  {
6     Set test set T = φ;
7     while (F is not empty){
8        Select a target fault f from F;
9        Generate test vector t for f and add t into T;
10       Divide test vectors in test set T into n-bit blocks;
11       while(the number of X blocks in T is over N_d){
12          Select a secondary target fault g from F;
13          Try to extend test vector t to detect g;
14       }
15       if(the number of X blocks in T is over N_x)
16          Specify_X_block(T, N_r);
17       Perform fault simulation with test vectors completely specified in T;
18       Drop detected faults from F;
19    }
20    return T;
21 }
```

図 6　高圧縮かつコンパクトなテスト生成アルゴリズム (HCTG+dycomp)

Fig. 6　Compressible and compact test generation algorithm. (HCTG+dycomp)

図において 11 から 14 行目までが動的テストコンパクションに相当し，15，16 行目が動的高圧縮可能テスト生成に相当する．テストコンパクションから高圧縮可能テスト生成に移行する条件はパラメータ N_d によって決まり，生成したテストベクトル中の X ブロックの数が N_d より大きい間は，テストコンパクションを実行し，生成したテストベクトル中の X ブロックの数が N_d より小さくなれば，高圧縮可能テスト生成に移る（図 6 の 11 行目）．

パラメータ N_d を制御することで，二つの異なる目的をもった手法のトレードオフを制御できる．例えば，N_d を大きくすれば高圧縮可能テスト生成で利用するドントケアビットの割合が大きくなるため，圧縮しやすいテスト集合が生成されるが，テストコンパクションに使用するドントケアビットの割合が減るため，テスト数は増加すると考えられる．圧縮後のテストデータ量は（テストベクトル数）×（テストベクトル長）×（圧縮率）で求まるため，圧縮後のテストデータ量を減らすためには，テストベクトル数と圧縮率のどちらを優先して小さくすべきなのかは，テストベクトル数と圧縮率がどのようなトレードオフの関係にあるのかで決定する．次章では実験的に複数の N_d に対して提案手法を試みることで，適切な N_d について考察を行う．

5. 実験結果

提案手法を C 言語で実装し，ISCAS'85 及び ISCAS'89 の組合せ回路部に対する実験を行った．用いた計算機はワークステーション Sun Ultra 10 (UltraSPARC-IIi，440 MHz) である．テスト生成に用いた基本となるアルゴリズムは SOCRATES [15] であり，動的テストコンパクションとして maximum compaction [4] を用いた．また，ブロックサイズは 8 ビットとし，パラメータはそれぞれ $N_d = \{0, 10, 30, 50, 70, 100\}$，$N_x = \{0, 10, 50, 100\}$，$N_r = 0$ のすべての組合せに対して行った．なお，各パラメータの値の設定範囲は文献 [10] の結果に基づいている．文献 [10] では，パラメータ N_x を 10 から 2000 までの範囲で変化させて実験を行っており，ほとんどのベンチマーク回路に対して 100 以下の場合が最も高い圧縮率を示していることが報告されている．そのため，本実験は N_x の範囲を 100 以下とした．またパラメータ N_d の範囲も N_x と同じく 0 から 100 までの範囲とした．一方，パラメータ N_x の値は文献 [10] では 1 から 1500 までの範囲で実験を行っており，その結果，ほとんどのベンチマーク回路で $N_r = 1$ のときが最も高い圧縮率を示すことが報告されている．そのため，なるべく N_r は小さい方がよいという判断から，今回の実験では $N_r = 0$ として実験を行った．

表 3 にテスト生成の結果を示す．回路名と外部入力数に続いて，三つの手法 (*HCTG*，*dycomp*，*HCTG+dycomp*) によるテスト生成結果である，テストベクトル数，圧縮後のテストデータ量（ビット数），圧縮率，そしてテスト生成時間（秒）を示している．圧縮率は（圧縮後のテストデータ量)/((外部入力数）×（テストパターン数））で求めた．*HCTG* は図 3 に示した文献 [10] で提案している手法の結果であり，*dycomp* は図 5 の動的テストコンパクションのみを用いて，テスト生成を行った結果である．また，*HCTG+dycomp* は図 6 に示した提案手法の結果を示している．*HCTG* と *HCTG+dycomp* の結果は，N_d と N_x のすべての組合せに対して実験を行い，最も圧縮後のテストデータ量が少ないものを示している．そのときの N_d と N_x の値は，表の最後の 2 列に示す．

表 3 より，提案手法である *dycomp+HCTG* は，c499 と c1355 を除くすべての回路において，圧縮後のテストデータ量を他の手法よりも小さくできることが分かる．最後の行に示した平均値による比較を行うと，*dycomp+HCTG* で得られたテストベクトルの数は，*dycomp* で得られたテストベクトル数と同程度に少なく，*dycomp+HCTG* の圧縮率は *HCTG* よりも

表 3　テスト生成結果
Table 3　Results of test generation.

		HCTG [10]				*dycomp*				*HCTG+dycomp*					
回路	外部入力数	テスト数	テストデータ量	圧縮率	時間	テスト数	テストデータ量	圧縮率	時間	テスト数	テストデータ量	圧縮率	時間	N_d	N_x
c432	36	78	1088	0.38	0.10	48	1530	0.88	0.12	48	1087	0.63	0.14	0	0
c499	41	81	2115	0.63	0.16	57	2029	0.86	0.15	61	2174	0.87	0.17	0	50
c880	60	107	2124	0.33	0.22	33	1734	0.87	0.15	31	1198	0.64	0.30	0	0
c1355	41	134	3653	0.66	1.73	90	3394	0.91	0.88	91	3387	0.91	2.19	10	100
c1908	33	194	5286	0.82	0.53	132	4170	0.95	0.73	134	3851	0.87	0.88	0	10
c2670	233	199	10200	0.21	1.55	63	11643	0.79	1.15	66	5757	0.37	1.95	0	10
c3540	50	273	5931	0.43	1.95	126	6122	0.97	2.58	127	3980	0.63	4.12	0	10
c5315	178	283	11076	0.21	6.17	78	13755	0.99	2.53	75	7030	0.53	3.36	10	50
c6288	32	130	2510	0.60	2.01	34	909	0.83	84.51	35	908	0.81	4.82	0	0
c7552	207	420	22935	0.26	22.89	99	20251	0.98	10.23	95	12165	0.62	13.38	10	100
s9234	247	672	33445	0.20	28.63	138	33573	0.98	11.47	138	14428	0.42	14.16	10	0
s13207	700	1084	108101	0.14	30.95	272	183358	0.96	16.70	254	34244	0.19	25.06	70	100
s15850	611	835	79154	0.15	30.86	135	80709	0.97	22.29	132	21990	0.27	28.41	0	0
s35932	1763	574	129385	0.12	83.26	18	27994	0.88	181.38	18	11443	0.36	207.28	0	0
s38417	1664	3868	881040	0.13	409.17	102	165138	0.97	185.63	98	65777	0.40	221.50	30	50
s38584	1464	4737	916349	0.13	760.50	128	187190	0.99	274.92	125	62301	0.34	315.96	10	50
平均	460	854.31	138399.5	0.34	86.29	97.06	46468.68	0.92	49.71	95.5	15732.5	0.55	52.73		

少し大きくなっている．また，提案手法がテスト生成に要した時間は，動的テストコンパクションを用いた *dycomp* と同程度であった．

表 3 の最後の 2 列に示したように，提案手法において圧縮後のテストデータ量が最小となるときのパラメータ N_d の値は，16 の回路のうち九つの回路で $N_d = 0$，五つの回路で $N_d = 10$ であった．このように小さい N_d が効果的であるということは，生成されたテストベクトルに対してテストコンパクションを完全に（$N_d = 0$ のとき）若しくはほぼ完全に（$N_d = 10$ のとき）行った後に，高圧縮可能テスト生成を行うことが効果的であることを意味している．また s13207 や s38417 のように，N_d の値が大きくなる場合は，N_x も大きくなる傾向があることが分かる．

表 4 に各パラメータの値を $N_d = N_x = N_r = 0$ とした場合のテストベクトル数，圧縮後のテストデータ量，圧縮率を示す．また，最後の列には，表 3 の 12 列目に示した最小の圧縮後のテストデータ量に対する，$N_d = N_x = N_r = 0$ 時の圧縮後のテストデータ量（表 4 の 3 行目）の増加率を示す．つまり，回路 c432 や c880 では，$N_d = N_x = 0$ の場合に圧縮後のテストデータ量が最小となるため，増加率は 0%となる．この表から，$N_d = N_x = N_r = 0$ の場合の圧縮後のテストデータ量は，最小の圧縮後のテストデータ量に比べて平均で 1.71%程度の増加となっており，パラメータを $N_d = N_x = N_r = 0$ と固定しても，十分に高圧縮可能かつコンパクトなテスト集合が生成できることが分かる．これは，実用上は圧縮後のテストデータ量を最小にするためのパラメータを探索する必要はなく，このように比較的小さなパラメータ値を与えればよいことを意味する．

表 4　$N_d = N_x = N_r = 0$ の場合の結果
Table 4　Results for $N_d = N_x = N_r = 0$.

回路	テスト数	テストデータ量	圧縮率	増加率 (%)
c432	48	1087	0.63	0
c499	64	2281	0.87	4.92
c880	31	1198	0.64	0
c1355	94	3398	0.88	0.32
c1908	140	4010	0.87	4.13
c2670	66	5760	0.37	0.05
c3540	127	3999	0.63	0.48
c5315	77	7267	0.53	3.37
c6288	35	908	0.81	0
c7552	100	12544	0.61	3.12
s9234	147	15191	0.42	5.29
s13207	263	35026	0.19	2.28
s15850	132	21990	0.27	0
s35932	18	11443	0.36	0
s38417	98	66269	0.41	0.75
s38584	130	63981	0.34	2.70
平均	98.13	16022	0.55	1.71

表 5 に提案手法と文献 [11] の静的高圧縮可能テスト生成法の比較を示す．表から，テスト集合の圧縮率はほぼ同じであるが，文献 [11] の方がテストベクトル数が 2〜3 割少ないため，圧縮後のテストデータ量も 2〜3 割少ないことが分かる．しかしながらテスト生成に要する時間は，表 3 に示したように提案手法が動的テストコンパクションだけを用いたテスト生成とほぼ

表 5　文献 [11] との比較
Table 5　Comparison with [11].

	HCTG+dycomp			文献 [11]		
回路	テスト数	テストデータ量	圧縮率	テスト数	テストデータ量	圧縮率
s9234	138	14428	0.42	111	11022	0.40
s13207	254	34244	0.19	235	31271	0.19
s15850	132	21990	0.27	97	16685	0.28
s35932	18	11443	0.36	12	4638	0.22
s38417	98	65777	0.40	87	48818	0.34
s38584	125	62301	0.34	114	50597	0.30
平均	127.5	35030.5	0.33	109.33	27171.83	0.29

同じであるのに対して，文献 [11] の手法では同様なテスト生成時間に加えて，そのテスト生成時間の数倍の処理時間（ドントケアビットを決定する時間）が必要になっている．このことから，提案手法は比較的短い計算時間で効果的なテスト集合を生成することができるため，効率的であると考えられる．

6. むすび

本論文では，動的テストコンパクションと動的高圧縮可能テスト生成法を用いた統計型符号化のためのテスト生成法を提案した．実験結果から，従来のテスト生成法と同程度の計算時間で，テストベクトル数が少なく，高圧縮可能なテスト集合が生成でき，結果としてテストデータ量を小さくできることが分かった．また，動的テストコンパクションを十分に行った後に，動的高圧縮可能テスト生成を行うことが，最も効果的であることも分かった．今後は，静的高圧縮可能テスト生成法 [9], [11] と提案した動的手法を組み合わせた方法についても考察が必要であると考えている．

謝辞　テスト生成プログラムを提供して頂いた九州工業大学の梶原誠司教授に深く感謝致します．本研究は一部，日本学術振興会の科学研究費補助金（課題番号 1530021）及び栢森情報科学振興財団研究助成金（交付番号 K14 研 VII 第 142 号）の助成を得て行った．

文　献

[1] 樋上喜信，梶原誠司，市原英行，高松雄三，“論理回路に対するテストコスト削減法—テストデータ量および実行時間の削減，” 信学論（D-I），vol.J87-D-I, no.3, pp.291–307, March 2004.

[2] P. Goel and B.C. Rosales, “Test generation and dynamic compaction of tests,” Dig. Papers 1979 Test Conf., pp.189–192, 1979.

[3] I. Pomeranz, L.N. Reddy, and S.M. Reddy, “COMPACTEST: A method to generate compact test sets for combinational circuits,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.12, no.7, pp.1040–1048, 1993.

[4] S. Kajihara, I. Pomeranz, K. Kinoshita, and S.M. Reddy, “Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.14, no.12, pp.1496–1503, 1995.

[5] V. Iyengar, K. Chakrabarty, and B.T. Murray, “Built-in self testing of sequential circuits using precomputed test sets,” Proc. VLSI Test Symposium, pp.418–423, 1998.

[6] A. Jas, J. Ghosh-Dastidar, and N.A. Touba, “Scan vector compression/decompression using statistical coding,” Proc. VLSI Test Symposium, pp.114–120, 1999.

[7] P.T. Gonciari, B.M. Al-Hashimi, and N. Nicolici, “Improving compression ratio, area overhead, and test application time for system-on-a-chip test data compression/decompression,” Proc. DATE, pp.604–611, 2002.

[8] E.H. Volkerink, A. Khoche, and S. Mitra, “Packet-based input test data compression techniques,” Proc. ITC, pp.154–163, 2002.

[9] H. Ichihara, K. Kinoshita, I. Pomeranz, and S.M. Reddy, “Test transformation to improve compression by statistical encoding,” Proc. VLSI Design, pp.294–299, 2000.

[10] H. Ichihara, A. Ogawa, T. Inoue, and A. Tamura, “Test generation for test compression based on statistical coding,” IEICE Trans. Inf. & Syst., vol.E85-D, no.10, pp.1466–1473, Oct. 2002.

[11] S. Kajihara, K. Taniguchi, K. Miyase, I. Pomeranz, and S.M. Reddy, “Don’t care identification and statistical encoding for test data compression,” IEICE Trans. Inf. & Syst., vol.E87-D, no.3, pp.544–550, March 2004.

[12] H. Ichihara, M. Shintani, and T. Inoue, “A test decompression scheme for variable-length coding,” Proc. ATS, pp.426–431, 2004.

[13] A. Chandra and K. Chakrabarty, “Test data compression for system-on-a-chip using golomb codes,” Proc. VTS, pp.113–120, 2000.

[14] A. Chandra and K. Chakrabarty, “Frequency-directed run-length (FDR) codes with application to system-on-a-chip test data compression,” Proc. VTS, pp.42–47, 2001.

[15] M.H. Schulz, E. Trischler, and T.M. Sarfert, “SOCRATES: A highly efficient automatic test pattern generation system,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.7, no.1, pp.126–137, 1988.

（平成 16 年 8 月 30 日受付，12 月 14 日再受付）

市原　英行　（正員）

平 7 阪大・工・応用物理卒．平 9 同大大学院工学研究科応用物理学専攻博士前期課程了．平 11 同専攻博士後期課程了，博士（工学）．同年広島市立大学情報科学部情報機械システム工学科助手．平 16 より同大助教授．VLSI 回路のテスト生成，テストデータ圧縮，テスト容易化設計などの研究に従事．IEEE 会員．

井上　智生　（正員）

昭 63 明大・工・電子通信卒．平 2 同大大学院博士前期課程了．同年松下電器産業（株）入社．明大大学院博士後期課程を経て，平 5 奈良先端大情報科学研究科助手．平 11 広島市立大情報科学部助教授．平 16 より同大同学部教授．松下電器産業（株）において，マイクロプロセッサの研究開発に従事．明大，奈良先端大，広島市立大において，テスト生成，並列処理，テスト容易化設計/合成，再構成可能デバイスに関する研究に従事．博士（工学）．IEEE，情報処理学会各会員．