# 大規模組合せ回路のためのコンパクトなIDDQテスト集合の並列生成システム

篠木 剛[†] 林 照峯[†]

本論文は，大規模回路をターゲットにした，ブリッジ故障を IDDQ テスト法によって検出するための，コンパクトなテストパターン集合を生成する高性能なシステムについて述べる．本システムは，筆者らがすでに提案している逐次改善法（Shinogi, et al., 1998）をベースにしており，高速化のために 2 レベルの並列処理を用い，また 100%故障検出効率を達成するために deterministic ATPG を一部に利用する．評価実験結果により有効性を示す．

# A Parallel Generation System of Compact IDDQ Test Sets for Large Combinational Circuits

Tsuyoshi Shinogi[†] and Terumine Hayashi[†]

This paper presents a high performance compact IDDQ test generation system for detecting bridging faults, targeting large circuits. This system is based on the iterative-improvement-based method (Shinogi, et al., 1998). We use two-level parallel processing technique for speeding up the test generation significantly, and invoke the assist of a deterministic ATPG for attaining 100% fault efficiency. The experimental results demonatrate the effectiveness.

## 1. はじめに

IDDQ テスト法[1]は，CMOS の特徴である信号値の静止時にはほとんど電流が流れないことを利用し，短絡系の故障を，静止時の電源電流を測定すること（IDDQ 測定）により検出するテスト手法である．IDDQ テスト法は，論理テスト法では検出困難な故障も検出できることや，故障あたりのテスト生成計算コストが小さいことなどの特長を持つ．しかし，IDDQ 測定に時間がかかるため，LSI の検査速度が遅いという問題がある．このため，IDDQ テスト用のテストパターン数を小さくしたい（コンパクトなテスト集合）という要求が強い．

本論文は，組合せ回路を対象とし，論理ゲート回路レベルにおける 2 つのライン間のブリッジ故障の検出に焦点を絞った，IDDQ テスト用のコンパクトなテスト集合の生成法に関するものである．ブリッジ故障とは，2 つのライン間が短絡する故障で，それら 2 つのラインの論理値が互いに背反になるような外部入力パターンを与えれば，IDDQ テスト法により検出できる[1]（**図 1**）．

このようなブリッジ故障検出のためのコンパクトな IDDQ テスト集合の生成法としては，ある補助回路を付与し，deterministic ATPG（Automatic Test Pattern Generation）によってコンパクトパターンを直接生成する手法[2]，遺伝的アルゴリズムを用いた手法[3]，また，与えられたテスト集合を，(拡張）必須故障に着目して圧縮する手法[4] などが，過去に提案されている．さらにその後，筆者らによって，IDDQ テスト法に対する乱数パターンの有効性を利用した手法，すなわち，乱数パターンをさらにより多くの未検出故障を検出する方向に逐次改善していくことにより，コンパクトパターンを得る手法が提案されている[5]．以降，これを「乱数逐次改善法」と呼ぶ．

しかし，これらの従来手法はいずれも，大規模回路

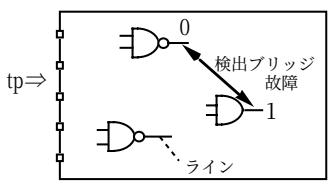


**図 1** 検出ブリッジ故障
Fig. 1 Detected bridging fault.

† 三重大学工学部電気電子工学科
Department of Electrical and Electronic Engineering, Faculty of Engineering, Mie University

に対してはやはり，テスト生成に大量の計算時間を要する．その一方で，大規模回路では，テストパターン数をまだ小さくする余地があることが予想される．乱数逐次改善法は，アルゴリズムはほとんど変えずに，与える計算パワーを増やすことだけで，コンパクション性能を上げることができるという性質を持つ．しかし，筆者らは，文献 5) において，計算時間がかかりすぎるために，乱数逐次改善法に対して十分な計算パワーを与えることができず，大規模回路に対する乱数逐次改善法の能力を十分には示すことができなかった．そこで，本論文は，大規模回路をターゲットに置き，乱数逐次改善法に内在する個々の部分処理の独立性に着目し，2 つのレベルでの並列処理——レベル 1：複数の計算機による並列処理と，レベル 2：各計算機内におけるパターン並列論理シミュレーション——を利用した，乱数逐次改善法に基づく並列生成システムを構築する．これにより，LAN 結合された数台の計算機を用いることによって 100 倍程度の計算パワーを乱数逐次改善法に対して与えることができるようになり，本手法が持つ大規模回路に対するコンパクション能力を大きく引き出すことが可能になる．

さらに，乱数逐次改善法では乱数パターンを改善するが，そこでは残未検出故障が少なくなってくると検出パターンが得にくくなってくるため，特に大規模回路に対しては，故障検出効率 100%の達成が困難であるという，もう 1 つの問題がある．そこで，残未検出故障が少なくなった後で，deterministic ATPG を呼び，それが生成するテストパターンを乱数パターンに代わって逐次改善する方法を，上述の並列生成システムに組み込む．

本論文は，次のように構成する．2 章で乱数逐次改善法を紹介する．3 章と 4 章でそれぞれ，2 レベルの各並列化手法を述べ，5 章で deterministic ATPG の組み込みについて説明し，6 章で全体システムを示す．7 章では評価実験結果を示し，8 章でまとめる．

## 2. 乱数逐次改善法

乱数逐次改善法[5] は，乱数パターンを，より多くの未検出故障を検出する方向に，1 ビットずつ順に改善していくことにより，1 つのテストパターンでできるだけ多くの未検出故障を検出するようなパターンを作る手法である．各テストパターンが多くの新しい故障を検出するので，その結果，テスト集合全体のテストパターン数（テストサイズ）は小さくなる．**図 2** に手続きを示す．関数 main は，コンパクトパターン生成と検出故障ドロップを繰り返す．関数乱数逐次改善

```
main(){
  u ← 故障集合;
  for ( i←0; ！打ち切り条件*1; i++) {
    tpi[i] ← 乱数逐次改善(); 新検出故障をuから削除;
} }
乱数逐次改善(){
  tps←φ; ip←ランダム選択した外部入力ピン番号;
  p←乱数パターン; bv ← nNew_detF(p);
  while (1) {
    q←p; q中のipビット値を反転; v←nNew_detF(q);
    if v > bv then { p←q; bv ← v; }
    else { /* 非改善ステップ*/
      if 改善が収束した*2 then {
        tps ← tps∪{p}; if ｜tps｜≧ nseeds then goto epilog;
        p←乱数パターン; bv ← nNew_detF(p);
    } }
    ip ← ip +' 1; /* i +' j は，(i+j) mod n. (n:外部入力ピン数) */
  }
epilog:
  tps のなかで，新検出故障数が最も多いパターンを返す;
}
nNew_detF(pat){
  論理シミュレーション(pat);
  新検出故障数を計算し，その値を返す;
}
```

*1：打ち切り条件は，(1)十分な検出率が得られた，または，(2)これ以上新しく検出故障が見つけられそうもない（(2)については，現在のシステムでは，関数乱数逐次改善により生成されたパターンが新しい故障を 1 つも検出しないことが連続して nlimit回発生した時としている．nlimit値は経験的に定める）．

*2：非改善ステップばかりが続いたまま，ipが外部入力ピンを一周したとき．

**図 2** 乱数逐次改善法の手続き[5]

Fig. 2 Procedure for iterative improvement method of random patterns[5].

中のループで，現最良パターン $p$（最初は乱数パターンを与える）と，改善ビットポインタ $ip$ が指す $p$ 中の 1 ビット値を反転したパターン $q$（これを現最良パターンに対する「対抗パターン」と呼ぶ）との良い方（未検出故障を数多く検出する方）を新しく現最良パターン $p$ とした後 $ip$ を 1 つ進める操作を,「収束する」まで繰り返す（ここで,「収束する」とは，現最良パターンのいずれの 1 ビットを反転しても，現最良パターンより多くの未検出故障を検出するパターンが得られないようになることをいう）．この繰返しを「改善プロセス」と呼ぶ．これで得られる収束パターンは乱数パターンに依存するために，必ずしも最も多くの数の故障を検出するような真の最善パターンは得られないという弊害を緩和するために，多数（$nseeds$ 個，$nseeds$：経験的に定める値）の乱数パターンを改善し，最も良い収束パターンを採用する．関数 nNew_detF では，論理シミュレーションを行い，新検出故障数を求める．

異なる乱数パターンの改善プロセス間は独立であり，並列実行が可能である．3 章と 4 章で述べる分散並列逐次改善法は，関数乱数逐次改善を並列化により高速

化して，大量の乱数パターンを改善し，できるだけコンパクトなパターンを 1 つ探そうとするものである．

## 3. 計算機内並列化手法

ワード内並列論理演算を利用して，1 つの計算機内で 32 個（32 bit マシンの場合）の入力パターンの論理シミュレーションを並列に行う parallel-pattern evaluation [6]（本論文では,「並列論理シミュレーション」と呼ぶ．**図 3**）が知られており，本手法ではこれを利用する．このシミュレータに与える 32 個の入力パターンを，本論文では「並列入力パターン」と呼ぶ．

本並列法では，**図 4** に示すように，各改善プロセスに並列入力パターン中の 1 bit 分（1 slot）を与え，別々の 32 個の改善プロセスを，並列論理シミュレーションを用いて，並列に走らせる．すなわち，並列入力パターンは，$\langle p_0, p_1, \cdots, p_{n-1} \rangle$．ここで，$n$ は外部入力ピン数で，各 $p_i$ $(i = 0, 1, \cdots, n-1)$ は，外部入力ピン $i$ に与える 32 パターン分の信号値で，$p_i = \langle p_i.0, p_i.1, \cdots, p_i.31 \rangle, p_i.j \in \{0, 1\}$．$p_i.j$ は，$j$ $(j = 0, 1, \cdots, 31)$ 番目の入力パターンの外部入力ピン $i$ の値を意味する．各 $p_i$ は，計算機内では各々1 ワード（32 bit を仮定）内に実現する．$j$ 番目の入力パターンを $p.j$ と記し，それに対応する並列入力パターン $p$ 上の格納域を「slot $j$」と呼ぶ（図 4，**図 5**）．

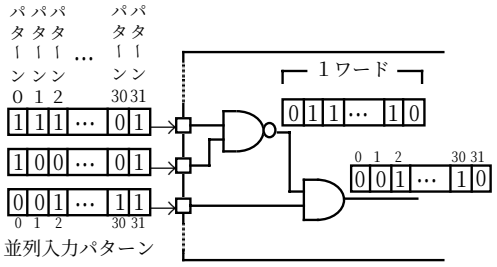


**図 3** 並列論理シミュレーション [6]
Fig. 3 Parallel-pattern logic simulation [6].

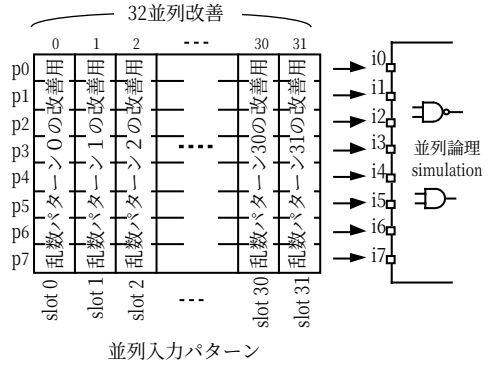


**図 4** 32 並列逐次改善法（PII32）
Fig. 4 32 pattern parallel iterative improvement method (PII32).

図 5 は 1 改善ステップでの動作を表す．32 個の各改善プロセスごとに現最良パターン $p.j$ $(j = 0, 1, \cdots, 31)$ とその新検出故障数 $bv_j$ を保持する．改善ビットポインタ $ip$ は，32 個の改善プロセス間で共通化する．これにより，図 5 のように，並列入力パターンとなる 32 個の対抗パターンが，1 つの反転命令で生成できるという利点以外に，前回の並列論理シミュレーション時に与えられた並列入力パターン（$ip$ は 1 つ前のビットを指していた）と今回の並列入力パターン間の差異を，多くとも 2 連続ビット（$ip$ が指すビットと 1 つ前のビット）の違いのみに局所化することができ，シミュレータのイベントドリブン化による高速化がはかれるという利点がある．

並列論理シミュレーション後，32 個の各改善プロセスごとに，現最良パターンと対抗パターン間で新検出故障数を比較し，各々slot ごとに，良い方のパターンを新しく現最良パターンとし，$ip$ を 1 つ進め，次の改善ステップに移る．

2 章で述べたように改善は収束するまで行われるが，乱数パターンごとに収束するまでの改善ステップ数は異なる．したがって，32 個のパターンすべてが収束するまで上述の並列改善を続行するのは，早く収束したパターンにとっては，それ以上改善されることはないので，無駄な処理となる．そこで，32 個のパターンのうちの 1 つのパターンが収束すると並列改善を中断し，その slot の収束パターンだけを取り出し，新しい乱数パターンに入れ替えた後，他の改善途中の 31 個の未収束パターンとともに，並列逐次改善を再開するようにして，無駄なく多数の乱数パターンの改善を次々に進めていくようにする．この場合のパターン入れ替え

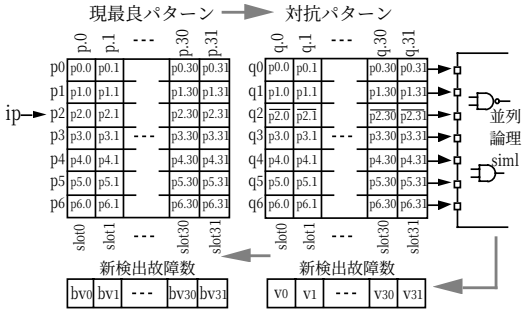


**図 5** PII32 における 1 改善ステップ
Fig. 5 One improvement step in PII32.

のための中断回数は，コンパクトテストパターン 1 個得るための処理ごとに $nseeds$ 回（$nseeds$ はたかだか数千）程度であり，これによる計算時間の損失は，本手法中の他の部分に費やされる膨大な計算時間量に比べると，無視できる．実際，7 章の実験結果で，乱数逐次改善法全体の計算時間はたとえば ISCAS89[8] の大規模回路で数十分以上必要であり，そのほとんどは，並列論理シミュレーションを行い，新検出故障数を数える部分（**図 6** 中の関数 para_nNew_detF）で費やされている．

図 6 に手続きを示す．ワード内並列処理に関すること以外は図 2 とほぼ同じ構造である．

```
乱数逐次改善(){          /* 32並列版  PII32 */
  tps←φ;
  ip←ランダム選択した外部入力ピン番号;
  j=0,1,...,31に対して { p.j ← 乱数パターン; }
  bv0, bv1,..., bv31 ← para_nNew_detF(p);
  while (1) {
    q ←p ;  qipのすべてのビットを反転;
    v0, v1, ..., v31 ← para_nNew_detF(q);
    j=0,1,...,31に対して {
      if vj > bvj then { pip.j ← qip.j; bvj ← vj; }
      else{ /* slot j は非改善 */
        if slot j の改善が収束した*1 then {
          tps ← tps ∪{ p.j };
          if | tps | ≧ nseeds then goto ret;
          p.j ← 乱数パターン; bvj ← 0;
    } }  }
    ip ← ip +' 1;  /* i +' j は (i+j) mod n */
  }
ret: tps と { p.j | j=0,1,...,31 }のなかで,
                        /* p.j: 未収束パターン */
     新検出故障数が最も多いパターンを返す;
}
para_nNew_detF(parapat) {
 イベントドリブン並列論理シミュレーション(parapat);
 各 slot j(j=0,1,...,31)に対して,新検出故障数を
 計算し，それら32個の値を返す;
}
*1 : slot jについて, 非改善ステップばかりが
   続いたまま，ipが外部入力ピンを一周したとき
```

**図 6** PII32 の手続き
Fig. 6 Procedure for PII32.

## 4. 複数の計算機によるさらなる並列化

前章では，1 つの計算機内での 32 並列化手法（単に PII32 と呼ぶ）を述べた．しかし，さらに大規模な回路では 32 並列での高速化では不十分な場合もでてくる．本章では，一般に使えるような，LAN 結合されている複数の計算機を用い，各計算機上で PII32 が走る分散並列システムを構築する．これにより，たとえば 4 台の計算機が使えれば 100 倍程度の高速化が期待できる．

このシステムは，PII32 が走る改善サーバが計算機台数個と，それらを制御する manager プロセス 1 つからなる．各改善サーバは，得た収束パターンを manager に報告する．manager は，システム全体の収束パターン数の総和が $nseeds$ 個に達したら，一番良い収束パターンをコンパクトパターンとして採用する．改善サーバ・manager 間の通信回数を低減するために，各サーバは収束パターンがある程度の個数（$[nseeds/$改善サーバ数$]$ 個）得られたら manager に報告する．さらに，次に説明するように，コンパクトパターンを 1 つ得るために生成する収束パターンの数は正確に $nseeds$ 個である必要がないことを利用して，分散計算ゆえに起こりうる CPU の遊びや無駄な計算をできるだけおさえるようにしている．

**図 7** に本分散並列逐次改善手続きの主要部を示す．manager や各サーバの最外部の繰返しごとに，1 つのテストパターン tpi[i] が得られる．サーバ内の PII32 の実行で $[nseeds/$サーバ数$]$ 個の収束パターンが得られたら，manager に $READY$ 通知する．ただし，ここではまだテストパターンは送付せず，PII32 をさらに続行する．manager 側では，この $READY$ 通知を，半数のサーバから受け取ったら，各サーバに，一番良いテストパターンを送るよう指示を出す（$SEND!$ 指令）．これは，この時点で，システム全体では，総収束パターン数がほぼ $nseeds$ 個に達していると推察されるからである．一方，サーバ側では，どのサーバも

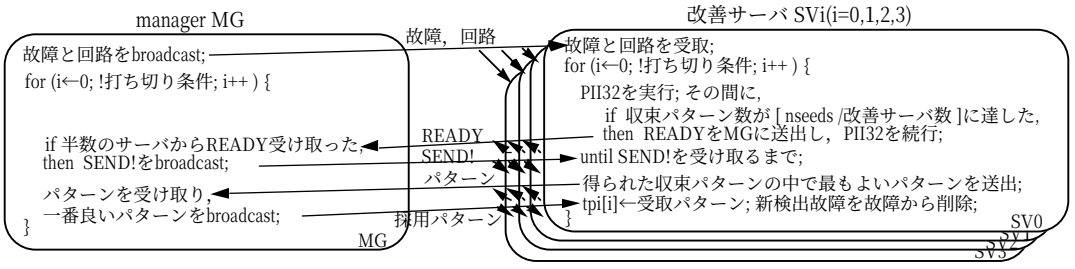


**図 7** 分散並列逐次改善手続き（主要部）
Fig. 7 Outline of the distributed algorithm.

PII32 を行っているが，*SEND*! 指令が来たとき初めて，この時点までに得られている一番良い収束パターン（とこれまでに得られた収束パターン数）を送る．manager は全体で最も良いテストパターンをコンパクトパターンとして採用する．（manager はシステム全体の収束パターン数の総和を求め，その結果その数がまだ十分ではなかった場合に対処する必要があるが，そのためのコードは省略した．）

## 5. deterministic ATPG によるアシスト

乱数パターンを改善する乱数逐次改善法は，残未検出故障が少なくなると，検出パターンが得にくくなるという欠点を持つ．そのため，大規模回路では，並列処理により大量の乱数パターンを改善しても未検出故障が残り，故障検出効率 100%を得るのは困難であるという問題が残る．

この問題に対して，次のように，deterministic ATPG のアシストを受けることにより解決をはかる．すなわち，乱数パターンの逐次改善によって，残未検出故障が少なくなり，新しく残未検出故障を検出するテストパターンが得にくくなったところで，deterministic ATPG を呼ぶ．deterministic ATPG では，すべての各残未検出ブリッジ故障 a-b に対し，justification（a xor b = 1）を行う．その後，justification が失敗した故障は検出不能であるので残未検出故障集合から取り除き，残った検出可能故障集合を対象にして，deterministic ATPG により得られたテストパターンを乱数パターンに代わって逐次改善する．このようにして，コンパクトでかつ故障検出効率 100%のテスト集合を得ることができる．

実用的な deterministic ATPG の実行のためには，前もって回路に関する学習[6] が必要であるが，deterministic ATPG は，乱数パターンの逐次改善の後に行われるので，学習についても，乱数パターンの並列改善と並行して行うことができる（**図 8** 右端部）．

なお，deterministic ATPG 部では，各残未検出故障ごとに justification が行われるが，ここでは検出故障ドロップを行う必要性はあまりないので，複数の justification の並列実行が可能である．しかし，deterministic ATPG 部の計算時間は，システム全体から比べると大きくないので，現在のところは，この部分の並列化は行っていない．

## 6. 全体システム

図 8 に，システム全体の並列動作フローを示す．時間は上から下へと流れていく．テスト生成に入る前に，

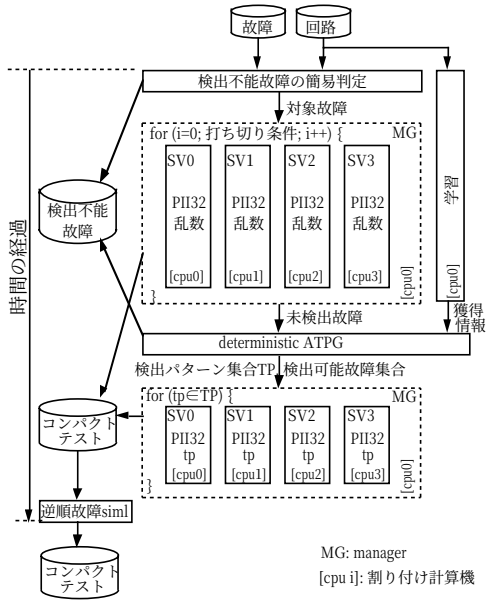


**図 8** 全体システムの並列動作フロー
Fig. 8 Execution flow of the parallel system.

検出不能故障の簡易判定（文献 2) と同様の手法）を行い，対象故障数を減らしておく．また，最後には，IDDQ テスト用の逆順故障シミュレーション[10] を行う．これら 2 つの効果はあまり大きくはないが，計算コストがきわめて小さいので，システムとしては組み込んでおいた方がよい．

なお，図 8 の [ ] 内に示す計算機の割付けは，4 台の計算機を使った 7 章の評価実験用システムのためのものであり，もう 1 台計算機が使用できれば，それに学習プロセスと manager プロセスを割り付けてもよい．

## 7. 評価実験結果

図 8 で示した全体システムを UNIX の socket システム[9] 上に試作し，性能評価実験を行った．本評価実験では，従来手法の結果との比較のために，すべての 2 つの論理ライン間の全ブリッジ故障を対象故障としたが，本手法自体は，レイアウト情報から抽出した起こりうるブリッジ故障を対象故障にする場合も，そのまま適用することができる．本試作システムでは，deterministic ATPG を呼ぶのは，乱数パターンの逐次改善によるテスト生成が進んできて，関数乱数逐次改善により生成されたパターンが新しい故障を 1 つも検出しないことが発生したとき（図 2 の関数 main 中の打ち切り条件の*1 の説明の中の条件 (2) の *nlimit* 値を 1 に設定することに対応）とした．本評価実験で使用した計算機の CPU はすべて，350 MHz Pentium-II

**表 1** 大規模回路に対する実験結果
Table 1 Experimental results for large circuits.

| s38417 | | nseeds数 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| テストサイズ | 平均 | 107 | 104 | 100 | 94 | 93 | 88 | 85 |
| | 最大 | 113 | 109 | 105 | 97 | 95 | 90 | 87 |
| | 最小 | 102 | 97 | 97 | 90 | 89 | 85 | 82 |
| 検出効率 | | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| 計算時間 (h:m) | 計算機1台 | 0:12 | 0:18 | 0:30 | 0:53 | 1:37 | 3:06 | — |
| | 2台 | — | 0:11 | 0:17 | 0:30 | 0:58 | 1:50 | — |
| | 3台 | — | — | 0:12 | 0:20 | 0:37 | 1:10 | — |
| | 4台 | — | — | 0:10 | 0:16 | 0:28 | 0:56 | 1:47 |

| s38584 | | nseeds数 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| テストサイズ | 平均 | 138 | 135 | 132 | 127 | 122 | 121 | 117 |
| | 最大 | 143 | 140 | 136 | 129 | 127 | 124 | 119 |
| | 最小 | 135 | 131 | 129 | 122 | 119 | 118 | 115 |
| 検出効率 | | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| 計算時間 (h:m) | 計算機1台 | 0:31 | 0:39 | 0:57 | 1:31 | 2:39 | 5:03 | — |
| | 2台 | — | 0:26 | 0:33 | 0:53 | 1:31 | 2:56 | — |
| | 3台 | — | — | 0:27 | 0:36 | 1:00 | 1:47 | — |
| | 4台 | — | — | 0:24 | 0:30 | 0:46 | 1:25 | 2:42 |

であり，本並列システムでの計算時間（表 1，図 10，表 3）は，計算機を占有して実行したときの経過時間を示し，図 8 中の「時間の経過」に対応する時間である．

すべての実験において，乱数改善後に残ったすべての故障は，deterministic ATPG により短時間で，検出不能故障と判断されるか，そうでない場合はテストパターンが得られ，故障検出効率 100%がつねに得られた．

**表 1** に，ISCAS89[8]（full scan を仮定し組合せ回路扱い）の中で最大の回路 s38417 と s38584 に対する実験結果を示す．テストサイズとしては，異なる乱数系列を用いた 8 回の生成実験の結果の平均，および，結果のばらつき度を示すため，最大値と最小値を示した．表 1 には，各コンパクトパターンを得るために生成する収束パターンの数 *nseeds* とテストサイズとの関係，および，計算機台数と計算時間との関係が示されている．**図 9** と**図 10** はそれらをグラフ化したものである．これを見ると，(1) *nseeds* を増やすと各テストパターンの新検出故障数が増えるため，その結果，図 9 に示されている範囲では着実にテストサイズが減っており，*nseeds* を 1024 程度にしてもまだ飽和していないこと，および，(2) 速度に対する台数効果がきわめて良いこと，特に *nseeds* が大きいとき，速度向上率は，少なくとも計算機 4 台までは，ほぼ線形に伸びていること（図 10）などが分かる．これら 2 つのことから，さらに大規模回路に対しては，さらに多数台の計算機によるいっそうの並列効果が期待できる．また，表 1 中に示したテストサイズの最大値と最小値の差を見ると，結果として得られたテストサイズのばらつきも，*nseeds* を増やすと減少していく傾向が見られる．

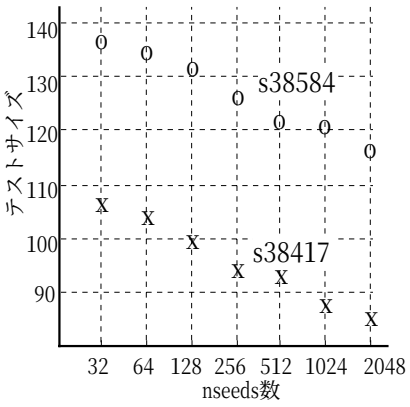


**図 9** *nseeds* 数 vs. テストサイズ
Fig. 9 Test set sizes for different *nseeds*.

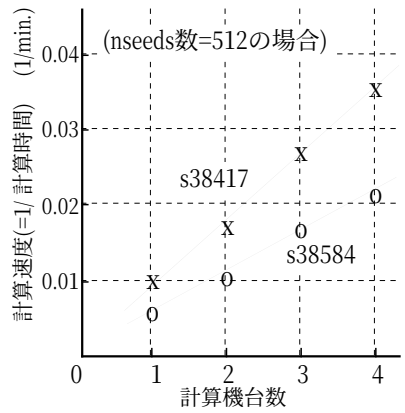


**図 10** 速度の台数効果
Fig. 10 Speed-up by multiple computers.

本システムでは，1 台の計算機を用いた場合には，実際に得られる収束パターン数は与えた *nseeds* 値と一致する．しかし，複数台の計算機を用いた場合には，分散計算ゆえに起こりうる CPU の遊びの抑制や無駄な計算の抑制のために，与えた *nseeds* に対して，実際に得られる収束パターン数は実験ごとに毎回変動するという性質を持つ（4 章）．したがって，複数台の計算機を用いた場合には，1 台のときよりも，同じ *nseeds* でもテストサイズ結果が安定しにくくなるという危険性がある．このことを実験により調べてみた．1 台の計算機と 4 台の計算機を用い，s38417 と s38584 に対して，異なる乱数系列を用いた 10 回の実

**表 2** テストサイズのばらつき
Table 2 Variance of test sizes.

| | 計算機 1 台での結果 | 計算機 4 台での結果 |
|---|---|---|
| s38417 | 88, 90, 93, 91, 93<br>94, 90, 87, 92, 94<br>平均 91.2<br>最大 94<br>最小 87 | 89, 95, 93, 89, 88<br>89, 95, 91, 93, 94<br>平均 91.6<br>最大 95<br>最小 88 |
| s38584 | 123, 117, 122, 125, 127<br>122, 128, 123, 123, 120<br>平均 123.0<br>最大 128<br>最小 117 | 121, 124, 124, 122, 125<br>120, 124, 122, 121, 126<br>平均 122.9<br>最大 126<br>最小 120 |

(nseeds=512を与えた時)

**表 3** 従来手法の結果との比較
Table 3 Comparison of experimental results with previous methods for large circuits.

| | 本並列システム CPU 4 台 nseeds 512 | 本並列システム CPU 1 台 nseeds 64 | 文献 5) nseeds= 20 for C, 5 for S | 文献 2) | 文献 3) | 文献 4) |
|---|---|---|---|---|---|---|
| c7552 | | | | | | |
| 検出効率 [%] | 100% | 100% | NA | 100.00% | 99.99% | 100% |
| テストサイズ | 29 | 31 | 27 | 36 | 51 | 31 |
| 計算時間 [h:m] | 0:02 | 0:01 | 0:05 | 1:32 | 0:25 | 0:25 |
| s15850 | | | | | | |
| 検出効率 [%] | 100% | 100% | NA | | 99.99% | |
| テストサイズ | 122 | 126 | 130 | | 194 | |
| 計算時間 [h:m] | 0:07 | 0:05 | 0:46 | | 4:36 | |
| s35932 | | | | not | | not |
| 検出効率 [%] | 100% | 100% | NA | | 100.00% | |
| テストサイズ | 22 | 23 | 22 | | 32 | |
| 計算時間 [h:m] | 0:26 | 0:18 | 1:26 | reported | 1:18 | reported |
| s38417 | | | | | | |
| 検出効率 [%] | 100% | 100% | NA | | 99.99% | |
| テストサイズ | 92 | 106 | 137 | | 257 | |
| 計算時間 [h:m] | 0:29 | 0:18 | 7:25 | | 18:48 | |
| s38584 | | | | | | |
| 検出効率 [%] | 100% | 100% | NA | | 99.99% | |
| テストサイズ | 123 | 134 | 174 | | 315 | |
| 計算時間 [h:m] | 0:46 | 0:39 | 7:51 | | 27:54 | |

本並列システムの計算時間は，図 8 の「時間の経過」に要した時間.

**表 4** 従来手法とのテストサイズの比較
Table 4 Comparison of test set sizes with previous methods.

| | テストサイズ | | | | |
|---|---|---|---|---|---|
| | 本システム nseeds 512 | 文献 5) | 文献 2) | 文献 3) | 文献 4) |
| c880 | 13* | - | 17 | - | 14 |
| c1355 | 63* | 63* | 66 | 68 | 63* |
| c1908 | 43* | 44 | 44 | (44) | 43* |
| c2670 | 21 | (20) | 23 | (29) | 20* |
| c3540 | 52 | 52 | 52 | (61) | 50* |
| c5315 | 24* | 25 | 27 | (36) | 26 |
| c6288 | 21* | 22 | 23 | 28 | 25 |
| c7552 | 29* | (27) | 36 | (51) | 31 |
| s5378 | 61* | (70) | - | 96 | - |
| s9234 | 101* | (102) | - | (162) | - |
| s13207 | 188* | (183) | - | 243 | - |
| s15850 | 122* | (130) | - | (194) | - |
| s35932 | 22* | 22* | - | 32 | - |
| s38417 | 92* | (137) | - | (257) | - |
| s38584 | 123* | (174) | - | (315) | - |

( ): 非100%検出効率
* : 100%検出効率でテストサイズ最小のもの

験により得られたテストサイズのすべての結果を**表 2** に示す．これを見ると，1 台の計算機の場合の結果と 4 台の計算機の場合の結果のばらつき方に差異はあまり見られず，本システムにおける，分散計算ゆえに起こる収束パターン数のばらつきによるテストサイズへの有意な影響は，ほとんどないことが分かった．

**表 3** に，従来手法との性能比較実験結果を示す．ISCAS85[7] の最大の回路 1 つと ISCAS89[8] の最大の回路 5 つの結果を示した．本並列システムの結果は，異なる乱数系列を用いた 10 回の生成実験の結果の平均である．*nseeds* としては，計算機 1 台の場合には 64，4 台の場合には 512 の結果を示した．文献 2) と文献 4) は Sparc-5，文献 3) は Sparc-2，文献 5) は，Pentium-Pro 200 MHz の結果である．我々の測定によると，350 MHz Pentium-II は 200 MHz Pentium-Pro の 1.5 倍程度速く，200 MHz Pentium-Pro は 110 MHz Sparc-5 に比べ 4 倍程度速い．

c7552 において，そのテストサイズが元の乱数逐次改善法[5] に比べ大きくなったのは，文献 5) では 100%故障検出効率を達成していなかったためと思われる．表 3 から，(1) 1 台の計算機では，s38417 と s38584 の両方の回路で，文献 5) に比べ，1 桁程度少ない計算時間で，テスト数が 23%減少していること，および，(2) 4 台の計算機では，さらに大量の乱数パターンを与えることができるようになり，同じ両回路で，文献 5) に比べ，1 桁程度少ない計算時間で，テストパターン数が約 30%程度（約 50 個程度）減少していることなどが分かる．さらに，(3) ISCAS 回路の中で比較的中規模の回路では，コンパクション性能への並列化効果はあまり大きくないが，逆にいうと，計算時間を大量に必要とする対象規模が大きい場合に，台数効果が顕著に現れるという，並列システムにとって望ましい性質があるといえる．

最後に，主要な ISCAS 回路に対する生成テスト集合のサイズを**表 4** に示しておく．ほとんどすべての回路において，5 つの結果の中で最小のテスト数が得られている．

以上，本章の評価実験により，大規模回路に対する，本並列生成システムの有効性が確認できた．

## 8. おわりに

本論文では，大規模な組合せ論理回路をターゲットとし，ライン間のブリッジ故障の検出のための IDDQ テスト用のコンパクトなテストパターン集合の高性能な生成システムを提案した．

乱数逐次改善法[5] に内在する個々の部分処理の独

立性に着目し，2 つのレベルでの並列処理を利用した並列生成システムを構築した．これにより，たとえば LAN 結合された 4 台の計算機を用いることによって 100 倍程度の計算パワーを乱数逐次改善法に対して与えることができるようになり，本手法が持つ大規模回路に対するコンパクション能力を大きく引き出すことに成功した．また，乱数パターンの逐次改善では故障検出効率 100%の達成が困難であるという問題に対しては，乱数逐次改善法により残未検出故障が少なくなった後で，deterministic ATPG を呼び，それが生成するテストパターンを乱数パターンに代わって逐次改善する方法を，上述の並列生成システムに組み込むことにより解決した．評価実験では，4 台の計算機を用いた場合，ISCAS の最大の 2 つの回路に対してともに，文献 5) の結果より，1 桁程度少ない計算時間で，テストパターン数を 30%程度（50 個程度）減少させることができ，本手法の有効性を確認した．

本研究の目的は，LSI の検査に要する時間を短縮するため，すなわち，LSI の検査時間コストの低減のために，テストパターン数の減少をはかるものであった．一般に，各テストパターンを生成するのに用いる乱数パターンの数（*nseeds* 値）を増やせばテストパターン数は減少するが（図 9），一方，テストパターン集合の計算コスト（計算時間 × 計算機台数）は，*nseeds* 値にほぼ比例して増える．このテスト集合計算コストと LSI 検査時間コストのトレードオフを考慮した総合的な効率の評価が，今後の課題の 1 つである．さらに，最小テストパターン数の真値にできるだけ近い下界を求めることにより，コンパクション性能を評価することも，今後の課題としてあげられる．

## 参 考 文 献


1) Chakravarty, S. and Thadikaran, P.J.: *Introduction to IDDQ Testing*, Kluwer Academic Publishers (1997).
2) Reddy, R.S., Pomeranz, I., Reddy, S.M. and Kajihara, S.: Compact Test Generation for Bridging Faults under IDDQ Testing, *VLSI Test Symposium*, pp.310–316 (1995).
3) Thadikaran, P. and Chakravarty, S.: Fast Algorithms for Computing IDDQ Tests for Combinational Circuits, *9th International Conference on VLSI Design*, pp.103–106 (1996).
4) Kondo, H. and Cheng, K.-T.: An Efficient Compact Test Generator for IDDQ Testing, *Asian Test Symposium*, pp.177–182 (1996).
5) Shinogi, T. and Hayashi, T.: A Simple and Efficient Method for Generating Compact IDDQ Test Set for Bridging Faults, *VLSI Test Symposium*, pp.112–117 (1998).
6) Abramovici, M., Breuer, M.A. and Friedman, A.D.: *Digital Systems Testing and Testable Design*, IEEE Press (1990).
7) Brglez, F. and Fujiwara, H.: A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran, *International Symposium on Circuits and Systems* (1985).
8) Brglez, F., Bryan, D. and Kozminski, K.: Combinational Profiles of Sequential Benchmark Circuits, *International Symposium on Circuits and Systems*, pp.1929–193 (1989).
9) Stevens, M.R.: *UNIX Network Programming*, Prentice Hall (1990).
10) Schulz, M.H., Trischler, E. and Sarfert, T.M.: SOCRATES: A Highly Efficient Automatic Test Pattern Generation System, *IEEE Trans. Computer-Aided Design*, Vol.7, No.1, pp.126–137 (1988).


(平成 11 年 9 月 14 日受付)
(平成 12 年 2 月 4 日採録)

**篠木 剛**（正会員）

1954 年生．1977 年東京工業大学理学部情報科学卒業．1979 年同大学院理工学研究科修士課程修了．同年（株）富士通研究所入社．1985 年より 1 年間米国オレゴン大学客員研究員．Lisp マシン Facom $\alpha$，並列推論マシン PIM/p の研究開発に従事．1998 年三重大学大学院工学研究科博士後期課程修了．工学博士．1998 年三重大学工学部講師．1999 年助教授．LSI のテスト生成システムや設計支援システム，並列/分散計算機システム等に興味を持つ．電子情報通信学会，IEEE 各会員．

**林 照峯**（正会員）

1947 年生．1969 年名古屋大学工学部電気学科卒業．1971 年同大学院工学研究科修士課程修了．同年（株）日立製作所日立研究所入社．1993 年三重大学工学部電気電子工学科教授．工学博士．LSI の設計自動化，論理回路のテストおよびテスト容易化設計等の研究に従事．電子情報通信学会，IEEE 各会員．