設 計 自 動 化 77−2
(1995. 10. 19)

実数値シミュレーションに基づくテスト生成法の性能向上について

篠木 剛　　内田 智之　　北 英彦　　林 照峯

三重大学

〒514 三重県津市上浜町1515
三重大学 工学部 電気電子工学科
計算機工学研究室
☎ 0592-32-1211 内線 3916
shinogi, uchida, kita, hayashi@hayashi.elec.mie-u.ac.jp

あらまし
シミュレーションベースのテスト生成法として実数値シミュレーション法がある。我々は、組合せ回路を対象にして、実数値シミュレーション法の基本能力の向上を試みた。すなわち、"実数値シミュレーション"の速度向上のために、①故障箇所に対する有効ゲート以外の計算を除去し、さらに、②値が変化するゲート以外の計算を除去した。また、検出率向上のために、③複数の乱数初期入力パターンによる逐次改善と④各ゲート出力値の補正手法を導入した。その結果、実数値シミュレーション法によって、ISCAS85、組合せ回路としてのISCAS89のすべてのベンチマーク回路で、冗長故障を除く故障を、実用的な時間内で100%検出することができるようになり、実数値シミュレーション法の基本能力を大幅に向上させることができた。

キーワード　テスト生成法、実数値シミュレーション、反復改善法、故障検出

On Improvement of an ATPG based on Real-valued Logic Simulation

Tsuyoshi Shinogi, Tomoyuki Uchida, Hidehiko Kita, Terumine Hayashi

Mie University

Computer Engineering Laboratory
Department of Electrical and Electronic Engineering
Faculty of Engineering, Mie University
1515 Kamihama-cho, Tsu-shi
Mie-ken, 514 JAPAN
Phone 0592-32-1211 ext. 3916
shinogi, uchida, kita, hayashi@hayashi.elec.mie-u.ac.jp

Abstract

An ATPG based on real-valued logic simulation has already been proposed. We enhance its basic performance by introducing the following techniques to the combinational ATPG. For speeding up, (1)eliminating the computation for real-valued logic simulation of ineffective gates for detection of each fault and also (2)eliminating the computation for gates unreachable from the selected primary input in real-valued logic simulation. And for improving fault coverage, (3)the iterative improvement procedure by multiple random input patterns and (4)adjustment of the output real value of each gate. As a result, by the ATPG based on real-valued logic simulation, we have achieved 100% fault coverage for all the faults without redundant faults in each ISCAS '85 and ISCAS ' 89 (assuming full-scan) circuit in practical time.

key words　automatic test pattern generation, real-valued logic simulation, iterative improvement method, fault detection

1．はじめに

半導体の設計技術や製造技術の進歩により、様々な用途のLSIが作られるようになり、LSIの少量多品種化や大規模化が進んでいる。それに伴い、開発期間の短縮や工数の削減のために、LSIのテストデータの自動生成能力のさらなる向上が必要である。これまで、論理回路のテスト生成技術は、単一縮退故障の検出方式に関する研究を中心にして発展してきた。それらの方式の多くは、ほぼ次の２つに分類される。

①回路構造を解析し探索しながらテストパターンをアルゴリズミックに計算する決定論的な(deterministic)方式([Fujiwara83][Schulz88][Teramoto93][Cheng89][Auth91]など)、および、

②初期入力パターンを適当に与え、評価関数（本論文ではコスト関数と呼ぶ）や論理シミュレーションを用いて、入力パターンの一部を変更していきながら、逐次的にテストパターンを誘導するシミュレーションベースの方式([Agrawal89][池田89][彦根92][Hatayama92][伊達94][Rivin94]など)、

である。特に、前者の決定論的な方式は古くから広く研究されてきており、特に組合せ回路のテスト生成法として優れた成果が生み出され、さらに順序回路にも適用され、実用化も進んでいる。後者のシミュレーションベースの方式については、当初から順序回路への適用をめざして研究が進められてきたが、まだ、大規模順序回路のテスト生成問題が、完全に解決されたと言えるところまでには至っていない。この方式に分類されるもの中に、"実数値シミュレーション"を用いたテスト生成法が提案されている[池田89][彦根92][Hatayama92][伊達94]。既に、対象を組合せ回路から順序回路、そしてその並列化へと研究が進められてきた。そして、単純な構造にもかかわらず、その有効性が明らかになってきている。しかし、実数値シミュレーション法には、次のような問題が一部残っている。

①組合せ回路においても冗長故障を除いた検出率が100%にならないことがある。

②大規模な回路（組合せ回路、順序回路共に）への適用能力が不明である。

最終的にはこれらの問題の解決、すなわち、①冗長故障を除けば100%検出できることと、②大規模回路にも適用できることを目標にし、大規模かつ複雑な順序回路を扱いうるテスト生成技術を確立することを目的とする。しかし、我々は、大規模順序回路に本格的に取り組む前に、まず、実数値シミュレーション法の基本能力を上げることに的を絞ることにし、組合せ回路を対象にして、実数値シミュレーション法の改良を行うことにした。そして、当面の具体的な目標を次のように設定した。

「実数値シミュレーション法によって、ISCAS85[Brglez 85]、組合せ回路としてのISCAS89[Brglez89]のすべてのベンチマーク回路で、冗長故障を除く故障を、実用的な時間内で100%検出すること」

この目標を達成するため、次のような手法を導入した。すなわち、"実数値シミュレーション"の計算量を減らすために、①故障箇所に対する有効ゲート以外の計算を排除するようにし、さらに、②値が変化するゲート以外の計算を排除するようにした。また、100%検出のために、③複数の乱数初期入力パターンによる逐次改善と、④"実数値シミュレーション"の各ゲート出力値の補正手法を導入した。その結果、上記目標が達成でき、実数値シミュレーション法の基本能力を大幅に向上させることができた。

以下、２章では、"実数値シミュレーション"を用いたテスト生成法を紹介し、３章で、計算量を減らすための手法や検出率100%をめざすための手法を述べ、各手法の効果の程を実験結果により評価する。４章では、実数値シミュレーション法の原方式や他のシミュレーションベースの方式との性能比較を行い、５章で、結論、今後の課題を述べる。

2．"実数値シミュレーション"によるテスト生成法

本論文では、単一縮退故障を検出するテストパターンの自動生成問題を扱う。テストパターンの自動生成（テスト生成）問題とは、論理回路と仮定故障集合が与えられたとき、各故障毎に、いずれかの同一の外部出力ピンに、正常回路と故障回路で異なる値が出力されるような、外部入力ピンに与えるデータパターン（入力パターンと呼ぶ）を求めることである。手続き１に"実数値シミュレーション"を用いたテスト生成法（原手法）を示し[池田89]、以下に説明する。

(1) 逐次改善法

手続きctg0は、初めに与えられた乱数入力パターンを、そのコストが改善される方向に、逐次的に変更していくものであり、一般に反復改善法(iterative improvement method) [西川82]と呼ばれる最適化手法の一種である。コストは、入力パターンが真のテストパターンからどれだけ離れているものであるかを示唆するものであり、後述の"実数値シミュレーション"を用いて計算される。ctg0では、まず初めに、乱数パターンを外部入力ピンに与え(PI)そのコスト(c)を計算する。コスト計算するときは、いつも同時に、正常回路と故障回路で論理シミュレーションを行ことによりテストパターンか否かの判定も行い(cost&siml)、テストパターンが見つかればその時点でその故障の処理を終わる。テストパターンではなかったとき、適当に１つ決めた外部入力ピン番号(n：現変更ピンと呼ぶ)に対応する値

```
main (){
  while 故障 ∈ 仮定故障集合 do
    if  ctg0(故障) == 見つかった  then
          故障シミュレーションによる
                    同時検出故障の除去;
}

ctg0(故障){
  PI ← 乱数初期入力ﾊﾟﾀｰﾝ;   /* PI:入力ﾊﾟﾀｰﾝ */
  c0 ← +∞;
  n  ← ランダムに選んだ入力ピン番号;
  do {
    c  ← cost&siml(PI);
    if  PIはﾃｽﾄﾊﾟﾀｰﾝ  then return 見つかった;
    else  if  c0 > c  then {
                PI0 ← PI;  c0 ← c;  i ← 0;
          }
    PI ← PI0の第 n bitをbit反転したパターン;
    n  ← (n + 1) mod 入力ピン数;
  } while i++ < 入力ピン数;
  return 見つからなかった;
}

cost&siml(入力パターン){
  ﾃｽﾄﾊﾟﾀｰﾝか否かを判定すると同時に、
       /* 正常/故障回路で論理simlを行い、   */
       /* 出力ピンに異なる値がでるかを判定 */
  実数値シミュレーションによりコスト計算する;
  return コスト値;
}
```

手続き1. 実数値シミュレーションを用いた逐次改善法

をビット反転し、その新入力パターンのコスト計算を行う。乱数初期入力パターンと新入力パターンのうち、コストの小さい方とそのコスト値を、最小コストパターン(PI0)、最小コスト値(c0)として記録しておく。以降、現変更ピン(n)を順次サイクリックに進め、最小コストパターン中の対応する値をビット反転した入力パターンのコスト値を計算し、最小コストパターン(PI0)、最小コスト値(c0)を更新していく。すなわち、常に、現在記録されている最小コストパターンに対して、現変更ピンに対応する値をビット反転し、そのコスト計算を行う。コスト改善があれば最小コストパターンとして上書き記録し、現変更ピンを進める。コスト改善がなければ、最小コストパターンは以前のままにし、現変更ピンだけを進める。この操作を繰り返し、テストパターンが見つかる（成功）まで、または、最小コストパターンが記録されてから現変更ピンが一回りしても(iで制御)、コスト改善が見られなくなる（失敗）まで行う。

(2)"実数値シミュレーション"によるコスト計算

次に、コスト計算とそれに用いる"実数値シミュレーション"について説明する。"実数値シミュレーション"も、論理シミュレーションと同様、外部入力ピンにデータを与え、シミュレーション計算を行い、外部出力ピンの値を求める。違いは、論理値の代わりに実数値を扱うこと、すなわち、外部入力ピンに論理値の代わりに実数値を与え、各ゲートでは論理演算の代わりに実数演算がなされることである（図１）。外部入力ピンへは論理０、１に対応して、ε、1.0 − ε（εは0に近い正の実数値）を各々与える。NOTゲートでは、1.0からゲート入力値を引いた値を、ANDゲートでは、ゲート入力値どおしの乗算結果を、それぞれゲートの出力値として付与する。他種のゲートも同様である。０／１縮退故障箇所には0.0／1.0を与える。そのようにして、外部入力ピンから外部出力ピンの方向に、すべてのゲートの出力値を計算し、全ての外部出力ピンの値を計算する。

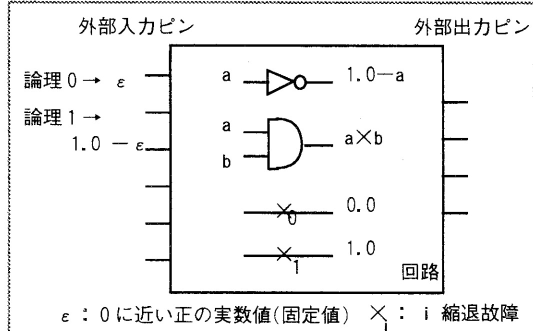

図１. 実数値シミュレーション

入力パターンに対するコスト値は次式で計算する。すなわち、各々の外部出力ピンすべてについて、正常回路と故障回路での"実数値シミュレーション"による出力値（実数出力値と呼ぶ）の差の絶対値を計算し、それらの総和の逆数をとる。

コスト（データパターン）＝

1／Σ｜正常回路の外部出力ピンiの実数出力値−故障回路の外部出力ピンiの実数出力値｜

Σは、すべての外部出力ピンについての総和

以下図２の回路例を用いて、このコスト値が、入力パターンが真のテストパターンからどれだけ離れているものであるかを示唆していることを示す。図３は１つのコスト計算例である。この回路では、(1,0,1)が唯一のテストパターンである。表１はこの回路のすべて

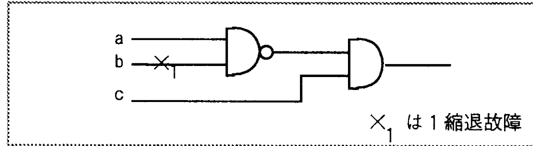

図２. 故障回路例

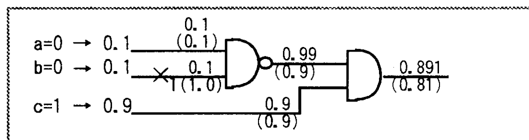

図３. コスト計算例

の入力パターンのコスト計算表であり、一番右側の列には真のテストパターンからの距離（入力パターンとテストパターンとの間で、同じビット位置で値が異なるビット数を数えたもの）を付記した。この表を見ると、コスト値が大きいものほど、距離が大きくなっていること、すなわち、テストパターンからより離れていることがわかる。つまり、上式がコスト関数式として適切であったことがわかる。

表１. コスト計算表と距離

外部入力パターン (a, b, c)	RVsimlの入力値 (a, b, c)	RVsimlの出力値 正常回路	RVsimlの出力値 故障回路	コスト $\frac{1}{\lvert 正 - 故 \rvert}$	テストデータからの距離
(0, 0, 0)	(0.1, 0.1, 0.1)	0.099	0.090	111.11	2
(0, 0, 1)	(0.1, 0.1, 0.9)	0.891	0.810	12.34	1
(0, 1, 0)	(0.1, 0.9, 0.1)	0.091	0.090	1000.00	3
(0, 1, 1)	(0.1, 0.9, 0.9)	0.819	0.810	111.11	2
(1, 0, 0)	(0.9, 0.1, 0.1)	0.091	0.010	12.34	1
(1, 0, 1)	(0.9, 0.1, 0.9)	0.819	0.090	1.37	0
(1, 1, 0)	(0.9, 0.9, 0.1)	0.019	0.010	111.11	2
(1, 1, 1)	(0.9, 0.9, 0.9)	0.171	0.090	12.34	1

RVsiml: 実数値シミュレーション　ε は仮に0.1とする。

３. 性能向上手法と各手法の評価

表２<A>に、２章で紹介した手法（原手法）によるISCAS85, ISCAS89の主なベンチマーク回路のテスト生成結果（残った未検出故障数とCPU時間）を示す。使用マシンは、DELL 466/M（OS：PC-UNIX, CPU：AMD DX4-100MHz, 62500dhrystone程度）である。この表２で上半分の"乱数フェーズあり"とは、簡単に見つかる故障を速く見つけるために、乱数入力パターンの生成＋故障シミュレーションを前フェーズにいれたものである。乱数フェーズは、乱数入力パターンが256個連続して新たな故障を検出しなくなったら終了し、逐次改善法に移る。そのCPU時間は、逐次改善法とその故障シミュレーション部に要した時間であり、乱数フェーズの時間は入っていない。また、表２の下半分は乱数フェーズを実行せず、すべての仮定故障に対して逐次改善法を適用したときの結果である。

表２. 高速化手法とタネの複数化の効果

	<A> 高速化なし ε =0.01		<B> 高速化あり ε =0.01		<C> 高速化あり ε =0.01タネ複数化 タネ数150で打切り		
	未検出故障数	CPU時間 (秒)	未検出故障数	CPU時間 (秒)	未検出故障数	CPU時間 (秒)	max seeds
乱数フェーズあり*							
c880	0	1	0	0	0	1	1
c1355	2	0	0	0	0	0	0
c1908	1	2	3	1	0	2	4
c2670	2	66	0	6	0	5	1
c3540	1	12	0	6	0	9	2
c5315	2	26	2	2	0	2	3
c6288	2	4	2	4	0	1	1
c7552	19	473	28	55	2	121	150
s420	1	1	4	0	0	0	5
s420.1	5	2	6	1	0	1	3
s641	2	2	3	0	0	1	2
s713	1	1	3	0	0	1	3
s820	1	1	0	0	0	0	1
s832	0	1	0	0	0	0	1
s838	51	21	70	5	3	72	150
s838.1	130	36	143	10	7	96	150
s953	0	2	2	0	0	0	1
s1196	5	16	5	7	0	7	9
s1238	7	18	8	7	0	10	10
s1423	0	5	2	1	0	3	5
s1488	0	1	0	0	0	0	1
s1494	0	1	0	0	0	0	1
s5378	2	385	5	22	0	24	4
s9234	67	3260	83	316	0	362	39
s13207	102	14085	75	1163	0	1611	55
s15850	114	9949	102	566	0	937	104
s35932	0	422	0	1	0	1	1
s38417	N.A.	N.A.	177	15133	1	17528	150
s38584	N.A.	N.A.	75	5791	0	5463	43
乱数フェーズなし							
c880	0	7	0	3	0	3	1
c1355	2	13	4	7	0	8	5
c1908	8	25	7	16	0	18	5
c2670	8	178	10	40	0	52	7
c3540	6	125	11	88	0	98	3
c5315	8	338	17	92	0	123	18
c6288	2	51	2	51	0	54	6
c7552	52	1200	47	228	0	265	48
s420	5	2	2	1	0	1	2
s420.1	5	4	4	2	0	2	3
s641	0	6	3	2	0	2	2
s713	4	7	3	2	0	2	3
s820	0	9	1	4	0	4	2
s832	0	9	1	4	0	4	1
s838	59	22	57	7	6	87	150
s838.1	138	46	129	12	7	118	150
s953	0	11	0	4	0	4	1
s1196	8	28	7	11	0	11	4
s1238	9	32	9	11	0	13	4
s1423	0	36	2	11	0	29	19
s1488	0	19	1	10	0	11	2
s1494	0	17	0	10	0	10	2
s5378	6	933	3	125	0	133	3
s9234	77	4843	63	806	0	871	44
s13207	69	15841	84	1806	0	2286	112
s15850	148	19549	125	2214	0	2578	40
s35932	0	6534	0	1336	0	1340	1
s38417	N.A.	N.A.	165	16345	1	18337	150
s38584	N.A.	N.A.	86	10530	0	10039	20

(*)乱数フェーズありのCPU時間は、逐次改善法の部分とその故障シミュレーションの部分に要した時間

3．1 計算量低減のための手法

表２<A>でCPU時間に着目すると、大規模なISCAS89回路（２万ゲートクラスのs38417やs38584）では24時間経過しても終了しない(N.A.)など、処理時間が大きいことが問題である。そこで、まず最初の課題を計算量の低減においた。

(1)有効ゲート

手続きctg0の目標は、正常回路と故障回路で、外部出力ピンの実数出力値の差をできるだけ大きくするような入力パターンを探すことである。２章で説明したコスト計算では、全ての外部出力ピンについて正常回路と故障回路で実数出力値を求め、その差の総和の逆数をコストとしている。しかし、故障箇所からfan-out方向に到達可能な外部出力ピン(fan-out到達可能外部出力ピン)以外の外部出力ピンでは、どんな入力パターンを入れても、正常回路と故障回路では同じ値が出力される。つまり、その差は常に０であり、それらの実数出力値の計算は無駄である。すなわち、図４のように、故障箇所からのfan-out到達可能外部出力ピン(有効外部出力ピン)の値のみを計算すればよい。そのためには、有効外部出力ピンからのfan-in到達可能外部入力ピン(有効外部入力ピン)に入力パターンを与え、有効外部出力ピンからのfan-in到達可能ゲート(有効ゲート)のみの計算をすればよい。

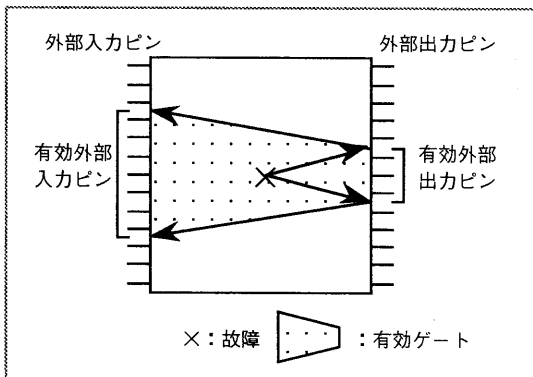

図４．有効ゲート

(2)変化ゲート

２章に示した逐次改善法ctg0では、改善に成功した時、次は新入力パターン（PI0）中の１ビットだけを変更し、その他のビットはすべて前のままの入力パターンでコスト計算が行われる。これは、変更した外部入力ピンからのfan-out到達可能ゲート以外のゲートの出力値は、前の入力パターン時の値と変わらないことを意味し、再計算する必要がない。但し、改善がなかった時は、変更ゲート値をもとに戻すための操作が必要であるが、この操作は次の外部入力ピンの変更による再計算と同時に行うことができる。（つまり、次のコスト計算時に２つの外部入力ピンからのfan-out到達可能ゲートの値を再計算する。）

図５は、コスト改善直後に、次の入力パターンのコストを得るのに再計算が必要なゲート(変化ゲート)を表している。(1)の有効ゲートの上に(2)の変化ゲートを重ね合わせた図である。

これら両方の計算量低減手法を適用した結果を表２<B>に示す。表２<A>の結果と比べると、比較的大きな回路では、５倍以上程度から２０倍に近い程度までの高速化がはかれており、s38417やs38584などの大規模な回路でも、実用的な時間内で終了するようになった。

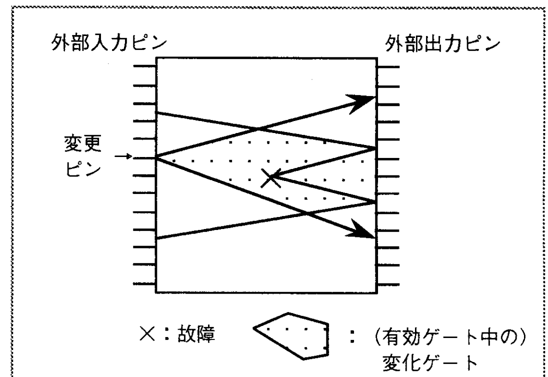

図５．変化ゲート（有効ゲートと組合せた結果）

3．2 検出率向上のための手法

次の課題は、すべての回路で未検出故障数を０にすることである。

(1)複数のタネによる逐次改善

２章に述べた逐次改善法ctg0は、乱数初期入力パターン（タネと呼ぶ）を１つ与え、それを改善していく手法であった。この方法では、図６(a)の概念的なグラフのように、局所的な極小値に落ちついてしまい、真のテストパターンまで到達しないことが考えられる。これに対し、いろいろな乱数初期入力パターン（複数のタネ）を使って逐次改善を行えば、図６(b)のようにテストパターンを発見する可能性が上がる。具体的には、手続きctg0の中を、１つのタネで失敗したら、別のタネで試してみるように変更する。

これによる実験結果を表２<C>に示す。この表中のmaxseedsは、与えたタネの数（故障毎に異なる）のうちの最大数である。この実験では、各故障に対して与えたタネが150個になってもテストパターンが見つからなかったなら、その故障の処理を打ち切り、次の故障に移る。この結果を見ると、与えるタネを複数にすることにより、ほとんどの回路では未検出故障数が０になり、しかも<B>に比べCPU時間はあまり増加しな

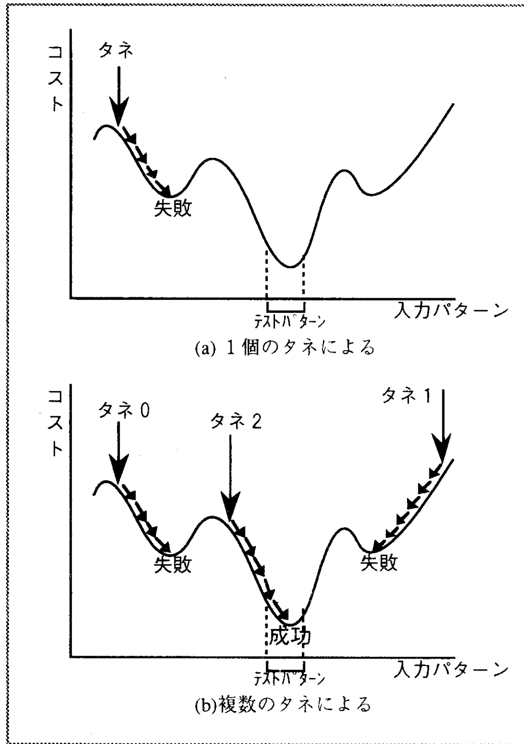

図6．複数のタネによる効果

いことがわかる。しかし、c7552、s838、s38417では未検出故障が残っており、c7552、s838ではCPU時間もかなりの割合で増加している。また、maxseeds、すなわち必要であったタネが、50個程度以上のものもある。maxseeds値が大きいことは、テストパターンが容易には見つからない故障が存在していることを示しており、100%検出が達成できない可能性があることを示唆している。

(2) ε 値の性能への影響

次に、ε 値の性能への影響をみるため、0.01以外に、0.2、0.1、0.001での測定を行った（表3<A><B><C><D>）。まず、明らかに、0.1や0.01の方が、0.2や0.001より優れている。ε 値が小さすぎる0.001では、実数値シミュレーション計算中に浮動小数点演算の桁落ちやアンダーフローが多く発生し、性能を落としている。次に、ε =0.1の場合をε =0.01の場合と比較する。0.1では、0.01で残っていたc7552、s838の未検出故障はなくなっているが、s38417は残っている。CPU時間では、s838で大きく改善されたが、c7552ほかいくつかの回路ではやや増加したものもある。maxseeds値では、一部改善がみられるが、まだ50個程度以上のものも多い。

(3)補正

[池田89]や[Rivin94]の論文にも指摘されているように、ε 値をあまり大きくとると、好ましくない計算が行われることがある。すなわち、ε 値が0.1の場合、8入力ANDゲートのすべての論理値が1の場合、論理出力値は1であるが、実数値シミュレーションの出力値は $(1.0-0.1)^8=0.4306$ になってしまう。論理出力値1の実数出力値が1.0からあまり離れるのは望ましくない。そこで、ε =0.1の場合に補正を試みることにした。すなわち、ANDゲートの補正は図7のグラフのように、ゲートの実数出力値を全体的に1．0の方に近づけてあげることを行う。同様な問題を持つ他のゲートについても同様に行う。

ε =0.1とし、図7の補正グラフのbを0.4にした補正付きの測定結果を表3<E>に示す。これによると補正の効果が表れており、未検出故障はすべてなくなり、CPU時間、maxseedsともに、ほぼ、補正無し0.1、0.01よりすぐれた結果が得られている。すなわち、ε =0.1で補正付きにすることで、未検出故障がすべてなくなり、100%故障検出ができるようになったといえる。

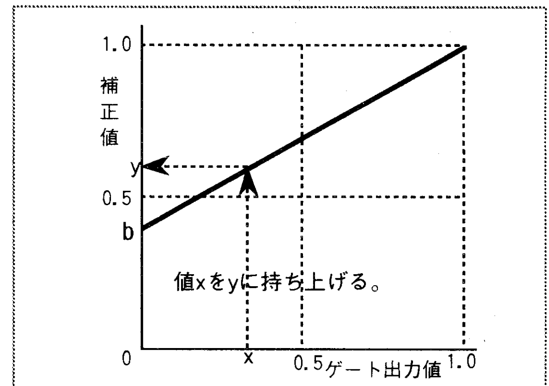

図7．ANDゲートの補正関数

4．他方式との比較評価

表4に原手法や他のシミュレーションベースの方式の"乱数フェーズなし"での比較結果を示す。"原手法"は、実数値シミュレーション法の原手法[池田89]を追実験したものである（表2<A>の一部の再掲）。"Rivin-Chakradhar"[Rivin94]の方式は、初期入力パターンとしてすべての外部入力ピンに0.5を与え、コスト計算に本方式とやや類似の実数計算を用い、一種の反復改善法により、各外部入力ピン値を徐々に0または1に近づけていく手法であり、データは該論文からとった。"Rivin-Chakradhar"の使用マシンはSilicon GraphicsのIndigoワークステーションである。"原手法"、"Rivin-Chakradhar"の手法は共に、未検出故障数が残っている回路が多く、CPU時間もかなり大きいのに対し、我々の手法ではすべて未検出故障はなくなり、かつ、

表3．ε値の影響と補正の効果

	<A>			<B>			<C>			<D>			<E>		
	ε =0.2 タネ複数化 タネ数150で打ち切り			ε =0.1 タネ複数化 タネ数150で打ち切り			ε =0.01 タネ複数化 タネ数150で打ち切り			ε =0.001 タネ複数化 タネ数150で打ち切り			ε =0.1 補正付き タネ複数化 タネ数150で打ち切り		
	未検出故障数	CPU時間(秒)	max seeds	未検出故障数	CPU時間(秒)	max seeds	未検出故障数	CPU時間(秒)	max seeds	未検出故障数	CPU時間(秒)	max seeds	未検出故障数	CPU時間(秒)	max seeds
乱数フェーズあり*															
c880	0	0	1	0	0	1	0	1	1	0	1	4	0	1	1
c1355	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c1908	0	1	2	0	2	2	0	2	4	0	2	5	0	2	4
c2670	0	5	1	0	5	1	0	5	1	0	6	1	0	6	1
c3540	0	9	3	0	7	1	0	9	2	0	15	7	0	12	3
c5315	0	97	145	0	15	20	0	2	3	0	4	3	0	6	4
c6288	0	3	2	0	5	3	0	1	1	0	5	3	0	2	1
c7552	0	1020	106	0	307	51	2	121	150	13	859	150	0	53	3
s420	0	0	1	0	0	1	0	0	5	0	2	42	0	0	1
s420.1	0	1	1	0	1	1	0	1	3	0	2	33	0	1	1
s641	0	0	1	0	0	1	0	1	2	0	1	10	0	0	1
s713	0	0	1	0	0	1	0	1	3	0	2	13	0	0	1
s820	0	0	1	0	0	1	0	0	1	0	0	2	0	0	1
s832	0	0	1	0	0	1	0	0	1	0	0	2	0	0	1
s838	0	3	1	0	4	5	3	72	150	42	317	150	0	5	6
s838.1	0	7	1	0	8	9	7	96	150	91	521	150	0	9	4
s953	0	1	1	0	0	1	0	0	1	0	1	7	0	0	1
s1196	0	8	14	0	7	5	0	7	9	0	9	9	0	9	9
s1238	0	7	5	0	9	9	0	10	10	0	11	18	0	10	2
s1423	0	9	10	0	57	120	0	3	5	0	5	4	0	4	4
s1488	0	0	1	0	0	2	0	0	1	0	0	2	0	0	1
s1494	0	0	1	0	0	1	0	0	1	0	0	1	0	0	2
s5378	0	21	4	0	23	3	0	24	4	0	27	5	0	22	3
s9234	0	361	67	0	310	12	0	362	39	0	723	76	0	316	21
s13207	0	1140	20	0	1368	44	0	1611	55	0	2459	92	0	1144	53
s15850	0	604	5	0	585	8	0	937	104	6	2118	150	0	836	9
s35932	0	1	1	0	1	1	0	1	1	0	1	1	0	1	3
s38417	11	24677	150	2	18738	150	1	17528	150	1	17727	150	0	16253	60
s38584	0	6578	71	0	5459	6	0	5463	43	0	6488	73	0	5192	32
乱数フェーズなし															
c880	0	3	1	0	3	1	0	3	1	0	3	3	0	4	1
c1355	0	29	25	0	6	2	0	8	5	0	9	6	0	17	11
c1908	0	16	2	0	17	3	0	18	5	0	18	4	0	19	6
c2670	0	44	4	0	41	6	0	52	7	0	49	13	0	60	9
c3540	0	96	6	0	98	26	0	98	3	0	98	8	0	113	7
c5315	0	216	122	0	212	64	0	123	18	0	100	7	0	114	12
c6288	0	57	6	0	49	5	0	54	6	0	46	2	0	76	13
c7552	2	1373	150	0	581	60	0	265	48	13	1013	150	0	279	6
s420	0	1	1	0	1	1	0	1	2	0	2	31	0	1	1
s420.1	0	1	1	0	2	1	0	2	3	0	3	24	0	2	1
s641	0	2	1	0	2	1	0	2	2	0	4	34	0	2	1
s713	0	2	1	0	2	1	0	2	3	0	3	14	0	2	1
s820	0	4	2	0	4	1	0	4	2	0	4	2	0	4	1
s832	0	4	1	0	4	1	0	4	1	0	4	2	0	4	1
s838	0	6	1	0	7	6	6	87	150	43	328	150	0	9	8
s838.1	0	11	1	0	11	6	7	118	150	88	526	150	0	13	7
s953	0	4	1	0	4	1	0	4	1	0	4	5	0	4	1
s1196	0	12	13	0	11	8	0	11	4	0	12	4	0	15	25
s1238	0	13	18	0	12	3	0	13	4	0	14	15	0	16	15
s1423	0	90	3	0	14	6	0	29	19	0	11	3	0	17	4
s1488	0	10	2	0	10	2	0	11	2	0	10	2	0	10	1
s1494	0	10	1	0	10	2	0	10	2	0	10	3	0	10	1
s5378	0	132	4	0	132	2	0	133	3	0	139	9	0	140	3
s9234	1	1050	150	0	862	49	0	871	44	0	1259	83	0	800	15
s13207	0	1727	8	0	1839	20	0	2286	112	0	3257	129	0	1841	47
s15850	0	2666	30	0	2274	6	0	2578	40	3	4351	150	0	2737	11
s35932	0	1244	1	0	1314	1	0	1340	1	0	1360	1	0	1333	7
s38417	8	23906	150	3	19493	150	1	18337	150	1	20663	150	0	18813	69
s38584	0	11059	46	0	10547	11	0	10039	20	0	11239	88	0	10592	12

(*)乱数フェーズありのCPU時間は、逐次改善法の部分とその故障シミュレーションの部分に要した時間

表4．原手法や他方式との性能比較

	原手法 ε=0.01 [池田89]の追実験		Rivin-Chakradhar [Rivin94]より		本手法 ε=0.1補正	
	未検出故障数	CPU(1)時間(秒)	未検出故障数	CPU(2)時間(秒)	未検出故障数	CPU(1)時間(秒)
乱数フェーズなし						
c880	0	7	20	7	0	4
c1355	2	13	12	39	0	17
c1908	8	25	20	74	0	19
c2670	8	178	7	3614	0	60
c3540	6	125	23	1250	0	113
c5315	8	338	6	320	0	114
c6288	2	51	0	1100	0	76
c7552	52	1200	267	3600	0	279
s420	5	2	21		0	1
s420.1	5	4	-		0	2
s641	0	6	2		0	2
s713	4	7	2		0	2
s820	0	9	9		0	4
s832	0	9	14		0	4
s838	59	22	20	不	0	9
s838.1	138	46	-		0	13
s953	0	11	-		0	4
s1196	8	28	41		0	15
s1238	9	32	24		0	16
s1423	0	36	24		0	17
s1488	0	19	16	明	0	10
s1494	0	17	14		0	10
s5378	6	933	80		0	140
s9234	77	4843	654		0	800
s13207	69	15841	357		0	1841
s15850	148	19549	149		0	2737
s35932	0	6534	0		0	1333
s38417	N.A.	N.A.	2007		0	18813
s38584	N.A.	N.A.	699		0	10592

(1)DELL 466/M（CPUはAMD DX4-100MHz）
(2)Silicon GraphicsのIndigoワークステーション

より高速である。

5．結論及び今後の課題

シミュレーションベースのテスト生成法の一方式である実数値シミュレーション法において、"実数値シミュレーション"の計算量を減らすために、①故障箇所に対する有効ゲート以外の計算の排除、②値が変化するゲート以外の計算の排除などの手法を導入した。また、冗長故障を除く故障の100%検出のために、③複数の乱数初期入力パターンによる逐次改善、④"実数値シミュレーション"の各ゲート出力値の補正、などの手法を導入した。その結果、当面の具体的な目標として設定した「実数値シミュレーション法によって、ISCAS85、組合せ回路としてのISCAS89のすべてのベンチマーク回路で、冗長故障を除く故障を、実用的な時間内で100%検出すること」が達成できた。すなわち、実数値シミュレーション法の基本能力を飛躍的に向上させることができ、今後の大規模かつ複雑な順序回路を扱うための基礎固めができた。今後は、本研究で得られた技術を基礎にして、当初の目標である大規模順序回路を対象としたテスト生成法に取り組む。

[参考文献]

[池田89]. 池田光二、畠山一実、林照峯："バックトラック処理不要な組合せ回路テスト生成手法"、信学技報FTS89-35、1989年10月

[彦根92]. 彦根和文、池田光二、畠山一実、林照峯："実数値シミュレーションを利用した順序回路テスト生成"、電子情報通信学会論文誌D-I Vol. J75-D-I No.11 、1992年11月

[Hatayama92]. Kazumi Hatayama, Kazunori Hikone, Mitsuji Ikeda, Terumine Hayashi："Sequential Test Generation Based on Real-Valued Logic Simulation," IEEE Int'l Test Conference 1992

[伊達94]. 伊達博、中尾教伸、畠山一実："DESCARTES：順序回路を対象とした実数値シミュレーションに基づく並列テスト生成システム"、Workshop on Parallel and Distributed LSI-CAD、Int'l Symp. on Fifth Generation Computer Systems 1994、1994年12月

[Rivin94]. Igor Rivin, Srimat T. Chakradhar："Discrete Test Generation by Continuous Method," 12th IEEE VLSI Test Symposium 94, 1994

[Agrawal89]. Vishwani D. Agrawal, Kwang-Ting Cheng, Prathima Agrawal："A Directed Search Method for Test Generation Using a Concurrent Simulator," IEEE Trans. Computer-Aided Design, Vol. 8, No.2, Feb.1989

[Fujiwara83]. Hideo Fujiwara, Takeshi Shimono："On the Acceleration of Test Generation Algorithms," IEEE Trans. Computers, Vol. c-32, No.12, Dec.1983

[Schulz88]. Michael H. Schulz, Erwin Trischler, Thomas M. Sarfert："SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," IEEE Trans. Computer-Aided Design, Vol. 7, No.1, Jan.1988

[Teramoto93]. Mitsuo Teramoto："A Method for Reducing the Search Space in Test Pattern Generation," IEEE Int'l Test Conference 1993

[Cheng89]. W. Cheng, S. Davidson："Sequential Circuit Test Generator (STG) Benchmark Results," Proc. Int'l Symp. Circuits and Systems '89, 1989

[Auth91]. Elisabeth Auth, Michael H. Schulz："A Test-Pattern-Generation Algorithm for Sequential Circuits," IEEE Desighn & Test of Computer, Vol. 8, No.2, June 1991

[西川82]. 西川、三宮、茨木："最適化", 岩波講座情報科学-19, 岩波書店、1982

[Brglez85]. F. Brglez and H. Fujiwara："A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," Proc. Int'l Symp. Circuits and Systems '85, 1985

[Brglez89]. F. Brglez, D. Bryan, K. Kozminski："Combinational Profiles of Sequential Benchmark Circuits," Proc. Int'l Symp. Circuits and Systems '89, 1989