設計自動化 62－29
（1992. 5.28）

多分決定グラフ表現を用いた高速な故障シミュレーション手法

下迫田義則　石浦菜岐佐　白川 功
大阪大学工学部 情報システム工学科

あらまし　本稿では，多分決定グラフ (Multi Decision Diagram) 表現を用いた順序回路の高速な故障シミュレーション手法を提案する．本手法は PVL(Parallel Value List) 法を改良したものであり，線形リストの代わりに多分決定グラフを用いることによって，ゲート評価，故障の挿入，故障のドロップの処理が高速に行なえるようにしたものである．ISCAS'89 ベンチマーク順序回路に対し実験を行なった結果，PVL 法に比べ最高で 1.6 倍の高速化が達成された．

A Fast Fault Simulation Method Using Multi Decision Diagram Representation

Yoshinori Shimosakoda, Nagisa Ishiura and Isao Shirakawa
Department of Information Systems Engineering,
Faculty of Engineering, Osaka University

Abstract This paper presents a fast fault simulation method for sequential logic circuits using Multi Decision Diagram representation. This method is an improvement of a PVL(Parallel Value List) fault simulation method, and the operations of gate evaluation, fault insertion and fault dropping are and accelerated by Multi Decision Diagram instead of linear lists. The experimental results on ISCAS'89 sequential benchmark circuits show that new method is 1.6 times faster than the conventional PVL method in the best case.

1 はじめに

近年のVLSI技術の発展により，VLSIはより益々広範な応用に使用されるようになっており，故障検査による回路の信頼性確保が重要なものとなっている．故障シミュレーションは，故障存在下での論理回路の動作をシミュレーションすることによってテストパターンの故障検出率の評価，テストパターンの生成，故障辞書の作成を行なうものであり，論理回路の故障検査に不可欠なものである．論理シミュレーションが正常な論理回路だけをシミュレーションするのに対し，故障シミュレーションでは仮定された多数の故障についてシミュレーションを行なわなければならない．そのため，回路規模の増大による計算時間，記憶量の増大が大きな問題となっており，これを克服すべく多くの研究[1, 2, 3, 4, 5, 6, 7, 8, 9]がなされている．組合せ回路においては種々の高速化手法が成果を上げている[7, 8]が，これらをそのまま順序回路に適応することは難しく，順序回路の高速な故障シミュレーション手法の開発が期待されている．

そこで本稿では，順序回路に対する高速な故障シミュレーション手法を提案する．本手法は，故障の伝播の特性(正常値依存性)を利用し，多分決定グラフ(Multi Decision Diagram)を用いて処理の高速化を実現した手法である．本手法をC言語を用いて実現し，ISCAS89ベンチマーク順序回路[10]に対して行なった結果，PVL法に比べて最高1.6倍の高速化が図られた．

以下，2章では故障シミュレーションについて，3章では本手法のデータ構造，ゲート評価法，故障の挿入・ドロップの方法について説明する．4章では実験結果を示す．5章では結論と今後の課題について述べる．

2 故障シミュレーション

故障シミュレーションは，論理回路とその回路に発生しうる故障の集合とテストパターン(外部入力に対する入力ベクトルの系列)を入力として，それぞれの故障について回路のシミュレーションを行ない，その故障が与えられたパターンで検出できるか否かを識別するものである．その結果，テストパターンの故障検出率が算出される．一般に故障シミュレーションはテスト生成の一過程として用いられる．

故障のモデルとしては，一般に単一縮退故障(single stuck-at-fault)モデルが採用されている．単一縮退故障とは回路中でただ1つの信号線において信号値が0あるいは1に固定してしまう故障のことをいう．このモデルは扱いが容易であり，かつ実際に起こる物理的故障(physical failure)の多くがこのモデルに帰着されることが知られている[13]．

通常，故障シミュレーションの入力として仮定される故障の集合は全信号線の単一縮退故障である．この中には回路の結線関係から等価(どんな入力に対しても同じ振舞いをする故障)と判定できるものがあり，これらを1つの代表故障にまとめることにより，対象故障数の削減が図られる．この処理は等価故障解析やfault collapsingと呼ばれる．また，故障検出率を算出する場合，すでに検出された故障に対するシミュレーションを続ける必要はない．一度検出された故障を以後のシミュレーションの対象から除外することを故障のドロップ(fault drop)という．故障のドロップの効果は大きく，これを行なうことは必須であると考えられる．

信号値のモデルは組合せ回路では2値(0, 1)論理で十分であるが，順序回路に対しては記憶素子の初期値として不確定値(以後Xと表記する)が必要なので3値(0, 1, X)論理が用いられる．

以下ではいくつかの既存の故障シミュレーション手法をあげ，その手法の特徴を述べる．

2.1 既存の故障シミュレーション手法

代表的な故障シミュレーション手法には，演繹法[1]，コンカレント法[2]，パラレル法[3]がある．

演繹法とコンカレント法は共に，故障回路の正常回路と異なる部分だけをシミュレーションすることにより，計算量を削減している手法である．演繹法は信号線で検出できる故障の集合を持ち，集合演算を用いてゲート評価を行なう．コンカレント法はゲートごとに正常回路と異なる故障の信号値だけを線形リストで持ち，イベント駆動方式によってシミュレーションを行なう．これらの手法は，共に信号線またはゲートごとに大きなリストを持つため，必要記憶量が大きくなるという問題がある．また，ゲート評価のたびにこのリストを走査するため，計算の効率は必ずしも良いとはいえない．

一方，パラレル法はコンピュータの1ワードに対するビット並列演算を利用し，同時に複数の故障に対する回路のシミュレーションを行なうものである．パラレル法では冗長な計算(正常回路と同じ計算)を何度も行なうため計算量が大きくなるという問題点があった．

従来は計算量や取り扱い易さの点からコンカレント法が一般的であったが，パラレル法を改良したPVL法[4, 5]やPROOFS[6]が記憶量や速度の点で注目されるようになってきた．

PVL法は各信号線において，仮定された故障が発生した時の信号値を図1のようなデータ構造を用いて表現するものである．故障はk個(この例では$k=4$)ごとにグループ化され，グループごとにまとめて扱われる(ビット並列演算を利用するため，一般にはk="1ワードのビット数"が用いられる)．すべての故障グループの中から正常回路と異なる故障を持つグループだけが選ばれ，線形リストで表現される．このデータ構造により，演繹法のように正常回路と同じ信号値をとる故障に対する冗長な計算を避けることができ，かつパラレル法のビット並列演算を生かした高速な計算が可能になる．

パラレル法を改良したもう一つの手法であるPROOFSはパラレル法とSFP(Single Fault Propagation)[7]を組み合わせた手法であり，現在順序回路に対してはPVL法と共に最も高速な手法の1つであると考えられている．

図1: PVL法の故障リスト

3 本手法

本稿で提案する故障シミュレーション手法は，PVL法の線形リストの代わりに多分決定グラフ(Multi Decision Diagram: MDD)表現を用いるものである．本手法は，順序回路を扱うため3値論理を用い，故障のモデルとして単一縮退故障を用いて，故障のドロップを行なうことができる．

3.1 多分決定グラフのデータ構造

図2は図1の線形リストを等価な多分決定グラフ(以後MDDとする)で表したものである．MDDは非巡回の有向グラフである．非終端節点(nonterminal node)からは最高m本(図の例では$m=4$)の枝(edge)が出ており，終端節

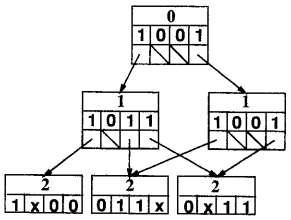

図 2: MDD のデータ構造

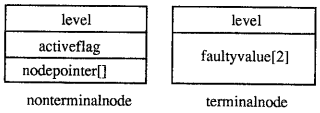

図 3: MDD の節点のデータ構造

点 (terminal node) は n 個 (例では $n=4$) の信号値を持っている．各節点にはレベル (level) が定義されている．どこからも枝がない節点のレベルは 0 であり，節点 u から節点 v に枝がある時，$\mathrm{level}(v) = \mathrm{level}(u)+1$ と定義される (但し，level() は節点のレベルを表す)．各節点は図 3のようなデータ構造で表現することができる．非終端節点は，処理の効率を良くするために m 個の activeflag を持っている．これはその位置に対応する枝の存在を示すフラグであり，枝がある時は 1，ない時は 0 になっている．実際には，activeflag は 1 ワードに詰め込み，信号値 0,1, X は 2 ビットで符号化すれば，効率よく表現できる．

MDD において全故障数を x とする時，最大レベル (以後 ML とする) は $\lceil \log_m(x/n) \rceil$ と定まる．MDD を構成するには，図 4のようにまず各々の故障に対する信号値を n 個ごとのグループに分けた後，グループを m 個ごとにグループ化するという操作を $ML-1$ 回繰り返すことにより，階層的なグループ化を行なう．次に階層の上位から 0，1，2 の順にレベルを割り当てる．MDD は階層的なグループ化をした故障をグラフ的に表現したものである．MDD のレベル i の節点はレベル i のグループに対応する．節点 u に対応するグループが節点 $v_1, v_2, \cdots, v_m$ に対応するグループより構成される時，u から $v_1, v_2, \cdots, v_m$ に有向枝がはられる．但し，次の 2 つによって，グラフの大きさの削減が図られている．

1. v_i に対応するグループの故障のすべての信号値が正常値と等しい場合には，u から v_i への枝は張られない．
2. u と v の信号値が全て等しいならば，u を削除して v に枝をつなぐ．

各故障 f は ML 個のグループに (階層的に) 属している．今，f がレベル $l(0 \le l \le ML-1)$ のグループ内では $p_l(0 \le p_l \le m-1)$ 番目のグループに属しており，レベル ML のグループ内では $p_{ML}(0 \le p_{ML} \le n-1)$ 番目の故障であったとすると，f は $[p_0, p_1, \cdots, p_{ML}]$ によって一意に識別できる．$f = [p_0, p_1, \cdots, p_{ML}]$ と表される時，f は全体では $q([p_0, p_1, \cdots, p_{ML}]) \equiv \sum_{i=0}^{ML-1} p_i n m^{ML-i-1} + p_{ML}$ 番目の故障である．すなわち，レベル 0 の節点から始めて $p_0, p_1, \cdots, p_{ML-1}$ 番目の枝を次々に辿って着く終端節点の p_{ML} 番目の信号値は，全体で $q([p_0, p_1, \cdots, p_{ML}])$ 番目の故障が発生した時の信号値を表している．

PVL 法では，グループ番号 g のビット位置 $p(0 \le p \le k-1)$ の故障の故障番号は $gk+p$ と表される．例えば図 1の group12 の 3 番目 (最も左を 0 番目とする) の (信号値が X の) 故障の故障番号は $51(= 12 \cdot 4 + 3)$ である．これは図 2の MDD では [3,0,3] の故障に対応する故障番号も $q([p_0, p_1, \cdots, p_{ML}]) = 3 \cdot 4 \cdot 4^{2-0-1} + 0 \cdot 4 \cdot 4^{2-1-1} + 3 = 51$ であり，それぞれの故障は対応している．同様に図 1の group2 の 0 番目，group15 の 1 番目の故障は，それぞれ MDD における [0,2,0], [3,3,1] の故障に対応している．

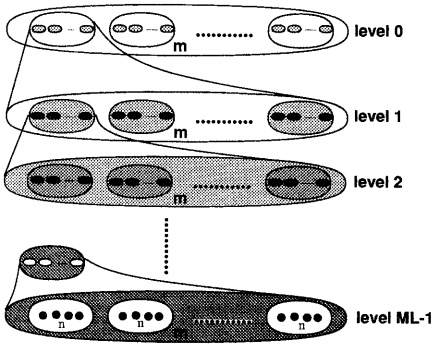

図 4: 階層的なグループ化

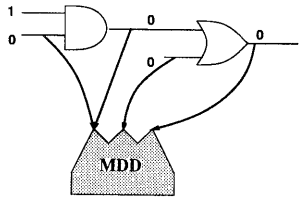

図 5: MDD におけるデータの共有の例

本手法では共有二分決定グラフ (SBDD)[11] と同様に，MDD 中に等価な節点 (同じ内容の節点を表す) はただ一つしか存在しないという制約を加える．故に，演算中に新しい節点を生成する際，等価な節点が存在していれば，その節点を共有することにより，新しい節点を作らないようにしている．等価な節点が存在するか否かの判定は，すべての節点をハッシュテーブルに登録，参照することにより行なう．この制約のもとでは，MDD の等価判定は MDD の節点を指すポインタ (すなわち枝) を比較するだけでよく，定数時間で行なうことができ，図 5のように MDD のすべてがそのまま伝播した時 (後述) やその一部が伝播した時などデータの共有ができ，記憶量の削減にも大きな効果をもつ．

3.2 MDD を用いた高速処理の基本的アイディア

ある信号線において，故障回路の値が正常回路の値と異なる時，その故障はその信号線に伝播している (あるいは単に，ある) という．あるゲートの入力側の信号線に伝播している故障が出力側の信号線にも伝播している時，その故障はそのゲートの入力から出力に伝播したという．

ここで例として 2 入力 AND ゲートを考えてみる．今，正常入力値が (a, b) = (1, 0) とすると，正常出力値は 0 となる．この時入力信号線 a だけにある故障は出力側に伝播しない．なぜなら，a にある故障は出力でその信号値が 0 となり正常値と等しくなるためである．また，b だけにある故障はその値のまま伝播する．a, b に共にある故障に関し

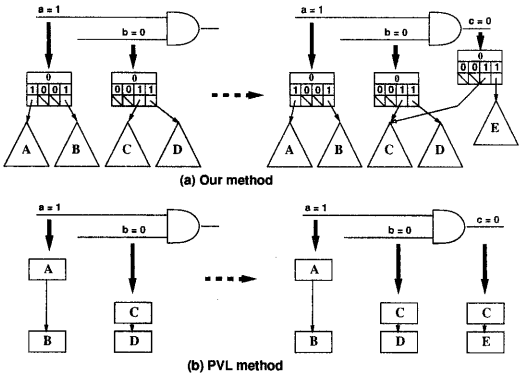

図 6: 本手法と PVL 法との比較

ては伝播する可能性があるのでそれらの 2 つの故障信号値を用いて評価した結果と正常値を比べ，伝播するか否か決定する必要がある．つまり，2 入力 AND ゲートにおいては，故障の伝播は他方の入力の正常値に依存している．故障は他方の正常値が 0 ならば伝播せず，1 ならばその故障信号値のまま伝播する．

本手法ではこのような故障の伝播の正常値依存性を利用し，高速化を図るために MDD を用いる．図 6の例を用いて，本手法と PVL 法を比較してみる．図 6中の A, B, C, D はそれぞれ故障のグループのグループであり，本手法では MDD で，PVL 法では線形リストで表現されている．

1. レベル 0 のグループ 0 に属している故障 (つまり，A) は，決して伝播しない (信号線 b の正常値が 0 で b にはそのグループの故障がないため)．この判定は b のグループ 0 のポインタを見るだけで行なえる．これに対して，PVL 法では少なくとも A のリストを走査しなければならない．
2. レベル 0 のグループ 1 に属している故障は両方の信号線にないので，伝播しない．この判定は信号線 a, b の対応するグループのポインタを見るだけで行なえる．
3. レベル 0 のグループ 2 に属している故障 (つまり，C) は，そのまま伝播する．本手法ではこの故障の伝播はポインタをつなぐだけで実現できる．PVL 法では，リスト C を走査し C をコピーする必要がある．
4. レベル 0 のグループ 3 に属している故障 (つまり，B,D) に対しては，再帰的処理により出力に伝播する故障を計算する．最悪の場合の計算量は PVL 法と同じになるが，下位の階層で上記の 1，3 を適用できれば計算量が減少する可能性がある．

すなわち，本手法の高速化のポイントは 1 と 3 である．

3.3 ゲート評価のアルゴリズム

本手法では，前述のような故障伝播の正常値依存性を用いてゲート評価を行なう．2 入力 AND ゲートの評価のアルゴリズムを図 7に示す．この関数 mdd_and() の引数は，a,b は MDD の節点を指すポインタ，gvalA, gvalB はそれぞれの正常値，level は評価をしているレベルを示す (すなわち，最初は level=0 として呼び出される)．

1. 以下最初の 4 行は，一方の入力に故障信号値リスト MDD がない場合の処理である．その信号線の正常値が 0 ならば NULL を返し (3.2 節の 1 に対応)，1 ならば他方の節点を指すポインタを返す (3.2 節の 3 に対応)．2. 以下の

```
NODE mdd_and(NODE *a, *b, int gvalA, gvalB, level);
{
  NODE tmpnode, tmpA, tmpB;
  int fvalA, fvalB, w;

1. if (a == NULL && gvalA == 1) return b;
   else if (b == NULL && gvalB == 1) return a;
   else if (a == NULL && gvalA == 0) return NULL;
   else if (b == NULL && gvalB == 0) return NULL;
2. else if (level == maxlevel) {
     if (a == NULL) fvalA = gvalA; else fvalB = a→fval;
     if (b == NULL) fvalA = gvalB; else fvalB = b→fval;
     tmpnode.fval = value_and(fvalA, fvalB);
3.   if (all faultyvalues of tmpnode are equal to good value) return NULL;
4.   else return get_node(tmpnode);
5. } else {
     if (a == NULL) tmpnode.aflag = b→aflag;
     else if (b == NULL) tmpnode.aflag = a→aflag;
     else {
6.     switch (gvalA, gvalB) {
         case (0, 0) tmpnode.aflag = a→aflag & b→aflag;
         case (1, 0), (X, 0) tmpnode.aflag = a→aflag;
         case (0, 1), (0, X) tmpnode.aflag = b→aflag;
         otherwise tmpnode.aflag = a→aflag | b→aflag;
       }
     }
7.   for (w = tmpnode.aflag, p = top[w]; p != 0; w = next[w]) {
       if (a == NULL) tmpA = NULL; else tmpA = a→nptr[p];
       if (b == NULL) tmpB = NULL; else tmpB = b→nptr[p];
       tmpnode.nptr[p] = mdd_and(tmpA, tmpB, gvalA, gvalB, level+1);
8.     if (tmpnode.nptr[p] == NULL)
         set p th bit of tmpnode.aflag to 0;
     }
9.   if (tmpnode.aflag == 0) return NULL;
10.  else return get_node(tmpnode);
  } }
```

図 7: 2 入力 AND ゲートの評価アルゴリズム

6 行は，終端節点に対する処理であり，両方の faultyvalue を引数とし value_and() を用いて出力の faultyvalue を計算する．但し，ポインタが NULL の時は正常値を用いて計算する．この計算はビット並列演算で行なう．もし，その出力 (tmpnode) の faultyvalue の全てが正常値と等しい場合 NULL を返す (3.)．そうでなければ，求めた節点 (tmpnode) を引数として get_node() を呼び出す (4.)．get_node() はハッシュテーブルを参照し，等しいものがあればその節点を指すポインタを返し，なければ求めた節点をハッシュテーブルに登録し，そのポインタを返す関数である．5. 以下の行は再帰的に処理を行なう部分である．まず，故障伝播の正常値依存性を用いて tmpnode の activeflag を定める．これはすなわち，伝播する可能性のない故障グループを識別し，処理の対象から除外することにも対応する．故障の伝播の様子を図 8にまとめる．図 8において 2 つの円はそれぞれの入力 (左は上側，右は下側の入力に対応) に伝播している故障の集合を表している．濃い影の部分は値を計算する必要があり，薄い影の部分はそのまま値を伝播し，その他空白部は伝播しないことを示している．出力側に故障が伝播しうるのは，両方の影の部分のみであるので，これらのグループについてのみ故障伝播の計算を行なえば良いことになる．6. の switch 文は図 8における両方の影の部分だけを残すように activeflag を定めている．"&" はビットごとの論理積，"|" はビットごとの論理和を表している．7. の for ループ文では activeflag が 1 であるビットに対応するグループのみについて mdd_and() を再帰的に呼び出し，1 つ下位のレベルの処理を行なう．ここでは top[], next[] という配列を用いて，activeflag が 1 であるポインタを高速に検索している．mdd_and() から NULL を返されれば，それに対応する activeflag のビットを 0 にする (8.)．for ループ文を抜けた後 activeflag の全てのビットが 0 であれば NULL を返し (9.)，そうでなければ終端節点の時と同様に get_node() を用いた後のポインタを返す (10.)．

図 9に 2 入力 AND ゲートの評価の例を示す．この例で

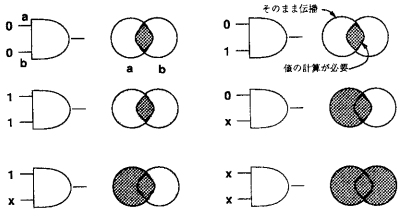

図 8: 故障の伝播

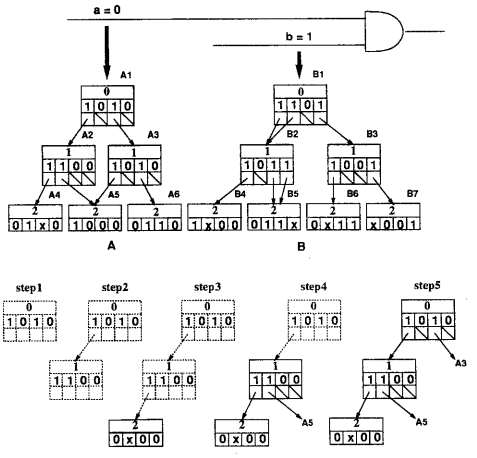

図 9: AND ゲートの評価の例

は，$m = n = 4$ としている．AND ゲートの入力信号線 a, b の正常値はそれぞれ 0，1 となっており，図 9に示した入力側の MDD から出力側の MDD を作成する場合を考える．正常値がこのような場合，前述したように信号線 b だけにある故障は伝播せず，信号線 a だけにある故障は伝播する．故に，基本的には MDD A の故障を出力側に伝播させればよいが，故障が a, b 共にある場合を考えながら次のような操作を行なっていく．

step1: MDD A の節点 (A1) の activeflag を仮節点の activeflag とし，その activeflag が 1 である位置の両方 (B がなければ A だけ) の MDD の nodepointer の指している節点 (A2, B2) を探索する．

step2: step1 と同様の操作を繰り返す．

step3: 節点 (A4, B4) のレベルが最下位なので faultyvalue を計算する．この時，全ての故障信号値が正常値 (この場合は 0) と等しければ，新しい節点は作られず，その上位の節点の activeflag の対応するビットを 0 にする．

step4: MDD B には対応する節点がないのでこの nodepointer は MDD A の対応する節点 (A5) を指す．

step5: step4 と同様の操作を行なう．

また，各節点はデータを共有するためにそれ以下の節点の操作が全て終了した時点でハッシュテーブルを見て，等しいものがあれば上からの nodepointer がそれを指すようにする．

前述したように本手法では，同じ節点は 1 つしか存在しないので，ゲート評価などの際，ある節点とある節点との演算結果をハッシュテーブルに登録しておけば，その後同じ節点を評価しなければならない時，それを参照するだけで評価可能であり，演算をする必要がなくなる．故障シミュレーションでは順序回路において，同じ評価をすることは頻繁に起こる．特に，正常信号値に変化がなく故障のみが伝播する場合には，変化のない部分の計算の繰り返しを避けることができるため,, この処理はシミュレーションの高速化に有効である．

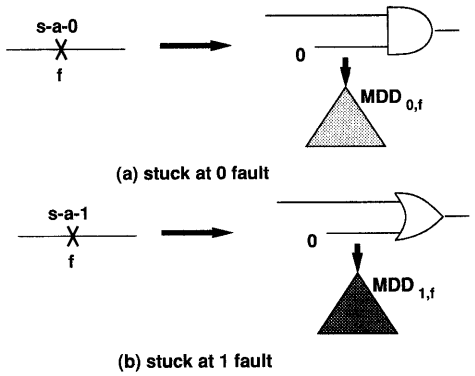

図 10: 故障挿入のモデル化

3.4 故障の挿入とドロップ

本手法における故障挿入の方法を示す．故障の挿入は図 10に示すようなモデル化に基づいて，ゲート評価に帰着することにより実現する．縮退 0 故障は図 10(a) のように，仮想的な AND ゲートを用いてモデル化される．$MDD_{0,f}$ は故障 f に対応する信号値のみ 0 で残りは 1 であるような信号値リストを表す MDD である．故障の挿入はこの AND ゲートを評価することにより実現される．同様に，縮退 1 故障の場合は図 10(b) のように，故障 f に対応する信号値のみ 1 で残りは 0 であるような $MDD_{0,f}$ との OR を計算することにより実現される．

故障のドロップに関しては，検出された故障に対応する信号値が 0，未検出故障に対応する信号値が 1 であるような MDD を用意し，ゲート評価終了時に AND のゲート評価を行なうことにより実現する．

3.5 故障のグループ化

故障のグループ化は本手法において重要な問題である．なぜなら，同じ信号線に同じ影響を与えるような故障を同じグループ内に割り当てることができれば，ビット並列演算の効率向上が期待できる．一般に故障のグループ化手法としては，故障箇所 (信号線) を節点とし，回路の結線関係と同様に枝をつなぐようなグラフを作り，外部出力からの深さ優先探索でグループ化を行なう手法が知られている [6]．そこで本手法では [9] で提案されたグループ化手法を階層的に用いる．この手法は深さ優先探索によるグループ化手法を，同じグループに含まれる 2 点間のグラフにおける距離を平均的に小さくなるように改良したものであり，ビット並列演算の並列度をより大きくすることを狙ったものである．

4 実験結果

本稿で提案した故障シミュレーション手法を C 言語を用いて実現し，ISCAS'89 ベンチマーク順序回路 [10] について実験を行なった．本実験においては MDD の非終端節点か

表 1: muscat との比較(ランダムな 1000 パターン)

circuit name	CPUtime(sec) our	CPUtime(sec) muscat	speedup ratio
s27	0.3	0.4	1.33
s208	3.1	3.2	1.03
s298	5.4	6.5	1.20
s344	5.2	6.6	1.27
s349	5.4	7.3	1.35
s382	7.6	9.0	1.18
s386	3.4	3.8	1.12
s400	8.0	9.7	1.21
s420	6.4	7.1	1.11
s444	9.9	11.5	1.16
s510	10.2	15.8	1.55
s526n	12.0	14.6	1.22
s526	12.3	14.6	1.19
s641	7.0	8.7	1.24
s713	8.1	9.3	1.15
s820	14.5	15.0	1.03
s832	15.1	15.1	1.00
s838	13.6	15.0	1.10
s1196	15.4	16.4	1.06
s1238	17.1	18.3	1.07
s1423	39.2	34.0	0.87
s1488	28.0	32.1	1.15
s1494	28.5	32.6	1.14
s5378	66.7	78.2	1.17
s9234	15.0	20.2	1.35
s13207	155.4	248.4	1.60

表 2: PROOFS との比較

circuit name	test vectors	fault coverage	CPUtime(sec) our	CPUtime(sec) PROOFS	speedup ratio
s208	111	63.7	1.2	1.0	0.83
s344	90	96.2	1.5	1.3	0.85
s349	90	95.7	1.6	1.4	0.87
s420	173	41.6	3.2	4.1	1.28
s641	133	86.3	1.8	2.5	1.39
s713	107	80.9	2.7	2.6	0.96
s820	411	81.9	17.9	9.6	0.54
s832	377	81.4	18.1	9.1	0.50
s838	137	29.6	6.2	7.6	1.23
s953	16	7.8	4.2	3.5	0.83
s1196	313	99.8	7.5	6.7	0.89
s1423	36	24.4	9.1	7.0	0.77
s5378	408	74.0	48.5	99.5	2.05

ら出る枝の最大数 m を 16，終端節点の持つ信号値数 n を 32 で実現した．

まず，[9] で提案された手法 (muscat) と比較した結果を表1に示す．テストパターンにはランダムな 1000 パターンを用い，実験は SPARCstation2 上で行なった．[9] の手法は，PVL 法を改良したものであり故障状態リストは線形リストの形で保持している．表1からもわかるように，全般において本手法が約 1.1 ～1.3 倍高速であり，特に s13207 では 1.6 倍高速である．

次に順序回路に対して最も高速な手法の 1 つである PROOFS[6] と比較した結果を表2に示す．テストパターンは STG3[12] によって生成された同一のものを用いて実験を行なった．但し，本手法は Sun4/330 上で PROOFS は Sun4/280 上で実験を行なったデータ [6] である．本手法と PROOFS を比較すると，ほぼ同程度の性能を持つと考えられるが，最も大きな回路 (s5378) では本手法が約 2 倍高速である．

以上の実験からもわかるように，本手法は比較的大規模な回路に対して有効であると考えられる．

5 まとめ

本稿では，順序回路に対する高速な故障シミュレーション手法を提案した．本手法では，伝播故障の集合を多分決定グラフ (MDD) 表現で表すことによって，ゲート評価，故障の挿入，故障のドロップの処理が簡素化し，シミュレーション全体の高速化を実現した．実験の結果，本手法は従来法と比較して有効であることが確認された．

今後の課題としては，さらに大規模な回路に対してもシミュレーションを可能にすることである．

謝辞

STG3 によるテストパターンを提供して頂いたイリノイ大学 Rudnick 氏に感謝致します．また，御討論頂いた本学白川研究室の諸氏に感謝します．

参考文献

[1] D. B. Armstrong, "A deductive method for simulating faults in digital logic circuits," *IEEE Trans. Computers*, vol. C-21, no. 5, pp. 464–471, 1972.

[2] E. G. Ulich and T. Baker, "Concurrent simulation of nearly identical digital networks," *Proc. 10th Design Automation Workshop*, vol. 6, pp. 145–150, 1973.

[3] S. Seshu, "On an improved diagnosis program," *IEEE Trans. Electronic Computers*, vol. EC-14, pp. 76–79, 1965.

[4] P. R. Moorby, "Fault simulation using parallel value lists," *Proc. IEEE ICCAD*, pp. 101–102, 1983.

[5] K. Son, "Fault simulation with the parallel value list algorithm," *VLSI Systems Design*, pp. 229–233, 1985.

[6] T. M. Niermann, W. T. Cheng, and J. H. Patel. "PROOFS: A fast, memory efficient sequential circuit fault simulator," *IEEE Trans. Computer-Aided Design*, vol. CAD-11, no. 2, pp. 198–207, 1992.

[7] F. Ozguner, et al., "On fault simulation techniques, " *Journal of Design Automation and Fault Tolerant Computing*, vol. 3, no. 2, pp. 83–92, 1979.

[8] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom and T. McCarthy, "Fault Simulation for Structured VLSI," *VLSI System Design*, pp. 20–32, Dec. 1985.

[9] 岩下洋哲，康 敏燮，出口 弘，白川 功，"故障シミュレーションの同期回路向き高速化手法," **信学技報**, CAS90-99, pp. 7–13, 1990.

[10] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. IEEE ISCAS*, pp. 1929-1934, 1989.

[11] 湊真一，石浦菜岐佐，矢島脩三，" 論理関数の共有二分決定グラフによる表現とその効率的処理手法," **情報処理学会論文誌**, vol. 32, no. 1, pp. 77–85, Jan. 1991.

[12] W.-T. Cheng and S. Davidson, "Sequantial circuit test generator(STG) benchmark results," *Proc. Int. Symp. Circuits Syst.*, pp. 1938–1941, May 1989.

[13] Y. Levendel and P. R. Menon, "Fault Simulation," *Fault-Torelant Computing: Theory and Techniques*, vol. 1, chapter 3, pp. 184–264, 1986.