ISSN 2186-5647

テスト環境生成結果を用いた階層テストのための動作合成法

日大生産工(院)　○藤原 浩顕　　日大生産工　細川 利典
大阪学院大　藤原 秀雄　　日大生産工(院)　井上 諒一

1. はじめに

近年，半導体微細化技術の進歩に伴い，大規模集積回路(Large Scale Integrated circuits : LSI)が大規模化，複雑化してきている．組合せ回路に関しては，効率の良いテスト生成アルゴリズムが研究開発されており，大規模回路に対しても高い故障検出効率を得ることができている[1]-[5]．しかしながら，順序回路については回路規模に対して指数関数的にテスト生成時間が増大し，大規模回路では膨大な時間となるため，従来のゲートレベルでのテスト生成アルゴリズムでは実用的な時間で高い故障検出率を達成することが困難となってきている．

一方，近年 LSI の設計はレジスタ転送レベル（Register Transfer Level:RTL）から開発を開始するのが一般的だが，LSI の回路規模の増加に比べ設計生産性が向上していないため，従来の設計方法では設計が困難になってきている[6]．それゆえ，より抽象度の高い動作レベルで設計し，RTL 回路を合成する動作合成[6]という設計方法論が注目されてきている[6]．

動作記述は RTL 記述での設計に比べ，記述量が少ないため，設計生産性に優れるが，その反面，レジスタや演算器の割当てなどを動作合成ツールが決定するため，動作合成ツールの性能が合成後の回路の性能に大きく影響する[6]．

動作合成は，スケジューリング[6]と呼ばれる，各演算操作を各時刻に割当てる処理と，バインディング[6]と呼ばれる，演算操作および変数をそれぞれ演算器およびレジスタに割当てる処理から構成されている．スケジューリングを実行した後のサイクル精度な記述の回路を，機能 RTL 回路という．スケジューリングとバインディングを実行した後の，演算操作と変数がそれぞれ演算器とレジスタに割当てられた回路を，構造 RTL 回路という．機能 RTL 回路や構造 RTL 回路は論理合成をおこなうことにより，ゲートレベル回路に合成される．

構造RTL回路に対して階層テスト生成[7]に基づくテスト容易化設計が提案されている[8]-[10]．しかしながら，文献[7]の構造 RTL 回路モデルはコントローラとデータパスが分離されており，データパスとコントローラに対して，別々のテスト容易化設計が必要になり，面積オーバーヘッドが大きくなる．

また，機能 RTL 回路を対象に，縮退故障の階層テスト生成手法が提案されている[11]-[13]．これらの手法は，機能 RTL 回路を割当決定図（Assignment Decision Diagram: ADD）[14]で表現したものを対象に，各機能要素のテスト環境(Test Environment)を生成し，各機能要素のゲートレベル回路に対する縮退故障テストパターンをテスト環境に代入することでテスト系列を生成している．これらの手法は，従来のゲートレベルでのテスト生成と比較して高速にテスト生成が可能となる[11]-[13]．

しかしながら，これらの手法はバインディングを論理合成に任せているため，各機能要素のテスト環境生成率[12]-[13]が低いと，生成したテスト系列が論理合成後の回路に対し高い故障検出率を得られる保証はない．文献[15]では，テスト環境生成後のバインディング時に、テスト環境を考慮してバインディングを行うことで回路のテスト容易性を向上させる手法が提案されている．

本論文は，文献[15]の手法を発展させ，機能要素中の演算を対象として ADD を用いて生成したテスト環境を考慮し，テスト環境生成が成功した演算と失敗した演算を優先的に同一演算器に割当てを行うことで，テスト系列生成困難な演算器を減らし，回路中の演算器のテスト容易化を実現するバインディング手法を提案する．

2. ADD

本論文の対象である ADD について説明する．ADD は，図 1 のように割当値，割当条件，割当決定，割当対象の 4 つの要素から構成されている．さらにこれらは，演算ノード，読込みノード，書込みノード，割当決定ノード（Assignment Decision Node: ADN）の 4 つのノードで構成されている．演算ノードは算術演算，論理演算を，読み込みノードは外部入力，変数，定数を，書込みノードは外部出力，変数を表現している．ADN では条件入力に対応する値入力が選択され，演算で計算された割当値が割当対象に入力される．

ADD は本来，動作合成内部の処理で用いるために提案されたものであるが，機能 RTL 回路を表現するモデルとして用いることもできる．コントローラとデータパスを一様に表現できるため，コントローラとデータパスを分離せずにテスト環境生成を行うモデルとして適している．

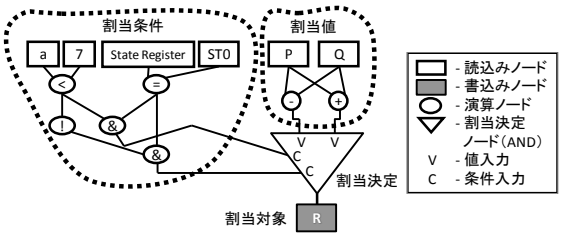

図 1 : ADD

A Binding Method for Hierarchical Testing Using the Results of Test Environment Generation

Hiroaki FUJIWARA　Toshinori HOSOKAWA　Hideo FUJIWARA　and　Ryoichi INOUE

図 2 : 動作合成フロー

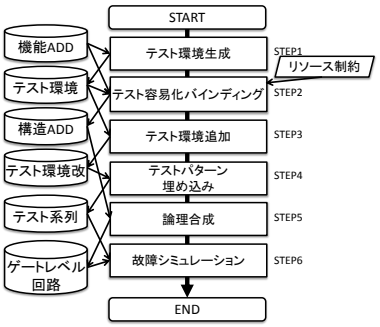

図 3 : テスト生成フロー

3. 動作合成

動作合成とは，C 言語や SystemC などで記述された動作記述から，ハードウェア記述言語である VHDL[16]や Verilog HDL[17]などの RTL 回路記述を生成するものである．動作合成の手順は図 2 に示すようにの 4 つのフェーズに分けられる：グラフ生成，スケジューリング，バインディング，RTL 回路記述生成．グラフ生成では，与えられた動作記述を ADD[14]やコントロール/データフローグラフ（Control / Data-Flow Graph: CDFG）[6]と呼ばれるグラフで表現する．スケジューリングでは，各演算操作をどの時刻に実行するかを決定する．バインディングでは，スケジューリングした情報をもとに，各演算操作や変数に，具体的な演算器やレジスタを割当てる処理をおこなう．RTL 回路記述生成では，割当てられた演算器やレジスタ間の結線を行いデータパスを生成する．さらに，制御信号を生成するコントローラを生成する．

ADD を用いた動作合成では，スケジューリングやバインディングをグラフ構造を変更することで実現する．スケジューリング済みの ADD を機能 ADD，バインディング済みの ADD を構造 ADD と呼ぶこととする．

4. 階層テスト生成

4.1. 階層テスト生成フロー

機能 RTL 回路をグラフ化した機能 ADD を対象に階層テスト生成を行う．階層テスト生成は以下の STEP1 のテスト環境生成と STEP2 のテスト系列生成の 2 つのフェーズから構成される．

(STEP1) 構成する各ノードに対して，任意の値を外部入力が割当てられている読込みノードまで正当化する経路を正当化経路，任意の値を外部出力が割当てられている書込みノードまで伝搬する経路を伝搬経路と呼ぶ．まず，各ノードごとに正当化経路と伝搬経路を有効にする値の時系列であるテスト環境を生成する．この際に使用する値は，文献[11]で提案された 9 値の代数を使用する．

(STEP2) 次に，テスト環境が生成された演算に対して，あらかじめライブラリ化しておいた各ノードのテストパターンをテスト環境に代入することで，テスト系列を生成する．代入するテストパターンは，ノードの種類ごとに論理合成し，生成したゲートレベル回路に対し，組合せテスト生成を実行して生成したものである．

4.2. 階層テスト生成の問題点

ADD の全ノード数に対し，テスト環境が生成されたノード数の割合であるテスト環境生成率[12]-[13]が高いほど，ゲートレベルでの故障検出率も高くなる．しかしながら，文献[11]の手法でも，ゲートレベルのテスト生成手法と比較して高い故障検出率を得られているとは言えない．この理由としては以下のことが挙げられる．

従来法では動作合成の手順の一つであるバインディングを考慮しておらず，バインディングが実行された際に，ある演算器に対して，テスト環境が生成できない演算のみが割当てられる可能性がある．この場合，動作合成後の RTL 回路においてテスト環境が存在しない演算器が残る．したがって，論理合成後のゲートレベル回路に対しても，生成したテスト系列で検出できない故障が残り，故障検出率が低下していると考えられる．

この問題を解決するために，本手法では，ある演算器に対して，テスト環境が生成できない演算のみが割当てられないように，動作合成の段階でバインディングすることで，故障検出率を向上させるバインディング手法を提案する．

5. テスト容易化バインディング

5.1. テスト全体フロー

本テスト生成法の手順について説明する．テスト生成フローを図 3 に示す．

(STEP1) 機能 ADD を入力とし，文献[11]の手法を用いてテスト環境を生成する．

(STEP2) STEP1 でテスト環境の生成が不可能なノードが存在した場合，与えられた演算器数制約のもとでテスト容易化を考慮したバインディング（後述 5.2）を行い，構造 ADD を生成する．テスト環境の生成が不可能なノードが存在しない場合は，与えられた演算器数制約のもとでバインディングを行い，構造 ADD を生成する．

(STEP3) STEP2 でバインディングを実行することで ADD 上に ADN が追加されるので，追加された ADN に対してテスト環境を生成する．

(STEP4) STEP3 で変更したテスト環境に被テストノードのテストパターンを代入し，テスト系列を生成する．

(STEP5) STEP2 で生成した構造 ADD を論理合成し，ゲートレベルの論理回路を生成する．

(STEP6) STEP4 で生成したテスト系列で故障シミュレーションを実行し，故障検出率を算出する．

5.2. 問題定式化

(定義 1:機能 RTL テスト環境生成率)

$$\mathrm{RTE_F} = \frac{\text{テスト環境生成に成功した機能要素数}}{\text{テスト対象の機能要素数}}(\%)$$

機能 RTL 回路において，テスト対象の機能要素数に対する，テスト環境生成に成功した機能要素数の割合を機能 RTL テスト環境生成率(RTE_F)という．本論文では，テスト対象の機能要素数を演算数とする．

(定義 2:構造 RTL テスト環境生成率)

$$\mathrm{RTE_S} = \frac{\text{テスト環境生成に成功した機能要素が割当てられているモジュール数}}{\text{テスト対象のモジュール}}(\%)$$

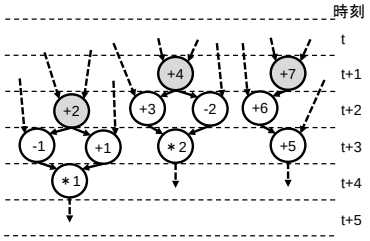

図 4 : スケジューリング結果

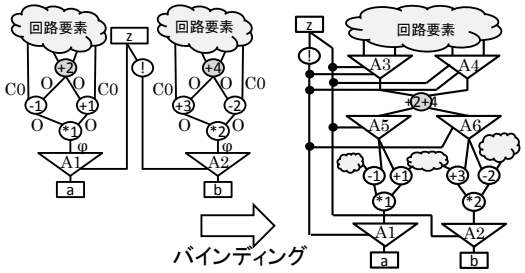

図 5 : バインディング例 1(+2 と+4 を共有化)

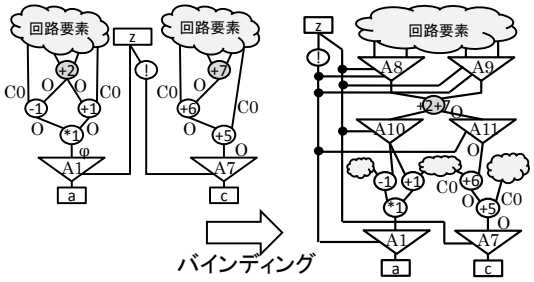

図 6 : バインディング例 2(+2 と+7 を共有化)

構造 RTL 回路において，テスト対象のモジュール数に対する，テスト環境生成に成功した機能要素が少なくとも 1 つ以上割当てられているモジュール数の割合を構造 RTL テスト環境生成率(RTE_S)という．本論文では，テスト対象のモジュール数を演算器数とする．

(定式化)

・入力 : 機能 RTL 回路におけるテスト対象の機能要素と各機能要素に対するテスト環境生成結果．$e_1, e_2, \cdots e_n$　n はテスト対象機能要素数．

・出力 : バインディング結果．$B_1, B_2, \cdots B_m$　m はモジュール数．

・最適化 : テスト環境生成が成功した機能要素が少なくとも 1 つ以上割当てられたモジュール数を最大化すなわち構造 RTL テスト環境生成率$\sum_{i=1}^{m} F_{Bi}$の最大化．ただしF_{Bi}はテスト環境生成に成功したe_j($1 \leqq j \leqq n$ の整数)が少なくとも 1 つ以上割当てられている場合は 1，それ以外の場合は 0 とする．

・制約 : モジュール数 m．

5.3. テスト環境考慮バインディング

提案するテスト容易化バインディングを，例を用いて説明する．図 4 のスケジューリング結果をもとに，図 5 と図 6 にバインディングを行った例を示す．

加算+2 と加算+4 に対して，テスト環境を生成した場合，故障の影響を観測できるという意味の代数記号 O[11]が，それぞれ乗算*1 と乗算*2 の出力に伝搬できず，テスト環境の生成に失敗する．加算+7 に関してはテスト環境を生成できている．加算+2 と加算+4 を同じの加算器にバインディングした場合（図 5），この加算器に対するテスト環境は存在せず，テストができない．一方，加算+2 と加算+7 を一つの加算器にバインディングした場合（図 6），加算+7 に対して生成したテスト環境を使用することができ，この加算器に対するテストが可能である．

```
1:    Binding_for_testability (an operation set OP) {
2:      for(each operation Ni with a test environment){
3:        for(each operation Nj without a test environment){
4:          if(Ni and Nj mergeable) {
5:            merge Ni and Nj to a new operation Nnew;
6:            add Nnew to OP;
7:            delete Ni and Nj from OP;
8:            break;
9:          }
10:       }
11:     }
12:     for(each operation Ni){
13:       for(each operation Nj except Ni){
14:         if(Ni and Ni mergeable){
15:           merge Ni and Nj to a new operation Nnew;
16:           delete Ni and Nj;
17:           add Nnew to OP;
18:           break;
19:         }
20:       }
21:     }
22:     assign each node in OP to each operational unit;
23:   }
```

図 7 : 提案手法疑似コード

本手法では，テスト環境生成に成功した演算と，テスト環境生成に失敗した演算を優先的にバインディングを行うことにより，リソース制約を満たす範囲内で構造 RTL テスト環境生成率の向上を目指す．図 7 に本手法の疑似コードを示す．

(1 行目) テスト環境生成結果を付加した演算集合 OP を対象とする．
(2 行目)テスト環境生成成功演算 Ni の数だけ 3～9 行目を繰り返す．
(3 行目)テスト環境生成失敗演算 Nj の数だけ 4～8 行目を繰り返す．
(4 行目)Ni と Nj が同一演算に割当て可能ならば 5～8 行目が実行される．
(5 行目)Ni と Nj を同一演算 Nnew に割当てる．
(6 行目)Nnew を OP に追加する．
(7 行目)Ni と Nj を OP から削除する．
(8 行目)2 行目に戻る．
(12 行目) OP の演算 Ni の数だけ 13～19 行目を繰り返す．
(13 行目) OP から Ni を除いた演算 Nj の数だけ 13～19 行目を繰り返す．
(14 行目) Ni と Nj が同一演算に割当て可能ならば 14～18 行目が実行される．
(15 行目) Ni と Nj を同一演算 Nnew に割当てる．
(16 行目) Nnew を OP に追加する．
(17 行目) Ni と Nj を OP から削除する．
(18 行目) 12 行目に戻る．
(22 行目)OP 内の各演算を演算器に割当てる．

6. 実験結果

paulin, diffeq, DCT の 3 つの回路に対し，文献[11]の手法でテスト環境，テスト系列を生成し，テスト環境考慮演算バインディングと，テスト環境を考慮していない演算バインディングであるレフトエッジアルゴリズム(Left Edge Algorothm:LEA)[18]を用いて作成した回路を作成した．

表 1 は各回路でのテスト環境生成結果，表 2 は各回路でのバインディン

表 1.テスト環境生成結果

	テスト環境生成成功	テスト環境生成失敗
paulin	+1,+2,−2,*1,*2,*4,*6	−1,*3,*5
diffeq	+1,+2,−2,*1,*2,*4,*6,<	−1,*3,*5
DCT	+1,+2,+3,−1,−2,*1,*2,*3,*4,*5	+4,+5,+6

表 2.バインディング結果

	演算	TE考慮	TE非考慮
paulin	MUL1	*3,*6	*3,*5
	MUL2	*1,*2,*4,*5	*1,*2,*4,*6
	ADD	+1,+2	+1,+2
	SUB	−1,−2	−1,−2
diffeq	MUL1	*4,*5	*3,*5
	MUL2	*1,*2,*3,*5	*1,*2,*4,*6
	ADD	+1,+2	+1,+2
	SUB	−1,−2	−1,−2
	LES	<	<
DCT	MUL1	*1,*2,*3	*1,*2,*3
	MUL2	*4,*5	*4,*5
	ADD1	+1,+3,+4,+6	+1,+3,+3,+5
	ADD2	+2,+5	+4,+6
	SUB	−1,−2	−1,−2

表 3.テスト環境生成率

	機能RTLテスト環境生成率(%)	構造RTLテスト環境生成率(%)	
		TE考慮	TE非考慮
paulin	70.00(7/10)	100.00(4/4)	75.00(3/4)
Diffeq	72.73(8/11)	100.00(5/5)	80.00(4/5)
DCT	76.92(10/13)	100.00(5/5)	80.00(4/5)

表 4.テスト実行結果

bit幅	回路	故障検出率(%)		故障検出効率(%)		テスト実行時間(秒)	
		提案手法	従来法	提案手法	従来法	提案手法	従来法
4	paulin	100.00	89.65	100.00	89.65	0.03	0.02
	diffeq	96.58	96.58	96.85	96.58	0.04	0.03
	DCT	65.17	62.85	95.68	95.51	0.03	0.04
8	paulin	100.00	80.05	100.00	80.05	0.26	0.26
	diffeq	98.16	98.16	98.16	98.16	0.28	0.28
	DCT	60.19	59.06	99.57	98.62	0.11	0.12
16	paulin	99.17	94.97	99.17	94.97	3.91	3.80
	diffeq	98.96	98.96	98.96	98.96	5.27	5.12
	DCT	51.26	51.26	99.73	99.56	0.64	0.75

グ結果, 表 3 はバインディング前の演算操作のテスト環境生成率を示す機能 RTL テスト環境生成率と,バインディング後の演算器のテスト環境生成率を示す構造 RTL テスト環境生成率を示す.

それぞれの回路に対してテスト環境から生成したテスト系列で演算器に対して故障シミュレーションを実行し故障検出率, 実行時間を調べた. 表 4 より, 提案手法で生成した回路の方が, テスト環境を考慮しないバインディングを行った従来法の回路よりも, 階層テスト生成を実行した際に故障検出率が平均して 4.22%, 故障検出効率が平均 4.01%向上している. また, 提案手法の 4 ビット, 8 ビット幅の paulin 回路では, 100%の故障検出効率を得ることができた. DCT 回路の故障検出率が他の路に比べ低い理由は, DCT 回路中の乗算全てが片側入力が定数入力である定数演算であるため, バインディング後の乗算器中にテスト生成困難故障やテスト不可能故障が多数発生していることが考えられる. これを解決するためには, Ghosh[11] のテスト環境生成規則を拡張した文献[13]の生成規則を適用することなどが必要と思われる. また, 提案手法の回路と従来法の回路のテスト実行時間はあまり差異が認められなかった.

7. まとめ

本論文では, ADD を用いて生成したテスト環境を考慮したバインディング手法を提案した. その結果, テスト環境を考慮しない従来のバインディング手法に比べ, 高い故障検出率, 高い故障検出効率を示すことができた. また, 今後の課題として, 演算器以外に対してのテスト環境も考慮したバインディングや、更なる大規模回路での検証, 他のテスト容易化バインディング手法と組合せなどが挙げられる.

「参考文献」

[1] M. Schulz, E. Trischler, and T. Sarfert,"SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," IEEE Trans. on Computer-Aided Design, Vol. 7, No. 1, pp. 126-137, Jan. 1988.

[2] W. Kunz and D. Pradhan,"Recursive Learning: An Attractive Alternative to the Decision Tree for Test Genration in Digital Circuits," Proc. IEEE International Test Conference on Discover the New World of Test and Design, pp.816-825, 1992.

[3] M. Henftling, H.C.Wittmann, K.J.Antreich,"A Single-Path-Oriented Fault-Effect Propagation in Digital Circuits Considering Multiple-Path Sensitization," Proc. 1995 IEEE/ACM international conference on Computer-aided design, pp.304-309, 1995.

[4] C.Wang, S.Reddy, I.Pomeranz, X.Lin, J.Rajski,"Conflict driven techniques for improving deterministic test pattern generation," Proc. IEEE International Conference on Computer-Aided Design, pp.87-93, 2002.

[5] E. Gizdarski and H. Fujiwara, "SPIRIT: a highly robust combinational test generation algorithm," IEEE Trans. on Computer Aided Design for Integrated Circuits and Systems, Vol. 21, No. 12, pp. 1446-1458, Dec. 2002.

[6] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin, HIGH-LEVEL SYNTHESIS Introduction to Chip and System Design, Kluwer Academic Publisher, 1992.

[7] B. T. Murray and J. P. Hayes, "Hierarchical test generation using pre computed tests for modules," IEEE Trans. Computer-Aided Design, Integrated Circuits & Syst., vol.9, no.6, pp.594-603, Jun. 1990.

[8] I. Ghosh, A. Raghunathan, and N. K. Jha, "Design for Hierarchical Testability of RTL Circuits Obtained by Behavioral Synthesis," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 16, pp. 1001-1014, Sep. 1997.

[9] S. Ohtake, T. Masuzawa, and H. Fujiwara, "A non-scan approach to DFT for Controllers Achieving 100% Fault Efficiency," Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 16, No. 5, pp.553-566, Oct. 2000.

[10] H. Wada, T. Masuzawa, K. K. Saluja and H. Fujiwara, "Design for strong testability of RTL data paths to provide complete fault efficiency," Proc. of 13th International Conf. on VLSI Design, pp. 300-305, Jan. 2000.

[11] I. Ghosh, M. Fujita, "Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams," Proc. of ACM/IEEE Design Automation Conference, pp.43-48, Jun. 2000.

[12] L. Zhang, I. Gosh, and M. Hsiao, "Efficient sequential atpg for functional rtl circuits," in Proc. International Test Conference, pp. 290-298, Sep. 2003.

[13] H. Fujiwara, C. Y. Ooi and Y. Shimizu, "Enhancement of test environment generation for assignment decision diagrams," 9th IEEE Workshop on RTL and High Level Testing (WRTLT'08), pp.45-50, Nov. 2008.

[14] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances," in Proc. Design Automation Conference, pp. 413-428, Jun. 1993.

[15] 井上 諒一,藤原 浩顕,細川 利典,藤原 秀雄, 動作記述を用いた順序テスト生成およびテスト容易化バインディング, 電子情報通信学会. DC, ディペンダブルコンピューティング 110(317), 143-148

[16] 中 幸政, VHDL と CPLD によるロジック設計入門, CQ 出版株式会社, 東京, 2005.

[17] 並木 秀明,前田 智美,宮尾 正大, ディジタル回路と Verilog-HDL, 株式会社技術評論社, 東京, 1996.

[18] F.J.Kurdahi and A.C.Parker, REAL: A program for regiter allocation, In Proc. Design