システムＬＳＩ設計技術　94－5
（2000．1．11）

一般二分決定グラフの生成法

片山 哲志　越智 裕之　津田 孝夫

広島市立大学情報科学部情報工学科
〒731-3194 広島市安佐南区大塚東 3-4-1
Tel: 082-830-1550
E-mail: {katayama, ochi, tsuda}@arch.ce.hiroshima-cu.ac.jp

あらまし　本稿では一般二分決定グラフ (GBDD: Generic BDD) の生成法を提案する．ここで GBDD とは OBDD (Orderd BDD) や FBDD (Free BDD) にみられる変数の出現順および変数の出現回数の制約を一切持たない，最も一般的な BDD である．これらの制約を緩めることで，論理関数を表現するのに必要なノード数が削減され，パストランジスタ回路の合成などに活用できると期待される．提案する方法では，与えられた論理式から構文木を構築し，構文木をグラフ化することにより GBDD を生成する．ISCAS'89, MCNC ベンチマークの論理関数を使った実験では，OBDD に対し最大約 23.1% のノードを削減した GBDD を自動生成することができた．

キーワード　論理合成，論理関数，スイッチング理論，二分決定グラフ，パストランジスタ論理

An Algorithm for Generating Generic BDDs

Tetsushi KATAYAMA, Hiroyuki OCHI, and Takao TSUDA
Department of Computer Engineering
Faculty of Information Sciences, Hiroshima City University
3-4-1, Ozuka-higashi, Asaminami-ku, Hiroshima, 731-3194, JAPAN
Tel: +81-82-830-1550
E-mail:{katayama, ochi, tsuda}@arch.ce.hiroshima-cu.ac.jp

Abstract　This paper proposes an algorithm for generating Generic BDDs (GBDDs), where GBDDs are the most generic BDDs. In contrast with OBDDs and FBDDs, GBDDs have no restrictions in variable ordering and variable appearance count on its paths. This flexibility enables us to represent Boolean functions with smaller number of nodes, which is useful for synthesis of pass-transistor logic. Given a Boolean formula, our method first generates parse trees, and consequently converts them to a GBDD. In experiments on ISCAS'89 and MCNC benchmark circuits, compared with OBDDs, the developed program achieved 23.1% reduction in size in the best case.

key words　Combinational Synthesis, Logic Function, Switching Theory, Binary Decision Diagram, Pass-Transistor Logic

1 はじめに

二分決定グラフ (BDD: Binary Decision Diagram)[1] はグラフによる論理関数の表現である．中でも順序付き BDD (OBDD: Orderd BDD)[2] は表現に一意性があり，効率良く処理できるため，多くの場合に有用である．OBDD を自動生成する BDD パッケージが開発され，形式的設計検証，論理合成，消費電力見積りなど，論理回路設計の諸分野で広く利用されている．

パストランジスタ回路の合成は BDD パッケージを応用した成功例の一つである．パストランジスタ論理は従来のディジタル回路の主流である CMOS 論理より低消費電力，高速，省チップ面積を実現できる回路方式として注目を集めつつある [3, 4, 5]．パストランジスタ回路は BDD の各ノードを MOS トランジスタ 2 個または 4 個からなるセレクタに置き換えることで生成できる．この時，ノード数の少ない BDD から回路を生成することはトランジスタ数の削減につながり，面積や消費電力の削減が期待できる．

パストランジスタ回路の生成のための BDD は BDD パッケージから得ることができる．しかし，BDD パッケージから得られる OBDD はより一般的な BDD と比べ，表現に一意性を持たせるための制約を持っており，それらの変数順に関する制約はパストランジスタ回路生成のための BDD には必ずしも必要ではない．実際，変数順に関する制約を緩めることで BDD のノード数を劇的に減らすことも可能である [6]．

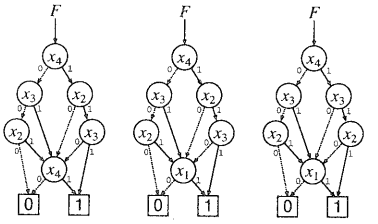
本稿では，変数順や変数の出現回数の制約を持たないより一般的な BDD の生成法として，一般二分決定グラフ (GBDD: Generic BDD) の生成法について述べる．GBDD は OBDD と比べ表現の制約が緩いため，ノード数の削減が期待できる．この手法では入力を論理式によって行い，論理式の構文に基づいた処理を行うことで GBDD を生成する．この手法から生成される GBDD のノード数は最悪でも論理式に出現するリテラルの数と等しくなる．

以下 2 章では準備として基本的な用語を定義し，3 章で GBDD の生成法について述べる．4 章では作成したプログラムを使った実験とその結果を示し，5 章でまとめを述べる．

2 準備

2.1 BDD

二分決定グラフ (BDD: Binary Decision Diagram)[1] は根付きの非巡回有向グラフによる論理関数の表現である．BDD は終端ノードと非終端ノードおよび非終端ノードが持つ 2 つの矢印によって構成される．終端ノードは論理値でラベル付けされており，非終端ノードは変数でラベル付けされている．非終端ノードが持つ 2 つの矢印はそれぞれ 0 エッジ，1 エッジと呼ばれ，0 エッジは変数に 0 が代入された場合に対応し，1 エッジは変数に 1 が代入された場合に対応している．



(a) GBDD (b) FBDD (c) OBDD

図 1: BDD

BDD は同じ論理関数を表す場合でも複数存在する．その中でも，根から終端ノードに至るあらゆる経路上で，各変数が高々 1 回しか出現しない BDD のことを Free BDD (FBDD)[7] という．更に，全ての経路上での変数の出現順がある全順序に従う FBDD を Orderd BDD (OBDD)[2] という．OBDD は変数順によってグラフの形やノード数が大きく変化する．OBDD のノード数がなるべく少なくなるような変数の順序を決めることを変数の順序付けと呼ぶ．

本稿では，FBDD や OBDD が持つ変数の出現順や変数の出現回数に関する制約を持たない，より一般的な BDD のことを一般二分決定グラフ (Generic BDD) という．図 1に GBDD，FBDD，OBDD の例を表す．GBDD による表現は FBDD や OBDD と比べ制約が緩いため，より少ないノードでの表現が期待できる．

2.2 BDD を用いたパストランジスタ回路の生成

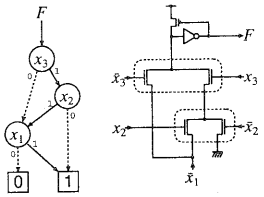
パストランジスタ回路は，与えられた論理関数を表現する BDD の各非終端ノードを MOS トランジスタ 2 個または 4 個からなるセレクタに置き換えることで生成できる．

例として，図 2(a) の BDD から生成したパストランジスタ論理 SPL (Single-rail Pass-transistor Logic)[5] の回路を図 2(b) に表す．この時，0 エッジ，1 エッジが共に終端ノードを指す非終端ノードについては，セレクタに置き換える代わりに，対応する変数の入力またはその否定を直接接続することができる．図 2(a) の BDD では変数 x_3 のノードがそれに相当する．

3 一般二分決定グラフの生成法

3.1 処理系での GBDD の生成

今回開発した処理系では以下の二段階の処理を行うことにより GBDD を生成する．



(a) BDD (b) SPL

図 2: パストランジスタ論理 SPL の回路生成

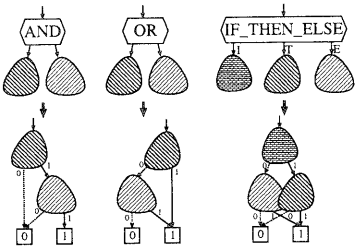
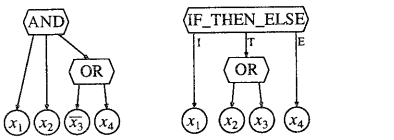
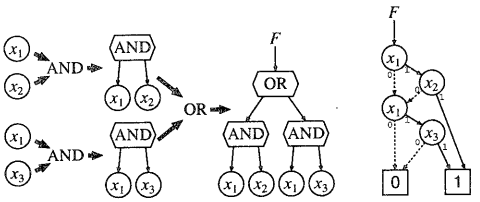


図 4: 連結法



(a) $x_1x_2(\overline{x_3}+x_4)$ (b) $x_1(x_2+x_3)+\overline{x_1}x_4$

図 3: 構文木の例



(a) 構文木を構築する過程 (b)GBDD

図 5: $F = x_1x_2 + x_1x_3$ を表す構文木と GBDD

1. 入力された論理式から構文木をつくる
2. 完成した構文木から GBDD を生成する

この手順によって生成される GBDD のノード数は最悪でも論理式に出現するリテラルの数と等しくなる.

構文木は論理関数を木で表したもので, 変数ノードと演算ノードによって構成される. 変数ノードはリテラル (変数またはその否定) でラベル付けされており, 演算ノードは AND, OR, IF_THEN_ELSE でラベル付けされている. 構文木の葉は必ず変数ノードになり, それらを演算ノードが束ねることで論理関数を表している.

AND, OR が束ねているエッジの先の構文木に関しては, 演算の優先順位が等しく, その部分で結合律が成り立っている. このため, 必要があれば入れ換えることもできる. 一方, IF_THEN_ELSE は常に 3 つのエッジをもつ変則的な演算ノードである. 一番左のエッジの先 (IF) の構文木の表す論理関数が真なら中央のエッジの先 (THEN) の構文木の論理関数に従い, 偽なら右端のエッジの先 (ELSE) の構文木の論理関数に従う. 構文木の例として, $x_1x_2(\overline{x_3}+x_4)$ と等価な構文木を図 3(a) に示し, $x_1(x_2+x_3)+\overline{x_1}x_4$ と等価な構文木を図 3(b) に示す.

構文木が完成したら, それをもとに GBDD を生成する. GBDD の生成は構文木の変数ノードをグラフ化した後に連結法を適用することで行う. 連結法はグラフ同士の連結により GBDD を生成する手法である. 構文木の各演算ノードに対応した連結法を図 4に示す.

3.2 構文木の構築

構文木の構築は入力された論理式の形に基づいて行う. 全体を表す構文木は, 論理式の変数に相当する変数ノードを生成した後に, それらの構文木を演算ノードで束ねる操作を繰り返すことで構築する. 否定演算に関してはド・モルガンの法則を適用し, 変数ノードにのみ否定の構文木が存在するようにする. (1) 式の論理関数から構文木を構築する過程を図 5(a) に表す.

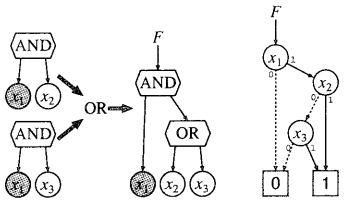
$$F = x_1x_2 + x_1x_3 \qquad (1)$$

図 5(a) で構築された構文木に連結法を適用することで図 5(b) の GBDD が生成される.

図 5の例からも明らかなように, 構文木に連結法を適用し GBDD を構築する場合, 生成される GBDD のノード数は構文木の変数ノードの数と等しくなる. そのため, 構文木を構築する段階では, できるだけ変数ノードの少ない構文木を構築することが重要である.

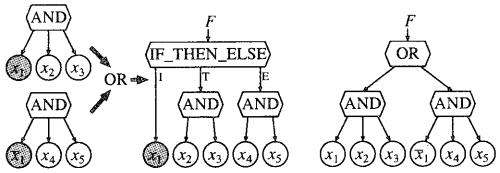
例えば, 図 5(a) の中央から右への OR 演算は図 6(a) のように行うことで変数ノードの数を減らすことができる. この操作は論理式の簡約化 $x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$ に着目した構文木の構築である. 図 6(a) の構文木から生成した図 6(b) の GBDD は図 5(a) の構文木から生成した図 5(b) の GBDD よりもノードが少なくなる.

図 6(a) の例は論理式の分配律に着目したものであるが, 吸収律やベキ等律などに着目することでも構文木の簡約化を行うことができる. GBDD の処理系では論理



(a) 簡約化　(b)GBDD

図 6: 論理式の簡約化に対応した構文木の構築



(a)IF_THEN_ELSE を導入した構文木　(b) 構文木

図 7: IF_THEN_ELSE による構文木の変形

式の式変形に対応した構文木の簡約化を，変数ノードが減る場合についてのみ適用している．

(2) 式の論理関数に相当する構文木を作成する場合，図7(a) のように IF_THEN_ELSE を導入することで，より変数ノードの少ない構文木を構築できる．

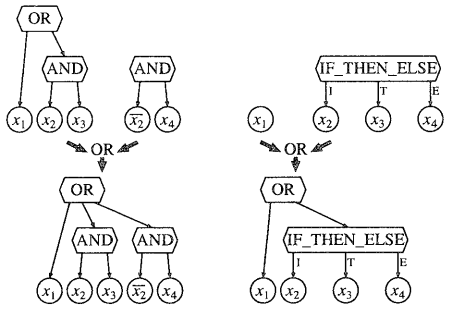
$$F = x_1x_2x_3 + \overline{x_1}x_4x_5 \quad (2)$$

図7(a) の場合，$x_1x_2x_3$ と $\overline{x_1}x_4x_5$ の OR 演算を行う時に，変数ノード x_1 が否定の関係であることに着目すると IF_THEN_ELSE を導入できる．図7(a) の構文木の変数ノードの数は5となり，IF_THEN_ELSE なしで表現した図7(b) の構文木よりも変数ノードは少なくなる．

(3) 式の論理関数をグラフ化する場合，構築される構文木の変数ノードの数は OR 演算を行う順番によって異なる．

$$F = x_1 + x_2x_3 + \overline{x_2}x_4 \quad (3)$$

$x_1 + x_2x_3$ の OR 演算を先に行った構文木を図8(a) に表し，$x_2x_3+\overline{x_2}x_4$ の OR 演算を先に行った構文木を図8(b) に表す．図8(a) と比べ図8(b) の構文木の変数ノードの数が少ないのは $x_2x_3+\overline{x_2}x_4$ OR 演算を行う過程で IF_THEN_ELSE を導入できたためである．このような場合，自動的に変数ノードの少ない構文木が得られる演算順が選択されることが望ましい．そこで，GBDD の処理系では演算の優先順位の等しい項が3つの時は全ての組み合わせで構文木を構築し，変数ノードの数が最も少ない構文木を演算結果としている．



(a)$(x_1 + x_2x_3) + \overline{x_2}x_4$　(b)$x_1 + (x_2x_3 + \overline{x_2}x_4)$

図 8: 演算順によって異なる構文木

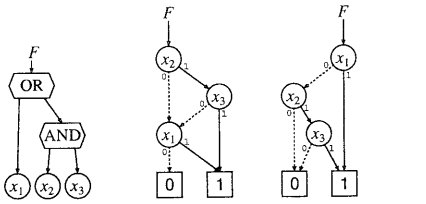
上記の場合を含め，構築される構文木は入力された論理式の形によって変ってくる．このため GBDD の処理系では，既に構築されている演算ノードを展開し，再び演算を行なう「演算ノードの展開」や，OBDD を生成し OBDD から構文木を構築する「OBDD の利用」などを行う．GBDD の処理系ではこれらの手法を使って，より変数ノードの少ない構文木の構築を行っている．

3.3 等価なノードの共有

等価なノードを共有することは全体のノード数の削減につながる．GBDD の処理系での等価なノードの共有は，完成した構文木から GBDD を生成する時に行う．その時，単にグラフ化されたノードに対し共有を試みるのではなく，連結法を適用する時の自由度を生かし，より多くのノードを共有できるように GBDD を生成する．

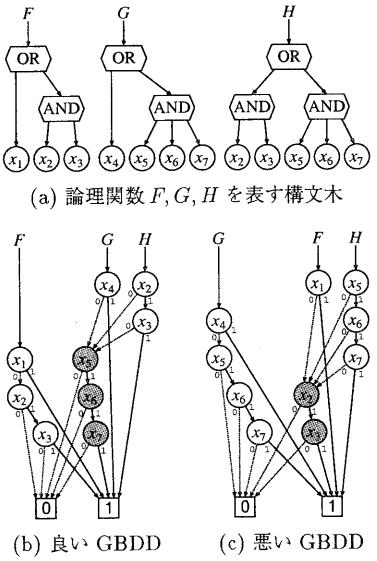
AND, OR の構文木に連結法を適用する場合，結合律が成り立つ部分についてはグラフ化した時の上位下位を選択できる．例えば，図9(a) の構文木に連結法を適用する場合，図9(b) や図9(c) の GBDD を生成することができる．ここで重要なのは図9(b) のようにグラフ化を行うと x_2+x_3 を表すノードは存在しなくなり，図9(c) のようにグラフ化を行うと x_1 を表すノードは存在しなくなることである．以下では，図9(b) は x_1 を優先して生成した GBDD，図9(c) は x_2+x_3 を優先して生成した GBDD と呼ぶことにする．

図10(a) に表す構文木から GBDD を生成する場合，$x_5x_6x_7$ を優先してグラフ化を行うことで部分グラフを共有できる (図10(b))．x_2x_3 を優先してグラフ化した時でも部分グラフを共有できるが (図10(c))，$x_5x_6x_7$ と x_2x_3 の両方を同時に優先することはできない．これは，論理関数 H を表す構文木をグラフ化する時に $x_5x_6x_7$ と x_2x_3 のどちらかが上位になるためである．この場合，より多くのノードを共有できる $x_5x_6x_7$ を優先する．こ



(a) 構文木　(b) x_1 優先　(c) $x_2 + x_3$ 優先

図 9: 構文木から GBDD を構築する時の自由度



(a) 論理関数 F, G, H を表す構文木

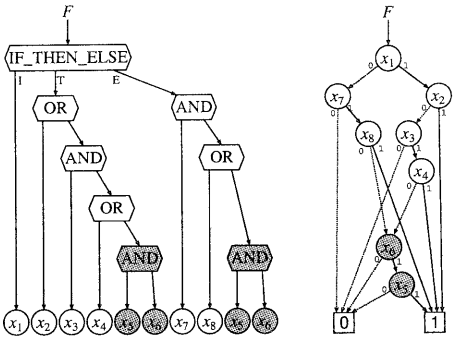
(b) 良い GBDD　(c) 悪い GBDD

図 10: 効果的な共有

れらの情報は構文木を探索することで得られる．図 10の例のように，共有できる形が複数存在する場合はできるだけ多くのノードを共有できる形を選ぶことにより効果的な共有を行う．

図 10の例は異なる論理関数を表すグラフ間での共有であるが，単一の論理関数の中でも共有できる場合がある．それは，構文木の中に IF_THEN_ELSE が含まれる時で，THEN に相当する構文木と ELSE に相当する構文木の中に等価な部分が存在する時に共有を行える．図 11(a) の構文木では x_5x_6 を表す部分が共有できる．THEN に相当する構文木と ELSE に相当する構文木をグラフ化する時に x_5x_6 を表す部分を優先することで，図 11(b) のような共有した GBDD を生成することができる．

図 10や図 11の例では共有に必要な情報は構文木から得ていた．しかし，構文木からの情報だけでは，既にグラフ化されたノードに関して効果的な共有を行えない場合がある．このため，GBDD の処理系では GBDD の全ノードを図 12のように論理関数と対応づけ，可能な限り共有を行っている．共有を行うに時に必要な等価性の判定には既存の OBDD パッケージを利用している．図 12の情報は優先する部分を選択する時にも有効に利用できる．



(a) 構文木　(b) GBDD

図 11: IF_THEN_ELSE に関する共有

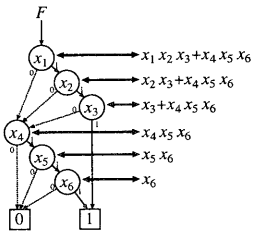


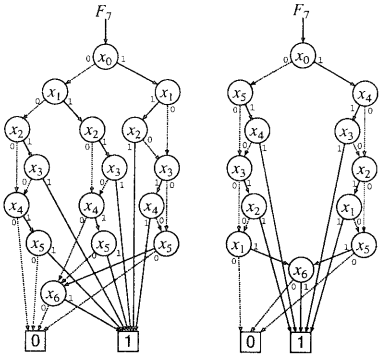
図 12: 各ノードが表している論理関数

4　実現と評価

4.1　実現

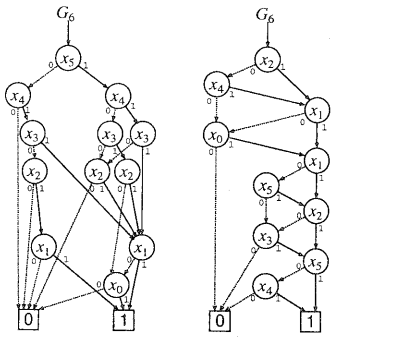
提案した GBDD 生成法のプログラムを C 言語 (約 4,600 行) で計算機上 (CPU: UltraSPARC-II 360 MHz, Memory: 392 MB) に実装し，実験を行った．入力は bem (Boolean Expression Manipulator) フォーマット [8] により記述した論理式によって行い，出力は GBDD を表すテキスト形式ファイルによって行う．

実験結果の比較対象としている OBDD は，動的変数順序付け [9] で得られた OBDD と，OHO (OBDD Heuristics Online) システム [10] を利用して求めた変数順序付けで得られた OBDD のうちノード数のより少ない方であり，ノード数はほぼ最小であると考えられる．



(a) OBDD (16 node)　(b) GBDD (12 node)

図 13: F_7 を表す OBDD と GBDD



(a) OBDD (12 node)　(b) GBDD (10 node)

図 14: G_6 を表す OBDD と GBDD

4.2 評価

まず，(4) 式と (5) 式の論理関数を使って実験を行った．

$$F_7 = x_0(x_1x_2 + x_3x_4 + x_5x_6) + \overline{x_0}(x_2x_3 + x_4x_5 + x_6x_1) \quad (4)$$

$$G_6 = (x_0 + x_1)(x_2 + x_3)(x_4 + x_5)(x_0 + x_2 + x_4)(x_1 + x_3 + x_5) \quad (5)$$

図 13は (4) 式の論理関数から生成した OBDD と GBDD を表し，図 14は (5) 式の論理関数から生成した OBDD と GBDD を表している．いずれの場合も OBDD よりもノードの少ない GBDD を生成できた．

図 13(b) の GBDD は根から葉に至るパスにおいて，異なる変数順を指定することでコンパクトに表現できている (実際，この GBDD は FBDD である)．図 14(b) の GBDD は根から葉に至るパスにおいて，同じ変数が複数回出現することでコンパクトに表現できている．

表 1は (4) 式，(5) 式の論理関数の変数の数を増やした時 (F_n, G_n) の実験結果を表している．表中，ノード数の欄は OBDD，GBDD それぞれの非終端ノードの数を表している．どちらの論理関数でも変数の数が増えるにつれて，GBDD と OBDD のノード数の差は大きくなる．

次に，ISCAS'89 ベンチマークおよび MCNC ベンチマークの論理関数を使って実験を行った．対象は 1 出力の論理関数 680 と多出力の論理関数 49 である．

実験の結果，32 の 1 出力論理関数と 4 の多出力論理関数で OBDD よりも少ないノードの GBDD を生成できた．表 2，表 3はそれぞれ 1 出力および多出力の論理関数をグラフ化した結果のうち，OBDD よりも少ないノードの GBDD を生成できたものを表している．表中，ノード数の欄は OBDD，GBDD それぞれの非終端ノードの数を表している．ノード数の欄の括弧内の数字は図 2の方式でパストランジスタ回路を生成した時のセレクタの数を表しており，値が小さいほどトランジスタ数の削減につながる．

表 1: F_n, G_n での実験結果

論理関数	OBDD ノード数 (node)	GBDD ノード数 (node)		GBDD CPU 時間 (s)
F_7	16	12	(−4)	0.2
F_{11}	28	20	(−8)	0.7
F_{15}	40	28	(−12)	2.1
F_{19}	52	36	(−16)	5.6
F_{23}	64	44	(−20)	13.4
F_{27}	76	52	(−24)	23.5
G_6	12	10	(−2)	0.3
G_9	32	15	(−17)	0.8
G_{12}	55	21	(−24)	3.2
G_{15}	78	27	(−51)	9.6
G_{18}	101	33	(−68)	30.1
G_{21}	124	39	(−85)	96.7
G_{24}	147	45	(−102)	300.8
G_{27}	170	51	(−119)	728.8

実験結果より，1 出力の論理関数では OBDD のノード数を最大で約 23.1%(セレクタの数では約 27.3%)，多出力では最大で約 12.5% (セレクタの数では約 11.0%) 削減した GBDD を自動生成することができた．

5 おわりに

本稿では，変数順や変数の出現回数の制約を持たないより一般的な BDD (GBDD) の生成法について提案し，実験を行った．その結果，与えられた論理関数によっては OBDD のノード数を 70% 削減した GBDD を自動生成することができた．

表 2: OBDD よりもノード数の少ない BDD で表現できた 1 出力の論理関数

論理関数名		OBDD ノード数 (node)		GBDD ノード数 (node)		GBDD CPU 時間 (s)
s1196	o2	143	(137)	141	(136)	281.4
	o4	55	(52)	52	(46)	16.7
	o5	52	(49)	48	(44)	10.8
	o6	31	(30)	27	(25)	10.7
	o7	39	(36)	34	(33)	9.7
	o9	58	(55)	54	(49)	14.4
	o18	10	(7)	9	(7)	0.2
	o19	39	(36)	37	(35)	5.9
	o29	11	(9)	10	(8)	0.2
s1238	o1	55	(51)	51	(47)	12.1
	o3	56	(51)	53	(47)	16.3
	o4	53	(49)	51	(45)	10.1
	o7	31	(30)	28	(27)	10.7
	o8	39	(36)	34	(33)	9.7
	o9	57	(54)	53	(47)	12.9
	o18	10	(7)	9	(7)	0.2
	o19	38	(36)	37	(35)	5.9
	o29	11	(9)	10	(8)	0.2
s510	o3	12	(10)	11	(9)	0.2
	o4	11	(8)	10	(7)	0.3
	o5	10	(8)	9	(8)	0.2
	o9	16	(13)	15	(13)	0.7
s820	o21	42	(37)	41	(37)	9.1
s832	o21	42	(37)	41	(37)	9.1
s953	o3	29	(26)	27	(25)	1.0
Z5xp1	o12	16	(12)	15	(13)	2.7
apla	o13	16	(15)	15	(14)	0.7
	o15	16	(14)	15	(14)	0.4
dc2	o12	19	(17)	18	(17)	11.5
f51m	o10	21	(19)	18	(16)	4.6
	o12	13	(11)	10	(8)	0.3
misex1	o12	10	(8)	9	(8)	0.2

また，ISCAS'89 ベンチマーク，MCNC ベンチマークの論理関数を用いた実験において，OBDD よりも少ないノードの GBDD を自動生成することができ，実用的な論理関数においてもある程度の有効性を示せた．

実験では GBDD と OBDD のノード数比較を行ったが，FBDD との比較を行う必要もある．(4) 式，(5) 式をグラフ化した実験では，F_n から生成した GBDD は FBDD であった．また，ISCAS'89 ベンチマーク，MCNC ベンチマークの論理関数をグラフ化した実験では，OBDD よりもコンパクトに表現できた 36 の論理関数のうち 20 の論理関数では結果として FBDD を生成していた．

今後は FBDD とのノード数比較やどのような論理関数が GBDD でコンパクトに表現できるかなどの，処理系としての評価が課題になる．

表 3: OBDD よりもノード数の少ない BDD で表現できた多出力の論理関数

論理関数名	OBDD ノード数 (node)		GBDD ノード数 (node)		GBDD CPU 時間 (s)
s208	57	(52)	52	(49)	5.7
s420	112	(109)	100	(97)	440.4
con1	16	(12)	14	(12)	0.2
sex	50	(45)	47	(43)	1.4

謝辞 本研究を行うにあたり，日頃から御討論いただいた広島市立大学コンピュータアーキテクチャ講座の諸氏に感謝いたします．

参考文献

[1] S.B. Akers : "Binary decision diagrams," IEEE Trans. Comput., vol. C-27, no. 6, pp. 509-516, June 1978.

[2] R.E. Bryant: "Graph-based algorithms for Boolean function manipulation", IEEE Trans. Comput., vol.C-35, no.8, pp.677–691, Aug. 1986.

[3] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu: "A 3.8-ns CMOS 16×16-b multiplier using complementary pass-transistor logic", IEEE J. Solid-State Circuits, vol.25, no.2, pp.388–395, Apr. 1990.

[4] K. Yano, Y. Sasaki, K. Rikino, and K. Seki: "Top-down pass-transistor logic design", IEEE J. Solid-State Circuits, vol.31, no.6, pp.792–803, June 1996.

[5] 瀧和男, 李副烈: "パストランジスタ論理に基づく低消費電力回路方式と設計事例", 電子情報通信学会論文誌 A 分冊, vol.J80-A, no.5, pp.1–12, 1997 年 5 月.

[6] J.R. Burch: "Using BDDs to verify multipliers", Proc. 28th ACM/IEEE Design Automation Conf., pp.408–412, 1991.

[7] J. Gergov and C. Meinel: "Efficient Boolean manipulation with OBDDs can be extended to FBDDs", IEEE Trans. Comput., vol.43, no.10, pp.1197–1209, 1994.

[8] 湊, 石浦, 矢島 : "共有二分決定図を用いた記号シミュレーション," 1989 年秋期信学全大, pp. 1.206-207, 1989.

[9] R. Rudell: "Dynamic variable ordering for ordered binary decision diagrams", Proc. ICCAD'93, pp. 42–47, Nov. 1993.

[10] A. Wagner: "The OHO system —OBDD heuristics online—", http://kalliope.uni-trier.de/~wagner/html/bdd_html/oho.html.