積和形論理式の非明示的表現における効率的カーネル生成法

An Efficient Method for Generating Kernels on Implicit Cube Set Representations

澤田 宏，山下 茂，名古屋 彰
Hiroshi Sawada, Shigeru Yamashita, Akira Nagoya
NTT コミュニケーション科学基礎研究所
NTT Communication Science Laboratories

1 はじめに

論理合成・最適化の処理において，複数の論理関数から共通の論理を抽出することは，規模の小さな論理回路を得るために重要である．現在用いられている手法のうち多くのものは，積和形論理式表現に基づいている．それらは基本的に，個々の論理式に対してカーネル [1] と呼ばれる部分論理式を生成し，複数のカーネルの間に同じ積項があれば，それを共通の論理式として抽出する．また，抽出する論理式の候補を，二つの積項の論理式と二つのリテラルの積項だけに限定し，効率良く処理を行う手法 [2] も提案されている．

積和形論理式は積項の集合と見なすことができ，一般にはその集合を明示的に列挙して扱う．しかし，加算器などの XOR 演算を多く含む回路を積和形論理式にすると，積項の数が膨大で扱えきれない場合がある．そのため，積項の集合を BDD (Binary Decision Diagram) [3] やそれに似たグラフ構造を用いて非明示的に表現する方法がいくつか提案された [4]，[5]．その結果，扱える積和形論理式の規模が増大し，適用範囲が広がった．

しかしながら，共通論理を抽出する手法の多くは，積項を明示的に列挙して扱うものが多く，非明示的表現の特長を活かしているもの [6] は少ない．そこで本稿では，積和形論理式の非明示的表現において，効率的にカーネルを生成する手法を提案する．本手法は，カーネルグラフと名付けたグラフ表現を用いて，カーネルの生成過程を記憶することを特徴とする．これにより，過去に行ったことのある処理と同じようなカーネル生成の処理が要求された場合は，その処理を省略することが可能となる．

以下では，まず 2 章で準備として，積和形論理式とその非明示的表現，およびカーネルについて説明する．3 章では提案するカーネルグラフについて詳細を述べる．4 章では，カーネルグラフを用いて，共通論理を抽出する一手法について述べる．5 章では共通論理の抽出に関する実験結果を，6 章では結論を述べる．

2 準備

2.1 積和形論理式とその非明示的表現

まず，積和形論理式について述べる．リテラルとは，0 か 1 の値を取る変数 $(a, b, c, \cdots)$ およびその否定 $(a', b', c', \cdots)$ である．積項とはリテラルの積であり，例えば $ab'de$ や cde や f は積項である．積和形論理式は積項の和であり，例えば $ab'de + cde + f$ は積和形論理式である．本稿では，文脈から明らかであれば，積和形論理式を単に論理式と呼ぶことがある．

積和形論理式は積項の集合である．その非明示的表現とは，積項の集合をグラフ構造を用いて表現するものであり，Meta-product によるもの [4] や Zero-suppressed BDD によるもの [5] が提案されている．これらの表現では，各積項は定数 1 を表現する節点への各経路に対応しているため，積項の数が膨大な論理式でも少ない節点数で表現できることがある．また，同じ論理式は同じ節点で表現されているため，論理式の等価性判定が節点に割り当てられた識別子 (ポインタ等) の比較だけで行える．論理式の除算等の基本的な演算は，再帰的処理により行うことができ，演算結果を一時的に保存するキャッ

```
/* Sop: 積和形論理式 */
void genKernel(Sop K, int index) {
    for ( int i = index; i > 0; i − − ) { /* index以下のみに着目する */
        if ( literalCount(K, l_i) ≥ 2 ) { /* l_i が複数回現れるリテラルかどうか */
            Sop childF = K/l_i;
            Sop comCube = commonCube( childF ); /* childF のすべての積項に現れるリテラルを求める */
            if ( topLiteral( comCube ) < i ) { /* comCube のすべてのリテラルの添字が i より小さいか */
                genKernel( childF/comCube, i − 1);
            }
        }
    }
    registerKernel(K); /* カーネルとして登録 */
}
```

図 1: カーネル生成アルゴリズム

シュテーブルを用いることで効率良く行える．

2.2 カーネル

まず，cube-free という概念について述べる．積和形論理式において，すべての積項に現れるようなリテラルが存在しないとき，その論理式は cube-free であるという．例えば，$ab + a'c$ は cube-free である．一方，$ab + bc$ は，b がすべての積項に現れているので cube-free ではない．また，abc は積項が 1 つだけで，どのリテラルもすべての積項に現れていることになるので cube-free ではない．

与えられた論理式をある積項で割った商のうち，cube-free であるものはカーネルと呼ばれる [1]．例えば，$F_1 = abcg' + abd + ae + bc'fg + c'fh$ に対して，$F_1/a = bcg' + bd + e$ や $F_1/ab = cg' + d$ はカーネルである．一方，$F_1/c' = bfg + fh$ は，cube-free ではないのでカーネルではない．また，カーネルを得るために用いた積項はコカーネルと呼ばれる．上記の例では，コカーネル ab を用いてカーネル $cg' + d$ が作られたことになる．

与えられた論理式に対してすべてのカーネルを生成する手順が知られている．これを以下に示す．

1. 与えられた論理式 F が cube-free でなければ，すべての積項に現れるリテラルで論理式を割り，商の論理式を K とする．F が cube-free であれば，F をそのまま K とする．
2. 論理式 K の全てのリテラルを順序づけして，$l_n, l_{n-1}, \ldots, l_1$ とする．
3. 図 1に示す再帰的手続きを $genKernel(K, n)$ で呼び出す．

図 1に示す $genKernel()$ は，与えられた論理式を複数回現れるリテラルで割り，その結果の論理式が cube-free でなければ，すべての積項に現れるリテラルでさらに割り，結果に対し

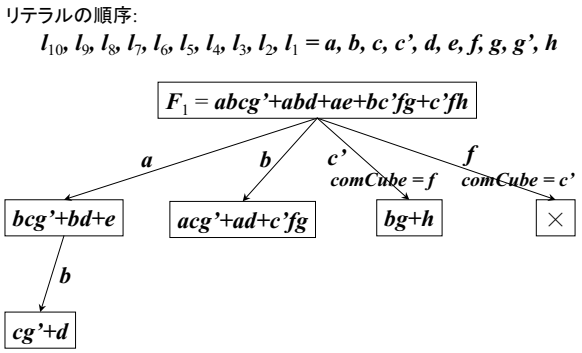

図 2: カーネル生成の例

$genKernel()$ を再帰的に呼び出している．ここで，$literalCount(K, l_i)$ は論理式 K に現れるリテラル l_i の個数を，$commonCube(F)$ は論理式 F のすべての積項に現れるリテラルの積を，$topLiteral(F)$ は論理式 F に現れるリテラルの添字の最大値を，それぞれ返す．$genKernel()$ では，同じカーネルを 2 回以上登録しないように，リテラルを順序付けして制御を行っている．着目するリテラルは添字が $index$ 以下のものだけであり，添え字が i 以上のリテラルが $comCube$ に存在すれば以降の処理は行わない．

例として，論理式 $F_1 = abcg' + abd + ae + bc'fg + c'fh$ に対してすべてのカーネルを生成する様子を図 2に示す．与えられた論理式は cube-free であるため，これをそのまま K とする．リテラルの順序は図 2に示すものとし，$genKernel(F, 10)$ を呼び出す．複数回現れるリテラルで添字が 10 以下のものは，a, b, c', f であり，それぞれについて処理を進める．c', f で割った結果は cube-free ではないため，すべての積項に現れるリテラルの積 $comCube$ が発生する．f の場合は，添字が 4 $(f = l_4)$ より大きいリテラル $c' = l_7$ が $comCube$ に存在するため，

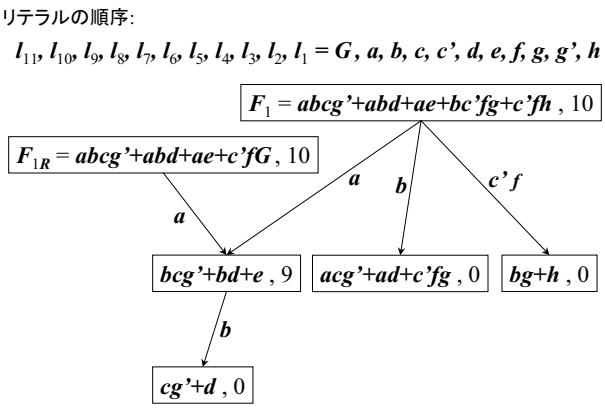

図 3: カーネルグラフの例 1

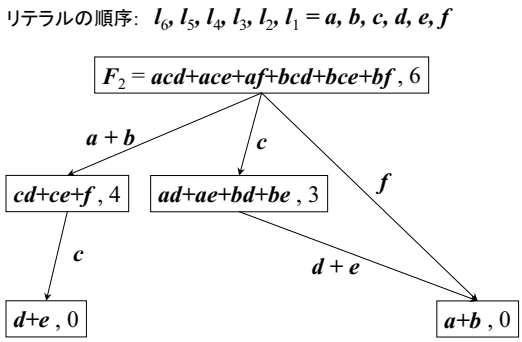

図 4: カーネルグラフの例 2

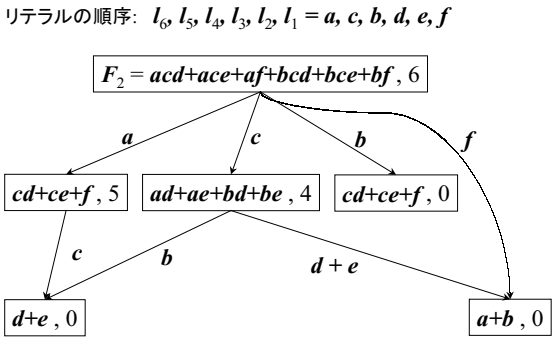

図 5: カーネルグラフの例 3

以降の処理は行わない.

3 カーネルグラフ

本章では，カーネルグラフと名付けた有向グラフを用いてカーネル生成の過程を記憶することで，非明示的表現で与えられた論理式に対するすべてのカーネルを効率的に生成する手法を提案する.

3.1 定義

カーネルグラフは，以下に定義する節点と有向枝で構成される.

節点 論理式 K と整数値 $topIndex$ を持つ.

有向枝 論理式 D を持つ.

論理式 K はその節点に対応するカーネルである．整数値 $topIndex$ は，その節点から出る枝において，論理式 D に現れるリテラルの添え字で最大のものであり，出る枝が無い場合には 0 となる．以下，これら K と $topIndex$ を持つ節点を $node(K, topIndex)$ と書くことにする.

$node(K, topIndex)$ からは，以下の三つの条件をすべて満たすリテラル l_i に関して枝が出ている.

1. $i \leq topIndex$,
2. $literalCount(K, l_i) \geq 2$
3. $topLiteral(comCube) < i$,
   ただし $comCube = commonCube(K/l_i)$

リテラル l_i に関する枝は，論理式 D の中に積項 $l_i \cdot comCube$ を持つ．論理式 K を持つ節点から出ていて論理式 D を持つ枝は，論理式 K/D を持つ節点を指し示す.

以下の条件が共に成り立つ二つの節点は，等価であるとする.

- 論理式 K が同じである.
- 整数値 $topIndex$ が同じである.

$topIndex$ を前記のように定義したことにより，等価である二つの節点では，それらから出るすべての枝に関して，論理式 D が同じであり，それが指し示す節点も等価である．カーネルグラフにおいては，二つの等価な節点は共有されて一つになるものとする．例として，図 3における節点 $node(bcg' + bd + e, 9)$ は，論理式 F_1 と F_{1R} を持つ節点から出ている枝で共有されている.

3.2 生成アルゴリズム

カーネルグラフの生成アルゴリズムは，2.2節で紹介したカーネルを生成するアルゴリズムと大枠は同じである．違いは，再帰的手続きである $genKernel(K, n)$ が，図 6に示す $genKernelGraph(K, n)$ に置き換わることである．$genKernelGraph()$ では，$genKernel()$ の処理に加えて，節点へのポインタを結果として返すという処理と，$topIndex$ を求めて等価な節点が既にあるかどうかを調べる処理を行っている.

まず，$topIndex$ の求め方について述べる．着目

```
/* KGnode*: カーネルグラフの節点へのポインタ */
/* Sop: 積和形論理式 */
KGnode* genKernelGraph(Sop K, int index) {
    KGnode* resultNode = 0; /* 結果となる節点へのポインタが入る */
    int topIndex = 0;
    for ( int i = index; i > 0; i -- ) {
        if ( literalCount(K, l_i) ≥ 2 ) { /* 条件 2 */
            Sop childF = K/l_i;
            Sop comCube = commonCube( childF );
            if ( topLiteral( comCube ) < i ) { /* 条件 3 */
                if ( topIndex == 0 ) { /* 初めてここに来たとき */
                    topIndex = i; /* i は減少していくのでこれが最大値 */
                    resultNode = lookup(K, topIndex); /* 等価な節点が既に存在するかどうかを調べる */
                    if ( resultNode ) return resultNode; /* もしあればそれを返し処理を終える */
                    resultNode = makeNode(K, topIndex); /* 新たな節点を作成 */
                }
                KGnode* childNode = genKernelGraph( childF/comCube, i − 1);
                addChild(resultNode, childNode, l_i · comCube);
            }
        }
    }
    if ( topIndex == 0 ) { /* 枝が無い場合 */
        resultNode = lookup(K, 0); /* 等価な節点が既に存在するかどうかを調べる */
        if ( resultNode ) return resultNode; /* もしあればそれを返し処理を終える */
        resultNode = makeNode(K, 0); /* 新たな節点を作成 */
    }
    return resultNode;
}
```

図 6: カーネルグラフ生成アルゴリズム

するリテラルの添字 i は減少していくので，前記の条件 2 と 3 を初めて満たしたリテラルの添字が *topIndex* となる．また，どのリテラルに関しても条件 2 と 3 を満たさなかった場合は， *topIndex* は初期値である 0 のままとなる．

topIndex が決定した後は， $node(K, topIndex)$ が既に存在するかどうかを調べる．論理式を非明示的に表現している場合は，論理式 K の識別子と整数値 *topIndex* のペアでカーネルグラフの節点の等価性判定を行うため，ハッシュテーブルを用いることで効率良く調べることができる．もし等価な節点が存在すれば，以降の処理は省略してその節点を結果として返す．存在しなければ，その節点を新たに作り再帰的に処理を進めていく．なお， *addChild*(*resultNode*, *childNode*, *cube*) は，節点 *resultNode* から *childNode* への枝がなければ，それを作り，その枝が持つ論理式 D を *cube* とする．枝がある場合は，その枝が持つ論理式 D を $D + cube$ に更新する．

3.3 例

まず，以前のものから少しだけ変化した論理式に対するカーネル生成において，処理が省略される例を示す．論理式 $F_1 = abcg' + abd + ae + bc'fg + c'fh$ が与えられ，図 3に示すカーネルグラフを生成したとする．その後，共通論理式として $G = bg + h$ が選ばれ， F_1 は $F_{1R} = abcg' + abd + ae + c'fG$ と更新されたとする．ここで，更新された F_{1R} に対してカーネルグラフを生成することを考える．まず，図 3に示すようにリテラルの順序付けを行い， $genKernelGraph(F_{1R}, 11)$ を呼び出す．複数回現れるリテラルは $l_{10} = a$ と $l_9 = b$ であるから， *topIndex* は 10 となる． $node(F_{1R}, 10)$ が存在するかを調べると存在しないので新たにこの節点を作る．その後，リテラル $l_{10} = a$ に対し $genKernelGraph(bcg' + bd + e, 9)$ を呼び出す．複数回現れるリテラルは $l_9 = b$ だけであり， $node(bcg' + bd + e, 9)$ が存在するかを調べる．この場合は，この節点が既に存在しているので以降の処理を省略できる．

また，一つの論理式に対するカーネルの生成過程においても，同じようなカーネルが生成され，処理が省略される例を示す．図 4に，論理式 $F_2 = acd + ace + af + bcd + bce + bf$ に対するカーネルグラフを示す． $node(cd + ce + f, 4)$ が論理式 $a + b$ を持つ枝で指し示されているということは，カーネル

F_2/a と F_2/b に対する処理において，生成すべき節点が同じであったことを示す．

これまでの例では，同じ論理式 K に対して違う節点が割り当てられることはなかった．多くの場合はそうであるが，違う節点が割り当てられることもある．図 4の論理式 F_2 において，リテラル b と c の順序を入れ替えると，図 5に示すカーネルグラフになる．ここでは，カーネル $cd + ce + f$ に対して，二つの節点 $node(cd + ce + f, 5)$ と $node(cd + ce + f, 0)$ が存在している．これらはリテラル c に関する枝を持っているものといないものであり，等価ではない．この例より，節点の等価性の判定のためには，整数値 $topIndex$ の比較も必要であることがわかる．

4 共通論理式の抽出処理への適用

提案した手法により，積和形論理式の非明示的表現から効率的にカーネルを生成することが可能となった．本章では，これを共通論理式の抽出に適用することを考える．

ここでは抽出するための基本的な枠組みとして，2 積項と 2 リテラル積項の並行抽出法 [2] を用いる．この手法では，抽出する論理式を，2 つの積項だけからなるものと，2 つのリテラルだけから成る 1 つの積項に限っている．その結果，1) 抽出する論理式において否定の関係を認識できる，2) すべての 2 積項を求めるための計算は，初めに一度だけ行い，後は差分の情報を用いて部分的に更新していくことが可能である，などの利点を持つ．

初めの 2 積項を求めるための計算では，すべての積項のペアに対し，双方の積項に現れるリテラルで割って登録するという操作を行う．しかし，積和形論理式の非明示的表現では，グラフで積項の集合を表現しているため，この操作を直接行うことはできない．この操作を行うために個々の積項を取り出すと，結局明示的な表現になってしまい，非明示的表現の利点を活かすことができなくなる．

そこで我々は，前章で提案したカーネル生成手法を用いてすべてのカーネルを生成し，以下のようにして積項のペアを作成した．例えば，$a+b+c$ のような複数回現れるリテラルを持たないカーネルに対しては，そのまま 2 積項 $a+b, a+c, b+c$ を作成する．また，$ac + ad + bc + bd + e + f + g$ のような複数回現れるリテラルを持つカーネルに対しては，複数回現れるリテラルを含む積項を取り除いた論理式 $e + f + g$ を作成し，2 積項 $e + f, e + g, f + g$ を作成する．このようにすることで，明示的に表現すべき論理式の規模を小さくすることができる．

ただし，上記の例での $ad + bc$ など，作成されない 2 積項が出てくる．しかし，このような 2 積項を抽出してしまうと，上記の例の $c + d$ など，他のより良い 2 積項の抽出を妨げることが多いので，これらを作成しなくてもそれほど大きな影響はないと考える．

5 実験結果

表 1に，ベンチマーク回路 [7] に対して実験を行った結果を示す．“回路” に，回路の名前 (“名前”)，入力数 (“入”)，出力数 (“出”) を示す．積項を明示的に列挙する手法ではとても扱えないものも含め，比較的規模の大きい積和形論理式や多段論理回路を対象とした．本実験では，まず回路構造を読み込み，外部出力が表す論理関数を BDD で表現し．その後，[8] の手法により，BDD から非冗長な積和形論理式を生成した．“抽出前” には，その生成されたばかりの積和形論理式に関する値を記す．“リテラル” は，積和形論理式のリテラル数の総和である．“コカーネル” は，すべてのコカーネルの総数であり，“節点” は，そのときにカーネルグラフで用いられていた節点の数である．

前章で述べた手法を用いて論理の抽出を行った結果を “抽出後” に示す．実行時間 (“時間”) は，Linux を OS とした 450MHz の Pentium II を持つシステムで測定した．“ハッシュテーブル” には，カーネルグラフを作るアルゴリズムで用いているハッシュテーブルに関する統計情報を示す．“参照” はテーブルを参照した回数，“ヒット” は要求した節点が既に存在したときの回数を示す．

提案手法により，規模の大きな積和形論理式に対しても，非明示的に表現されていれば，うまく論理の抽出ができることが分った．いくつかの例では，非常に多くのコカーネルが，それらの数に対して十分に少ない数の節点で表現されている．特に，本手法では，16 ビットのパリティ関数 (“parity”) のすべてのコカーネルを表現することに成功した．これらは，4 千万個程の積項であるが，非明示的表現により，効率良く表現されている．ハッシュテーブルのヒット率は，多くの例において十分に高く，これにより，カーネルの生成過程を記憶することは有効であると考えられる．

表 1: 実験結果

回路			抽出前			抽出後	時間	ハッシュテーブル		
名前	入	出	リテラル	コカーネル	節点	リテラル	(秒)	ヒット	参照	率
alu4	14	8	4961	16096 /	14646	1256	10.40	33625 /	83168	= 0.404
apex1	45	45	7017	6178 /	5558	1759	6.18	25595 /	46326	= 0.552
apex2	39	3	14728	14410 /	5042	358	4.91	10157 /	22651	= 0.448
apex3	54	50	4621	4888 /	4822	1718	4.05	13421 /	30405	= 0.441
apex4	9	19	7887	22235 /	21568	2598	11.25	50441 /	121497	= 0.415
apex5	117	88	7202	1321 /	946	968	2.42	4443 /	10103	= 0.440
cordic	23	2	18213	105654 /	8727	88	1.28	7976 /	17725	= 0.450
cps	24	109	6747	1458 /	1276	1415	3.36	8081 /	15088	= 0.536
dalu	75	16	24902	24976 /	9941	696	5.46	17325 /	36094	= 0.480
ex1010	10	10	4048	5891 /	5838	1892	4.88	24121 /	44922	= 0.537
frg2	143	139	27803	9212 /	1603	893	2.76	9494 /	14915	= 0.636
misex3	14	14	12101	17195 /	13185	1113	8.92	30346 /	79301	= 0.383
pair	173	137	130724	29796 /	7909	3126	37.67	85739 /	155161	= 0.553
parity	16	1	524288	42456897 /	131037	60	75.93	294292 /	523375	= 0.562
seq	41	35	17876	11615 /	8299	1972	12.14	41976 /	88383	= 0.475
spla	16	46	4608	2242 /	1436	634	1.75	3366 /	8509	= 0.396
t481	16	1	4752	4746 /	698	40	0.75	2522 /	4056	= 0.622
table3	14	14	5805	14988 /	14637	1265	9.82	22475 /	76906	= 0.292
table5	17	15	6319	14090 /	13195	1045	9.88	18820 /	63691	= 0.295

6 おわりに

グラフ表現を用いてカーネルの生成過程を記憶しておくことで，論理式の非明示的表現に対して効率的にカーネルを生成する手法を提案した．本手法は以下のような特長がある．

1. 過去に行ったことのある処理と同じカーネル生成の処理を省略でき，実行時間を削減できる．
2. カーネルグラフの節点の共有により，論理式のカーネル / コカーネルの関係を表現するための必要記憶量を削減できる．

論理式の非明示的表現では，それに対応する論理関数との相互変換が，明示的な表現に比べて容易である．今後は，共通論理式の抽出処理において，論理関数処理をうまく取り入れてより強力な手法となるよう検討を進めていく．

参考文献

[1] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Trans. CAD*, CAD-6(6):1062–1081, November 1987.

[2] J. Rajski and J. Vasudevamurthy. The Testability-Preserving Concurrent Decomposition and Factorization of Boolean Expressions. *IEEE Trans. CAD*, 11(6):778–793, June 1992.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, C-35(8):667–691, August 1986.

[4] O. Coudert and J. C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. Design Automation Conf.*, pages 36–39, June 1992.

[5] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. Design Automation Conf.*, pages 272–277, June 1993.

[6] S. Minato. Fast Factorization Method for Implicit Cube Set Representation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 15(4):377–384, April 1996.

[7] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. MCNC, January 1991.

[8] S. Minato. Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams. *IEICE Trans. on Fundamentals*, E76-A(6):967–973, June 1993.