システムＬＳＩ設計技術　94－4
(2000. 1. 11)

順序回路の状態探索向けBDDの動的変数順序づけ手法

樋口 博之 †　Fabio Somenzi ‡

†(株) 富士通研究所
‡University of Colorado at Boulder

〒211-8588 川崎市中原区上小田中 4-1-1
Phone: 044-754-2663
Email: higuchi@flab.fujitsu.co.jp

あらまし
順序回路の状態探索のための二分決定グラフ (BDD) の動的変数順序づけ手法を提案する。従来、BDD を用いた状態探索では、現状態変数とそれに対応する次状態変数の組それぞれをグループ化して扱う方法と現状態変数と次状態変数を完全に独立に扱う方法があった。本手法では、現状態変数と次状態変数の組それぞれについてグループ化するかしないかを動的変数順序づけ実行時に決定することにより、状態探索の性質に応じた変数順序づけを行う。実験結果より、無条件に全ての状態変数の組をグループ化したり非グループ化する方法に比べ、状態探索全体に要する時間を短縮できることが分かった。

キーワード 順序回路、有限状態機械、状態探索、像計算、二分決定グラフ、動的変数順序づけ

Lazy Group Sifting for Efficient Symbolic Traversal of Sequential Circuits

Hiroyuki Higuchi †　Fabio Somenzi ‡

†Fujitsu Laboratories Ltd.
‡University of Colorado at Boulder

4-1-1, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, JAPAN
Phone: +81-44-754-2663
Email: higuchi@flab.fujitsu.co.jp

Abstract

This paper proposes lazy group sifting for dynamic variable reordering during state traversal. The proposed method relaxes the idea of pairwise grouping of present state variables and their corresponding next state variables. This is done to produce better variable orderings during image computation without causing BDD size blowup in the substitution of next state variables with present state variables at the end of image computation. Experimental results show that our approach is more robust in state traversal than the approaches that either unconditionally group variable pairs or never group them.

key words sequential circuit, state traversal, image computation, binary decision diagram, variable reordering

1 はじめに

順序回路の状態探索は、論理検証、論理合成、およびテスト生成などに用いられ、論理回路の CAD において最も重要な要素技術のうちの一つである。近年の集積回路技術の進歩に伴い、より大規模なシステムが順序回路として実現されるようになり、大規模な順序回路の効率的な状態探索技術が望まれている。

順序回路の状態探索とは、与えられた初期状態の集合から到達可能な状態を探索していくことである。これは、与えられた順序回路の状態遷移グラフ上での探索問題として定式化することができる。しかし、フリップフロップ (Flip-Flop; FF) の数に対して状態数は最悪指数倍になるため、古典的なグラフアルゴリズムにより明示的 (explicit) に扱うことが現実的には不可能であった。

1989 年、Coudert らは、状態集合を二分決定グラフ (Binary Decision Diagram; BDD) により非明示的 (implicit) に表現し、BDD の処理に基づく幅優先探索を行うことにより状態探索する方法を提案した [1]。この方法は symbolic traversal と呼ばれている。symbolic traversal により、扱える順序回路の大きさは飛躍的に増大したが、設計される順序回路の規模とのギャップは依然として大きく、計算途中で BDD のサイズが爆発してしまう場合も多かった。

BDD のサイズは BDD の変数順序に大きく依存するため、効率的な symbolic traversal を行うには良い変数順序を求めることが必要不可欠である。状態探索を行う前に良い変数順序を求める静的変数順序づけ法がいくつか提案されているが [2, 6]、symbolic traversal では遷移関係を表す関数と状態集合を表す特徴関数という全く異なる関数を扱う上に、扱う状態集合自体も探索とともに変化していくため、それらの関数全てに対して良い変数順序を事前に求めておくことは大変難しい。そのため、symbolic traversal では、Rudell が提案した動的変数順序づけ法 [7] が用いられることが多い。動的変数順序づけ法は BDD の演算中に BDD のサイズが大きくなった時点で変数の再順序づけを行う。動的変数順序づけは symbolic traversal 途中での BDD のサイズを大幅に削減することができることが実験的に示されている。しかし、動的変数順序づけは BDD の隣接変数のスワップ操作を繰り返し行う必要があるため、symbolic traversal に要する時間の大部分が変数の再順序づけのための時間になってしまうことも少なくない。このため、近年では symbolic traversal 向けの動的変数順序づけ法の研究が注目を集めている [8]。

symbolic traversal における変数順序づけ固有の問題として、現状態変数と次状態変数をどのように扱うかという問題がある。symbolic traversal では状態集合の像を表す関数が次状態変数の関数として計算されるため、像計算の後に各次状態変数を現状態変数で置き換える操作が必要である。そのため、次状態変数をそれに対応する現状態変数とグループ化して変数順序を決定することが多い。この方法では、変数の置き換えの前後で次状態変数同士の相対的順序と現状態変数同士の相対的順序が同一なため、変数置き換えによって像を表す BDD の形は変わらない。しかし、この方法では状態対のグループ化により BDD の動的変数順序づけの際に変数順序が制約される。従って、像計算中に起こった動的変数順序付けの BDD サイズの削減能力が制限される可能性がある。一方、状態対のグループ化を全く行わない方法では、現状態変数とそれに対応する次状態変数が動的変数順序づけにおいて独立に扱われるため、像計算中の BDD サイズの削減能力は制限されない。しかし、現状態変数同士の相対的位置と次状態変数同士の相対的位置が一般に異なるため、像計算後の変数の置き換えによって像を表す BDD のサイズが爆発する可能性がある。そこで、本稿では、symbolic traversal の動的変数順序づけにおける状態対の扱い方に関し、状態対ごとにグループ化するかグループ化しないかを動的変数順序づけの最中に決定することにより、従来よりも良い変数順序を求める方法を提案する。この方法により、対象回路の性質およびその状態探索の計算の特性に合わせた状態対の扱いが可能となる。

全ての状態対をグループ化する方法と全ての状態対を独立に扱う方法を比較した場合、前者の方が順序回路の状態探索には有効であることが経験的に知られている [9]。本稿では、提案手法がそれら二つの方法より有効であることを実験結果により示す。

以下、2 章では、いくつかの定義と基本的な symbolic traversal アルゴリズムおよび BDD の動的変数順序づけアルゴリズムについて説明する。3 章では、提案手法である lazy group sifting について述べる。4 章では、実験結果を示し、それに関しての考察を行う。最後に 5 章でまとめと今後の課題を述べる。

2 準備

2.1 順序回路

本稿で取り扱う順序回路は単相クロックによる完全同期式順序回路であり、論理ゲートと結線により構成される組合せ回路部とフリップフロップ (FF) により構成される状態記憶部から成る。この回路の外部入力数を m、FF 数を n とする。

m 本の外部入力に対応する論理変数と、n 個の FF が現在持つ状態に対応する論理変数 (現状態変数) をそれぞれ $w_i (i = 1, 2, \ldots, m)$、$x_j (j = 1, 2, \ldots, n)$ と表す。さらに、これらをそれぞれ一まとめにしてベクトル表記し、外部入力変数ベクトル w、現状態変数ベクトル x と呼ぶことにする。すなわち、$w = (w_1, w_2, \ldots, w_m)$, $x = (x_1, x_2, \ldots, x_n)$ である。次状態 y_j の論理関数を $y_j = f_j(x, y)$; $j = 1, 2, \ldots, n$ とする。

2.2 BDD による集合表現

本稿では、特に断らない限り、すべての論理関数処理および集合処理は、二分決定グラフ (BDD) [10] を用いて行うものとする。BDD は論理関数の有向グラフによる表現である。これはブール展開 (ある入力変数に、0,1 の値を代入して 2 つの部分関数を得る手続き) を再帰的に繰り返すことにより得られる二分決定木のグラフを縮約したものである。本稿では、既約な順序付き二分決定グラフを単に二分決定グラフと呼ぶ。図 1 に BDD の例を

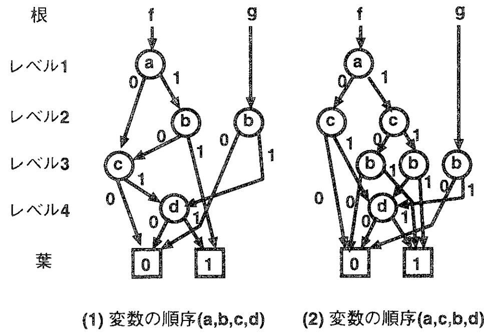

図 1: BDD の例 ($f = ab + cd, g = bd$)

示す。図のように、葉を除く BDD の各ノードは論理変数でラベル付けされている。BDD では根から葉に至る全てのパス上で論理変数の出現順は、ある全順序に従う。この順序を**変数順序**と呼ぶ。図 1(1) では変数順序は根の方から a,b,c,d、(2) では a,c,b,d である。BDD のノード数はそれぞれ 7 と 9 であり、BDD のノード数が変数順序に依存することが分かる。また、変数順序に従ってある変数に付けられた番号をこの変数のレベルと呼ぶ。

BDD を用いた論理関数の演算として、通常用いられる論理和 (+)、論理積 (·)、論理否定 (¯)、排他的論理和 (⊕)、等価演算 (≡) の他に、smoothing 演算を用いる。論理関数 $f : B^n \to B$ の入力変数のある部分集合が $s = \{x_1, \ldots, x_k\}$ であるとき、F の s に関する smoothing (existential quantification) $\exists s$ を以下のように定める:

$$\exists x_i.f = f_{x_i} + f_{\bar{x}_i}$$
$$\exists s.f = \exists x_1.\cdots\exists x_k.f.$$

n 変数論理関数 $f : B^n \to B$ は f を 1 にする入力ベクトルの集合、すなわち最小項 (minterm) の集合と見ることができる。ある 2 値ベクトルの集合 $C \subseteq B^n$に対し、

$$x \in C \Leftrightarrow \chi_C(x) = 1$$

なる n 変数論理関数χ_Cを集合 C の特徴関数とよぶ。従って、ある 2 値ベクトルの集合 C に対し、C の特徴関数を BDD で表現すれば C が BDD で表現できたことになる。この表現は集合の BDD により非明示的表現とも呼ばれる。例えば、ある 2 値ベクトルの集合 C = {000, 010, 100, 110, 111} に対する特徴関数は、i ビット目に変数 x_i を割り当てると、$\chi_C = x_1 \cdot x_2 + \overline{x_3}$となる。以下では、集合 C と特徴関数χ_Cを同一視し、χ_C も単に C と書く。

2.3 symbolic traversal の基本アルゴリズム

symbolic traversal による順序回路の状態空間の探索は像計算の繰り返しにより行うことができる。状態集合 $From$ の状態遷移関数 $f = (f_1, f_2, \ldots, f_n)$ による像 $Image(From, f)$ は以下のような状態集合である:

$$\{y \in \{0,1\}^n | y = f(w,x), w \in \{0,1\}^m, x \in From\}.$$

Traverse($S_0(x)$, $f(w,x)$) {
1. $Reach(x) = From(x) = S_0(x)$;
2. **while** (1) {
3. $To(y) \leftarrow Image(From(x), f(w,x))$;
4. $To(x) \leftarrow To(y)|_{y \leftarrow x}$;
5. $New(x) \leftarrow To(x) - Reached(x)$;
6. **if** ($New(x) = \emptyset$) **return** $Reached(x)$;
7. $Reached(x) \leftarrow Reached(x) + To(x)$;
8. $From(x) \leftarrow BestBdd(New(x), Reached(x))$;
9. }
}

図 2: symbolic traversal の基本アルゴリズム

像 $Image$ は以下の関数処理により得ることができる:

$$Image(From, f) = \exists w \exists x [From \cdot \prod_{i=1}^{n} (y_i \equiv f_i)].$$

関数 $T(w,x,y) = \prod_{i=1}^{n} (y_i \equiv f_i(w,x))$ は**遷移関係関数** (transition relation) と呼ばれる。T を表す BDD のサイズは通常極めて大きくなるため、実際には**分割遷移関係関数** (partitioned transition relation) $T_i (i = 1, \ldots, r)$ を用いることが多い。ただし $T = \prod_{i=1}^{r} T_i$である。分割遷移関係関数に基づく像計算は i が 1 から r まで以下の演算を繰り返すことにより行う:

$$P \leftarrow \exists s (P \cdot T_i). \quad (1)$$

ここで s は P と T_iの論理積を計算した後 smoothing 演算により消去できる変数からなるベクトルを表す。通常 $T_{i+1}, \cdots, T_r$のどの分割遷移関係のサポートにも含まれない外部入力変数と現状態変数が s に含まれる。上式において関数 P は**部分積** (partial product) と呼ばれる。

図 2 に symbolic traversal の基本アルゴリズムを示す。ここで、S_0は初期状態集合、$BestBdd(f,g)$ は $f \subseteq h \subseteq g$ を満足するできるだけ小さな BDD で表される関数 h を返す関数である。図 2 の 3 行目にあるように、関数 $Image(From, f)$ により計算される関数が次状態変数の関数であることに注意されたい。

2.4 BDD の動的変数順序づけ

Rudell は、BDD の生成過程で適宜変数の再順序づけを行いながら目的の BDD を構成する**動的変数順序づけ**という手法を考案した [7]。BDD の変数順序は BDD サイズに大きな影響を与えるため、BDD の動的変数順序づけは効率的な symbolic traversal には必要不可欠な技術である。Rudell の提案した動的変数順序づけでは、BDD パッケージが変数の再順序づけを自動的に、すなわちアプリケーションとは独立に行うため、アプリケーション側は動的変数順序づけを意識せずに BDD の演算を行うことができる。

動的変数順序づけのための変数再順序づけアルゴリズムとして **sifting アルゴリズム**が効率的であることが実験的に示されている [7]。sifting アルゴリズムでは、変数

を実際に上下に移動させ、最もBDDサイズが小さくなるレベルに戻す操作を繰り返し行うことにより、再順序づけを行う。具体的な手続きは以下のとおりである。

[sifting アルゴリズム]

BDDの各変数x_iに対して順に以下の(1)、(2)を行う。

(1) 変数x_iをBDDの一番上(下)のレベルから下(上)のレベルまで隣接変数のスワップ操作を繰り返すことにより実際に移動する。その際、各レベルでのBDDの全ノード数を記憶しておく。
(2) BDDのノード数が最も少なかったレベルを選び、そのレベルに変数x_iを戻す。 □

上記のアルゴリズムで変数を予めBDDのノード数の多い順などでソートしておいてもよい。

siftingアルゴリズムに基づく動的変数順序づけはBDDのノード数の爆発を抑制する方法としては大変有効であるが、変数を上下に移動させる操作を繰り返し行うため、大規模なBDDの再順序づけを行う場合には多くの計算時間を要するという欠点があった。計算時間削減のための手法としていくつかの方法が提案されている[11, 12, 8]。これらの手法はいずれも計算時間削減のためにsiftingによる探索空間に制約を加えるものである。従って結果として得られるBDDサイズは元のsiftingアルゴリズムを用いた場合と比べ大きくなる可能性がある。[11]では、あらかじめBDDの構造解析によりBDDのレベルをwindowと呼ばれるブロックにわけ、変数移動をそれぞれが属するブロックに限定している。[12]では、BDDの一部分をサンプルとして取り出し、サンプルに対してsiftingアルゴリズムを適用した結果を元のBDDの変数順序に反映させることにより、siftingに要する時間を削減している。[8]ではsymbolic traversalに特化した動的変数順序づけの高速化手法として、状態探索の計算の特性を考慮していくつかの変数のみ再順序づけする方法が提案されている。我々のアプローチも[8]と同様、状態探索のための効率化手法である。しかし、本稿では状態探索での動的変数順序づけに関する別の問題を扱う。本稿では、状態探索での状態探索において現状態変数と次状態変数をいかに扱うかという問題を考える。提案手法は上述の他の手法とは独立な問題を扱っているため、組み合わせて適用することが可能である。

2.5 動的変数順序づけにおける状態対の扱い

順序回路のsymbolic traversalプログラムの多くは、状態対の変数をグループ化し、状態探索の最初から最後まで常に隣接したレベルに状態対の変数を順序づけする。本稿ではこの方法を**状態対グループ化法**と呼ぶ。siftingアルゴリズムにおいて個々の変数の代わりに変数のグループを扱うために**group sifting**を用いる。group siftingは変数のグループを一まとまりとして変数移動を行うものである。

状態対の変数を扱うもう一つの従来法として、状態対を全く考慮せずsiftingアルゴリズムにおいてすべての変数を独立に自由に移動させる手法がある。本稿ではこの方法を**状態対非グループ化法**と呼ぶ。

状態対グループ化法の利点は現状態変数同士の相対的順序と次状態変数同士の相対的順序が常に同じであることである。これは像の状態集合を表すBDDのサイズが像計算後の変数置き換え(図2の4行目)の前後で変わらないことを意味する。しかし、状態対グループ化法は状態対の変数の位置に制限が加えられるため、像計算途中での動的変数順序づけのBDDサイズの削減能力が制限される。状態対をグループ化した場合の変数順序のBDDサイズがBDDの最少サイズより劇的に大きくなる関数は数多く存在するため、状態対グループ化法は像計算の途中でのBDDサイズの爆発や多くの動的変数順序づけの実行を引き起こす可能性がある。一方、状態対非グループ化法は像計算後の変数の置き換え時にBDDサイズの爆発や動的変数順序づけの実行を引き起こす可能性がある。像計算後の動的変数順序づけの実行は像計算途中の動的変数順序づけにより得られた像計算のための良い変数順序を破棄することになる。さらに、状態対をグループ化しない場合には状態対をグループ化する場合と比較して変数の置き換え自体にもより多くの計算時間を必要とする。

2.6 Group Sifting

本稿では[13]に基づくgroup siftingの手法を用いる。この手法では**hard group**と**soft group**という2種類のグループが存在する。hard groupは動的変数順序づけルーチンに対して外部から指定されるグループであり、動的変数順序づけの最初から最後までその変数はグループ化されたままとなる。hard groupはネストすることもできる。一方、soft groupは動的変数順序づけルーチンの中で一時的に行われるグループ化である。動的変数順序づけの途中でグループ化され、順序づけが終るとグループ化が解除される。siftingでの変数の移動時に隣接する変数とsoft groupを作るべきかどうかのチェックを行い、必要に応じてグループ化する。

元来、group siftingにおけるグループ化は隣接して順序づけすべき変数をあらかじめ見つけ出し、siftingの際に同時に移動させることが目的である。soft groupにするべき変数の選択方法として、対称変数やそれを拡張したものをグループ化する方法[14]やBDDの各レベルのノード数に関する2次差分の情報をもとにグループ化する方法[13]が提案されている。これらの方法はで、アプリケーションと独立にBDDのみの解析でsoft groupを求める。次章で我々はsymbolic traversalのための新しいグループ化法としてsymbolic traversalからの情報をもとにグループ化する変数を決定する方法を導入する。

3 Lazy Group Sifting

本章では、symbolic traversalでの動的変数順序づけの方法としてlazy group siftingを提案する。lazy group siftingは、状態対グループ化法と状態対非グループ化法を組み合わせ、状態対ごとにグループ化すべきかどうかを動的に判断し適宜グループ化を行う。以下、全体としての枠組およびどの状態対をグループ化するのかという判断基準について述べる。

状態対グループ化法は像計算後の変数置き換え処理をの計算量を軽減するため状態対の変数を常に隣接させておく。一方、状態対非グループ化法では全ての変数を自由に再順序づけする。従って、状態対非グループ化法は像計算の途中ではよりよい変数順序を生成しうる。しかし、現状態変数の相対的変数順序が次状態変数のそれと全く異なる可能性があり、像計算後の変数の置き換え処理での計算量が増大しがちである。すなわち、像計算途中での BDD サイズと像計算後の変数置き換えでの BDD サイズにはトレードオフがある。lazy group sifting はそのトレードオフをうまくとるための手法であり、以下の 2 つの手法からなる。

- symbolic traversal の計算状態に応じて各状態対についてグループ化すべきかどうかを決定する。
- 状態対の変数をグループ化しない場合でも現状態変数と対応する次状態変数の位置を考慮しできるだけ BDD サイズを増大させない限りできるだけ近くに再配置する。

以下、前者について 3.1 節で、後者について 3.2 節でそれぞれ説明する。

3.1 グループ化すべき状態対の選択

本節では、どの状態対をグループ化するべきかについて述べる。提案手法では、状態対のグループとして**静的グループ**と**動的グループ**の 2 種類が存在する。さらに、symbolic traversal を通じて常に非グループ化する状態対の選択も行う。このような非グループ化を本稿では**静的非グループ化**と呼ぶ。静的グループ化・非グループ化は状態探索を行う前に行い、動的グループ化は動的変数順序づけの各実行時に行う。

3.1.1 静的グループ化

静的グループ化は順序回路の次状態関数を計算した時点でその情報を元に行う。提案手法では以下のいずれかを満たす状態対 (x_i, y_i) を静的グループとする:

- 対応する FF がλ-FF [4] である。すなわち、どの次状態関数も現状態変数 x_i に依存しない。
- 次状態関数 f_i が現状態変数 x_i に依存していて、かつそれ以外の現状態変数には依存しない。

前者の場合、現状態変数 x_i は像計算の前に smoothing 演算により消去できるため x_i と y_i をグループ化しても BDD のサイズは増加しない。後者の場合、x_i と y_i は像計算においても強い依存関係があると考えられるため、常にグループ化しても問題はない。

静的グループは group sifting の hard group により実現する。静的グループは状態探索中常にグループ化されたままで、動的変数順序づけの際 hard group として sifting ルーチンに渡される。

3.1.2 静的非グループ化

順序回路の構造によっては次状態関数がそれと対応する現状態変数に依存しない場合がある。このような場合には、像計算ではその現状態変数とそれと対応する次状態変数には全く関係がない。本稿ではそのような状態対を**独立状態対**と呼ぶ。例えば、パイプライン型の回路は多くの独立状態対を持つ。像計算の間は独立状態対の変数を隣接して順序づけする必要はない。一般に、像計算の方が変数の置き換えより多くの計算量が必要なので、独立状態対の変数は状態探索全体を通して非グループ化しておく方がよいと考える。提案手法では、状態探索を行う前に独立状態対を見つけ出し、静的非グループ化しておく。非グループ化された状態対であっても、状態対の変数同士の距離を考慮してできるだけ近くの位置に置かれる。これについては 3.2 節で説明する。

3.1.3 動的グループ化

動的グループ化は soft group[13] と全く同一であり、動的変数順序づけルーチンの中で作られる。提案手法は以下の基準により現状態変数とそれと対応する次状態変数からなる動的グループを作る。

sifting アルゴリズムで隣接する 2 変数について以下の 2 条件を満たすかどうかを調べる:

- それらの変数が状態対変数である。すなわち、現状態変数とそれに対応する次状態変数である。かつ、
- それらがグループ化すべき状態である。

sifting アルゴリズムにおいてある変数を移動している際に動的グループが作られると、以降はもとの変数単独で移動するのではなくグループ化した状態対を一まとめとして移動を行う。動的グループは現在行われている動的変数順序づけの終了時にグループ化を解除する。

次の問題は、この枠組の中でどの状態対をグループ化するかということである。状態対の変数をグループ化すると動的変数順序づけでの BDD サイズの削減能力が制限される。しかし、動的変数順序づけが起こった時点で、2.3 節式 1 での部分積 P が依存しない変数が存在する。これらの変数は、まだ部分積 P と積をとっていない分割遷移関係のみに依存する変数や、既に smoothing 演算により部分積から消去された変数などである。部分積 P は像計算において唯一関数として変化するものであり、かつ多くの場合 BDD のサイズとして最も大きい関数である。従って、P が依存していない状態変数は sifting アルゴリズムで単独に移動させても部分積のサイズに変化はなく、その変数と対応する状態変数とグループ化しても動的変数順序づけでの BDD サイズの削減能力の制限の度合が少ないと考えることができる。この点を考慮して、上述の提案手法の概略アルゴリズムは以下のように詳細化される。

[lazy group sifting アルゴリズム]

BDD の変数 $x_i (i = 1, \ldots, n)$ に対して順に以下の (0) から (2) を行う。

(0) 現在の BDD の全ノード数 N を記憶する。

(1') 変数 x_i を BDD の一番上 (下) のレベルから下 (上) のレベルまで BDD の隣接変数のスワップ操作を繰り返すことにより実際に移動する。その際、各レベル

において、BDDの全ノード数を記憶し、かつ、以下の**条件を満足すれば変数 x_i と隣接する変数 v をグループ化し、それ以降の移動はグループとして移動する:**

- **x_i と v が状態対変数である、かつ、**
- **BDDの現ノード数が N 以下である、かつ、**
- **部分積 P が x_i と v の少なくとも一方には非依存である。**

(2) BDDの全ノード数が最少のレベルを選び、そのレベルに変数 x_i(または変数グループ)を戻す。 □

上のアルゴリズムで太字の部分が基本的なsiftingアルゴリズムと異なる部分である。上のアルゴリズムのステップ(1')の2つ目の条件はグループ化を制約するための条件であり、グループ化する際のノード数が大きすぎる場合にはグループ化は不向きだと考えグループ化は行わないためのものである。

3.2 状態対の距離の考慮

状態対がグループ化されない場合でも、像計算後の変数置き換えのことを考慮するとBDDの全ノード数が増加しないのであれば状態対の変数をできるだけ近づけて置く方がよい。これは前節のアルゴリズムのステップ2を以下のステップ2'に置き変えることにより行うことができる:

(2') BDDのノード数が最も少なかったレベルを選び、そのレベルに変数 x_i(または変数グループ)を戻す。**ノード数が最少のレベルが複数あり、変数が状態変数で、かつ変数のグループ化が起こらなかった場合には、最少のレベルのうち対応する状態変数に最も近いレベルに変数 x_i を戻す。** □

像計算後の変数代入の際に依然BDDサイズが爆発する場合には、ステップ2'のかわりに以下のステップ2"を用いることもできる。

(2") BDDの全ノード数が $\phi(minimum, \epsilon)$ 以下のレベルのうち、状態対の距離が最短になるレベルを選び、そのレベルに変数 x_i を戻す。

ここで、ϵ は外部から与えられる数である。関数 ϕ は $minimum + \epsilon$、$minimum \times (1 + \epsilon)$ などが考えられる.

4 実験結果

前章で述べたlazy group siftingの手法をCUDDパッケージ[16]と論理検証システムVIS[15]上に実現し、400MHzのPentium IIマシン(メモリ1GB)上で到達可能状態数え上げの実験を行った。実験はISCAS'89およびISCAS'89-addendumベンチマークなどの順序回路を用いて行った。到達可能状態数え上げにおいて、初期状態として全てのFFの値が0の状態を用いた。本実験ではlazy group siftingのための変更以外はVISやCUDDの設定は初期設定のままである。BDDの初期変数順序として、分割遷移関係のBDDが小さくなるような変数順序をあらかじめ求めておき、それを用いた。複数のsiftingアルゴリズムでの遷移関係の分割の仕方を統一するため、動的変数順序づけは分割遷移関係を計算した後に実行可能にした。動的変数順序づけの起動のタイミングは、CUDDのデフォルトの設定のとおりで、基本的には、BDDのノード数が直前の動的変数順序づけの結果のノード数の2倍を越えると起動される。

提案手法("lazy")と従来手法である状態対グループ化法("grp")および状態対非グループ化法("ungrp")を到達可能状態数え上げの計算により比較した実験結果を表1に示す。手法"lazy"は3節のアルゴリズムで $\phi = minumum$ としたとものである。表において、欄"回路"は回路の名前を示す。"s5378opt"はs5378を順序回路の冗長性除去により最適化したものである。"bpb"は2ステージの分岐予測バッファ回路である。"cps1364"はlanding gear controller回路である。"sfeistel"は暗号化回路である。"soap"は[17]のdistributed mutual exclusionアルゴリズムを実装した回路である。"s3278-8"はs3271の到達可能状態の探索が部分的であることを示し、深さ8までしか探索を行わなかったことを表す。欄"FF"は回路のフリップフロップ数を示す。欄"最大メモリ"は到達可能状態の探索で用いられた最大メモリ量をメガバイド単位で表したものの比較である。欄"最大ノード数"はBDDの最大使用ノード数の比較である。欄"計算時間"は動的変数順序づけに要した時間も含め到達可能状態計算全体に要した時間を秒で表したものの比較である。太字の数は比較した3つの手法のうち一番良い結果であったことを表す。一番下の行"平均"は、それぞれの項目で一番良い結果が1であるとしたときの値の平均値を示す。例えば、s1269で手法"lazy"を用いた時の最大メモリは287であり、それを一番良い結果262("grp"の場合)で割ると1.10という値が得られる。そのような値を各欄で平均をとったものが最下行"平均"の値である。

実験結果より全体的に提案手法が有効であることが分かる。最大ノード数に関しては、"lazy"は"s1269"を除いた全ての例で一番良い結果を得ている。計算時間に関しても、"lazy"はほとんどの例で一番良い結果を得ている。状態対グループ化法は"s4863"や"s5378-4"などでその他の二手法より極めて多くの計算時間を要している。これは状態対グループ化法が不得手な例題が存在することを意味する。一方、"s3271"や"cps1364"では、状態対非グループ化法は他の方法に比べて極端に結果が悪い。これらの結果より、従来の2つの方法にはそれぞれ不得手な例題が存在することを示している。それに対し、提案手法は従来手法ほど不得手な例題が存在せず、よりロバストであるといえる。最大メモリ量の点では提案手法がノード数や計算時間ほど有利でないが、その理由の一つとしてCUDDパッケージのメモリ割り当てが必要なメモリ量だけを割り当てるのでなく、割り当て可能ならできるだけ多く割り当てるというヒューリスティックに基づいていることが考えられる。

表2は到達可能状態探索における動的変数順序づけに関する情報を示したものである。表において、欄"動的順序づけに要した計算時間"は動的変数順序づけに要した時間の総和を秒で示したものの比較である。括弧内は像計算後の変数の置き換えの途中に起動された動的変数順

表 1: 到達可能状態数え上げの実験結果

回路	FF	最大メモリ (MB)			最大ノード数			計算時間 (秒)		
		lazy	grp	ungrp	lazy	grp	ungrp	lazy	grp	ungrp
s1269	37	287	**262**	363	3,216,445	2,644,876	**2,644,210**	**2,463**	4,432	2,849
s1423-10	74	133	**102**	253	**1,586,112**	1,980,587	2,214,442	**3,904**	5,073	7,023
s1512	57	**64**	68	65	**112,597**	147,632	159,372	**753**	1,142	1,179
s3271-8	116	**160**	169	238	**1,668,808**	1,958,046	2,311,196	2,357	**2,096**	6,325
s3330	132	**250**	278	297	**1,611,639**	1,930,568	1,661,904	**2,056**	2,404	4,075
s4863	104	74	**65**	143	**385,161**	591,278	400,872	**529**	12,000	1,137
s5378opt	121	153	**90**	161	**354,165**	475,777	593,286	**1,413**	3,349	1,556
s5378-4	179	191	350	**135**	**878,318**	2,590,726	926,990	**3,107**	18,711	3,546
bpb	36	**15**	16	28	**36,907**	49,319	215,752	**44**	82	393
cps1364	231	**140**	176	286	**903,844**	1,082,988	1,760,932	4,064	**3,757**	9,812
sfeistel	293	**37**	**37**	**37**	**151,705**	**151,705**	**151,705**	**241**	269	870
soap	140	32	**27**	31	**132,713**	147,364	148,307	175	167	**158**
平均		1.15	1.18	1.55	1.01	1.37	1.69	1.03	3.57	2.51

表 2: 到達可能状態数え上げでの動的変数順序づけに関する情報

circuit	動的順序づけに要した時間			動的順序づけ起動回数			$\lvert To(x)\rvert/\lvert To(y)\rvert$ の最大値		
	lazy	grp	ungrp	lazy	grp	ungrp	lazy	grp	ungrp
s1269	2252(0)	4177(0)	2522(0)	12(0)	14(0)	17(0)	1.09	1	3.74
s1423-10	3813(0)	4977(0)	6893(0)	11(0)	11(0)	13(0)	1.96	1	1.21
s1512	93(0)	77(0)	119(0)	9(0)	9(0)	10(0)	1.25	1	1.18
s3271-8	2276(0)	2011(0)	6170(21)	13(0)	13(0)	23(3)	1.37	1	2.44
s3330	1584(97)	1868(469)	3464(1297)	13(3)	11(3)	17(7)	2.07	1	2.96
s4863	498(273)	837(0)	829(420)	10(4)	11(0)	13(5)	3.95	1	13.35
s5378opt	416(19)	780(140)	832(790)	12(2)	10(1)	15(10)	1.50	1	10.92
s5378-4	2916(774)	18122(0)	3382(989)	27(7)	23(0)	25(7)	–	1	–
bpb	17(3)	12(0)	123(0)	5(1)	4(0)	8(0)	1.31	1	2.14
cps1364	3849(0)	3622(0)	9536(0)	5(0)	5(0)	8(0)	1.26	1	1.25
sfeistel	258(168)	282(0)	869(297)	8(5)	8(0)	18(7)	1.86	1	1.61
soap	128(0)	133(0)	88(67)	4(0)	4(0)	4(2)	1.31	1	1.58

序づけに要した時間の総和である。欄 "動的順序づけ起動回数" は動的変数順序づけが起動された回数を示す。括弧内は像計算後の変数の置き換えの途中の起動回数を示す。欄 "$|To(x)/To(y)|$ の最大値" は、変数の置き換え後の像を表す BDD のサイズを置き換え前の像を表す BDD のサイズで割った値を示す。ただし、動的変数順序づけが起動されなかった変数代入のみ考慮している。"–" は対象となる変数置き換えがなかった、すなわち、全ての変数置き換えで動的変数順序づけが起動されたことを表す。

表 2 から、提案手法が変数代入の際の動的変数順序づけの時間だけでなく像計算の際の動的変数順序づけの時間も短縮していることがわかる。"ungrp" と比較して、変数代入時に起動された動的変数順序づけの回数が多くの例題で削減されている。変数代入時に動的変数順序づけが起こらなかった場合でも、変数代入の前後で像を表す BDD のサイズの増加が削減されている。これらの結果は、状態対非グループ化法と比べて、提案手法は像計算においても変数代入においても効率的よく BDD のサイズが削減できていることを示している。

表 3 は提案手法においてグループ化された状態対の数を示したものである。欄 "静的グループ" は静的グループの数を示す。欄 "λFF" はλ-FF の数を示す。欄 "その他" はその他の静的グループの数を示す。欄 "静的非グループ" は静的非グループの数を示す。欄 "動的グループ" は動的グループの数を示す。動的グループの数は、起動された各動的変数順序づけ毎に異なるため、それらの平均をとっている。表 3 より、状態対非グループ化法が効果的な例題 ("s4863" や "s5378") に対しては提案手法でのグループ化される状態対が少なく、状態対グループ化法が効果的な例題 ("s3271-8" や"cps1364") に対しては提案手法でのグループ化される状態対が多いことがわかる。これは提案手法のグループ化の選択が問題の性質をうまくとらえていることを示している。

表 3: lazy group sifting における状態対グループの数

回路	FF	静的グループ		静的	動的グループ
		λFF	その他	非グループ	(平均)
s1269	37	1	8	8	5.6
s1423-10	74	2	1	1	49.5
s1512	57	0	11	0	27.2
s3271-8	116	1	26	4	111.2
s3330	132	12	0	3	90.0
s4863	104	0	0	104	0
s5378opt	121	37	0	82	1.7
s5378-4	179	17	0	161	1.0
bpb	36	0	16	0	17.2
cps1364	231	0	0	7	85.7
sfeistel	293	0	0	0	253.5
soap	140	0	0	24	55.0

5 おわりに

本稿では、symbolic traversal での動的変数順序づけを効率化するため lazy group sifting という手法を提案した。提案手法は、従来法である状態対グループ化法と状態対非グループ化法を組み合わせ、状態対ごとにグループ化、非グループ化を問題の性質に合わせて柔軟に変えるものである。この方法により、像計算とその後の変数代入双方で BDD の爆発を押えることが可能となる。実験結果より本手法が従来法より有効であることが示された。

今後の課題としては、グループ化する状態対の選択のヒューリスティックをよりロバストなものにすることがあげられる。例えば、3.2 節でのステップ 2" のϵの適切な値を求めるヒューリスティックの考案などである。また、本手法をモデルチェッキングに適用し、有効性を確かめることも重要である。

参考文献

[1] C. Berthet O. Coudert and J. C. Madre. Verification of sequential machines using boolean functional vectors. In *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[2] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of IEEE/ACM International Conference on CAD-90*, pages 130–133, November 1990.

[3] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–407, June 1991.

[4] G. Cabodi, P. Camurati, L. Lavagno, E. Macii, M. Poncino S. Quer, and E. Sentovich. Enhancing fsm traversal by temporary re-encoding. In *Proceedings of IEEE International Conference on Computer Design*, pages 6–11, October 1996.

[5] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-robdds. In *Proceedings of IEEE/ACM International Conference on CAD-97*, pages 388–393, November 1997.

[6] S. W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for fsm traversal. In *Proceedings of IEEE International Conference on CAD-91*, pages 476–479, November 1991.

[7] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of IEEE/ACM International Conference on CAD-93*, pages 42–47, November 1993.

[8] G. Kamhi and L. Fix. Adaptive variable reordering for symbolic model checking. In *Proceedings of IEEE/ACM International Conference on CAD-98*, pages 359–365, November 1998.

[9] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Proceedings of the Formal Methods in Computer Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, November 1998.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[11] C. Meinel and A. Slobodova. Speeding up variable reordering of OBDDs. In *Proceedings of the International Conference on Computer Design*, pages 338–343, October 1997.

[12] A. Slobodova and C. Meinel. Sample method for minimization of OBDDs. In *Presented at IWLS98*, June 1998.

[13] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on CAD-95*, pages 74–77, November 1995.

[14] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable rerodering of decision diagrams. In *Proceedings of the International Conference on CAD-94*, pages 628–631, November 1994.

[15] R. K. Brayton et al. Vis. In *Proceedings of the Formal Methods in Computer Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 248–256, November 1996.

[16] F. Somenzi. Cudd: Cu decision diagram package, 1995. ftp://vlsi.colorado.edu/pub/.

[17] J. Desel and E. Kindler. Proving correctness of distributed algorithms using high-level Petri nets: A case study. In *Proceedings of the International Conference on Application of Concurrency to System Design*, March 1998.