計算機アーキテクチャ　98－7
設　計　自　動　化　65－7
(1993. 1. 21)

2分決定グラフを用いた推移的閉包計算アルゴリズムと形式的検証への応用

松永　裕介†　Patrick C. McGeer‡　藤田　昌宏†

†(株) 富士通研究所
‡　カリフォルニア大

〒211 川崎市中原区上小田中 1015

あらまし　二分決定グラフで表現された状態遷移関係から効率よくその推移的閉包を計算する再帰的アルゴリズムについて述べる．本手法は状態遷移グラフの中の最大経路長に影響を受けずに推移的閉包を計算することができるため，特に最大経路長の長い状態遷移に対して効率的であるといえる．実際の順序回路の状態遷移関係を用いた実験でも，従来の反復法に比べて極めて高速に推移的閉包が計算できることが示されている．

さらに，この推移的閉包計算を用いた形式的検証アルゴリズムによれば，従来の手法が不得意としていたカウンタ等の検証が効率的に行える．

和文キーワード　2分決定グラフ，推移的閉包，形式的検証

An efficient computation of transitive closure using binary decision diagram and its application to formal verification

Yusuke Matsunaga†, Patrick C. McGeer‡, and Masahiro Fujita†

† Fujitsu Laboratories LTD.
‡ University of California, Berkeley

1015 Kamikodanaka, Nakahara-Ku, Kasawaki 211

Abstract　This paper presents an efficient algorithm to calculate transitive closure relation from given transition relation represented in BDD. Unlike other methods, this method is not affected by the longest path of a transition graph, so that this method is more efficient especially for a transition relation which has long transition path. Experiments using real sequential circuits show the efficiency compared with conventional iterative methods.

Furthermore, this transitive closure calculation is applicable to formal verification algorithm, such as sequential equivalence checking or CTL model checking. And it is more suitable for verification of a class of sequential machines which has long transition path.

英文 key words　binary decision diagram, transitive closure, formal verification

1 はじめに

設計した回路が仕様を満たしているかどうか確かめたり，修正後の回路が元の回路と等価であるかを確かめるための検証手法としては，古くからさまざまなシミュレーションを用いた方法が用いられてきたが，これとは別に回路や仕様を数学的にモデル化して検証を形式的に行う試みもいくつかなされている．特に，順序回路（有限状態機械）の形式的検証手法に関しては，BDD（Binary Decision Diagram: 二分決定グラフ）を用いて状態集合を間接的に扱う手法によってその適用可能な規模が飛躍的に拡大されたこともあって，最近，活発に研究が行なわれるようになってきている．

これらの形式的な検証手法の技術的に重要なポイントの一つは，与えられた順序機械に対して，その初期状態から到達可能な状態を求める点にある．

例えば，2つの順序機械の等価性検証は，2つの順序機械の対応する入力どうしをつなぎ，対応する出力の値が異なる時にのみ1となるような信号線をもった順序機械を仮想的に作り出し，その順序機械の到達可能な状態のなかで，その信号線を1とするようなものがないかを調べることで行なわれる．

また，CTL(computational tree logic) を用いたモデルチェッキングと呼ばれる検証手法の場合は少し複雑になるが，本質的には到達可能な状態を求める処理に帰着できる．例えば，CTL式の一例である「EFp」は「その状態からある経路に沿って，pという式が成り立っている状態に遷移可能」であるような状態集合を表しているので，この式を評価するためには，まずpが成り立っている状態集合を求め，この状態集合へ到達可能な状態の集合を求めれば良い．この場合，与えられた状態へ到達可能な状態の集合を求めるわけで，通常の与えられた状態から到達可能な状態の集合を求める処理とは向きが逆であるが，本質的には同じものであるといえる．

到達可能な状態を求める処理もそれ自体，容易なものではないが，BDDによる間接列挙法を用いた効率の良いものが提案されている．このような処理を単純に状態遷移グラフに対する深さ優先探索や幅優先探索で行おうとすると$O(|E|)$の手間がかかる．ここでEは状態遷移を表す枝の集合である．状態遷移のオーダーは最悪の場合，全状態数の2乗に比例する．さらに，状態数は順序回路のフリップフロップの数の指数乗に比例するため，小規模な順序回路でさえその状態数が数十～数百万を越えてしまうことがあり，このような単純な方法では現実的に扱うことは難しい．

これに対し，O.Coudert らはBDDを用いて状態集合を表すことによって，到達可能な状態の列挙を幅優先探索で行う手法を提案した[1]．基本的な考えは，状態集合をその特徴関数(characteristic function)として扱い，さらにその特徴関数をBDDを用いて表現するというもので，一つ一つの状態を個別に扱わずに集合として扱うことで，効率的な幅優先探索を行える点が大きなポイントとなっている．このような間接列挙法（implicit enumeration）によって，実際に扱う状態数にかかわりなく，その状態の集合を表すBDDのサイズが扱える範囲内であれば，到達可能な状態の列挙を行うことが可能となり，順序機械の等価性検証の適用可能な範囲が大幅に拡がった．

このCoudertらのアルゴリズムは元々は2つの有限状態機械の等価性を調べるものであったが，J.R.Burch らはこの手法を一般化し，CTLのモデルチェッキングによる形式的検証アルゴリズムとしての定式化を行っている．[2]

しかし，このようなBDDを用いた手法も万能というわけではない．明らかに適用不可能な場合の一つは，与えられた状態集合を表すBDD（もしくは状態遷移関係を表すBDD）が大きくなり過ぎて扱えない場合である．この場合は本質的に困難な問題であり対応が難しいが，そうでない場合もある．

例えば，nビットの2進カウンタの場合，この回路の状態遷移は各々の状態がただ一つの次状態への遷移を持つ形になっている（簡単のためリセットなどの機能は考えない）．結果として，状態遷移図は長さ2^nの大きな輪になる．この場合，初期状態からはじめてiステップ後に到達可能な状態は高々1つしかないので，そのような状態集合を表すBDDは容易に構築できる．しかし，すべての到達可能な状態を列挙するためには2^nステップを繰り返さなければならず，この場合の手間はBDDによる間接列挙を行わない従来手法と同程度のものとなってしまう．例えば，1ステップの処理に100nsかかるとして，32ビットカウンタの検証には2^{32}ステップを実行する必要があり，数日を要してしまう．

T.Filkornは直接的には到達可能な状態の計算を行わずに，外部からその出力値をみることで識別可能な状態対を求めることによって，2つの順序機械の等価性検証を行うアルゴリズムを提案している[3]．例えば，内部状態がnビットの出力ピンによって観測可能なnビット2進カウンタに対してはこのアルゴリズムは非常に効率的にふるまう．しかし，このアルゴリズムも外部から識別可能な状態対を初期状態として，そこへ到達可能な状態の集合を求めているだけなので，本質的には他のアルゴリズムと同様の計算を行っているに過ぎない．内部状態が隠されており，2^nクロックに1回だけ出力が1になるような2進カウンタの場合にはこのアルゴリズムでも他のアルゴリズムと同様に2^nステップを要する．

このように，従来の多くのアルゴリズムはBDDを用いて複数の状態を一度に扱うことによって処理の効率化をはかっているが，状態遷移図の最大経路長が極端に長い順序機械に対してはBDDの長所をうまく活かすことができない．そこで，本稿ではこのように従来手法があまり得意としていない場合を効率的に扱えるような検証手法として推移的閉包を用いた検証手法を提案し，また，効率的な推移

的閉包の計算アルゴリズムを提案する．

2 状態遷移関係と推移的閉包

2.1 用語の定義

定義 1 有限状態機械 $\mathcal{M}$ は 6 つ組 $(\Sigma, S, O, \lambda, \delta, s_0)$ で与えられる．ここで，Σは入力記号の集合，Sは内部状態の集合，Oは出力記号の集合，λは $S \times \Sigma$から Oへの写像，つまり出力値を決める関数，δは $S \times \Sigma$から Sへの写像，つまり状態遷移を与える関数，$s_0 \in S$は初期状態である．

定義 2 有限状態機械 $\mathcal{M}$ に対して状態遷移関係$\mathcal{R}$を以下の様に定義する．

状態 $s_1, s_2 \in S$に対して，関係 $s_1 \mathcal{R} s_2$ が成り立つのは，ある入力記号$\sigma \in \Sigma$に対して s_1から s_2 への状態遷移がある場合であり，かつその時に限る．

状態遷移関係はその特徴関数を用いて表すこともできる．即ち，状態遷移関係は次のような関数 $R(s,t) : S \times S \rightarrow \{0,1\}$,

$$R(s,t) = \begin{cases} 1 & \exists \sigma \in \Sigma : \delta(s,\sigma) = t \\ 0 & \text{それ以外} \end{cases}$$

と表される．

定義 3 状態遷移関係$\mathcal{R}$に対して，推移的閉包$\mathcal{R}^+$を次のように定義する．$s\mathcal{R}^+ t$が成り立つのは，$m \geq 2, s = u_1, t = u_m$ に対して列 $u_1 \mathcal{R} u_2, u_2 \mathcal{R} u_3, \ldots, u_{m-1} \mathcal{R} u_m$ が存在する時であり，かつ，その時に限る．

2.2 推移的閉包を用いた検証手法

Coudert のアルゴリズム [1] と Filkorn のアルゴリズム [3] はそれぞれある状態から到達可能な状態の集合とある状態へ到達可能な状態の集合を求めているが，推移的閉包を用いて同様の処理を行うこともできる．例えば，推移的閉包関係を$\mathcal{R}^+$とすると，初期状態 s_0から到達可能な状態の集合 Uは，

$$U = \{u | u \in S, s_0 \mathcal{R}^+ u \text{ が成り立つ } \}$$

で与えられる．つまり，$\mathcal{R}^+$を表す BDD，$R^+(s,t)$ に対して $s = s_0$ と制限した結果の BDD がそのような状態の集合を表している．逆にある状態へ到達可能な状態の集合も同様の処理で求めることができる．

CTL のモデルチェッキングも同様に推移的閉包を用いて計算することが可能である．この場合，もしも同じ状態遷移関係において，いくつかの異なる初期状態から到達可能な状態の集合を求める処理が繰り返されるような場合には，推移的閉包は最初に一度，計算しておけばよく，2 回目以降は計算済みの推移的閉包に対して初期状態を適用すればよい．このような例は特殊なわけではなく，一つのモデルに対していくつかの仕様を検証するような場合には起こりうるものと思われるので，推移的閉包を用いた場合にはその特徴が活かされることになる．

このように，与えられた状態遷移関係に対してその推移的閉包が求められれば，それを順序機械のさまざまな検証手法に応用することが可能である．残る問題は，いかにして推移的閉包を効率よく計算するか，ということになる．

2.3 推移的閉包の計算アルゴリズム

次に与えられた状態遷移関係$\mathcal{R}$に対して推移的閉包$\mathcal{R}^+$を計算する従来のアルゴリズムについて述べる．

2.3.1 単純な反復法

もっとも単純なアルゴリズムは次のように i ステップ以内に到達可能な状態の関係 R_iを計算して行き，$R_{i+1} = R_i$ となった時点で繰り返しを止めるものである．ここで，R_i は，

$$R_i(s,t) = \begin{cases} R(s,t) & i = 0 \\ (\exists u R_{i-1}(s,u) \wedge R(u,t)) \\ \quad \vee R_{i-1}(s,t) & i \geq 1 \end{cases}$$

で与えられる．つまりある状態 u があって，s から $i-1$ ステップで u に到達可能であって，かつ，u から t へ 1 ステップで到達可能であれば，s から t まで i ステップで到達可能である．また，s から t へ $i-1$ ステップ以内で到達可能でなく，このような u がなければ i ステップで s から t まで到達することはできない．

明らかなように，このアルゴリズムでは状態遷移図の中の最大経路長の数だけ上記のステップを実行することになり効率的でない．

2.3.2 反復二乗法 (iterative squaring)

これに対し，iterative squaring と呼ばれる手法が提案されている．この手法も同様に繰り返し R_iを計算して行き，$R_{i+1} = R_i$となったところで繰り返しを止めるものであるが，Ri の計算方法が次のように異なっている．

$$R_i(s,t) = \begin{cases} R(s,t) & i = 0 \\ (\exists u R_{i-1}(s,u) \wedge R_{i-1}(u,t)) \\ \quad \vee R_{i-1}(s,t) & i \geq 1 \end{cases}$$

つまり，R_0は 1 ステップで到達可能な状態の関係，R_1は 2 ステップ，そして R_2では 4 ステップ以内で到達可能な状態の関係を表すことになる．一般に R_iでは 2^i ステップ以内で到達可能な状態の関係を表すことになるので，最大経路長 L に対してたかだか $\log L$ 回の反復で上記のステップは収束することになる．

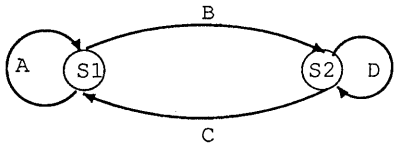

図 1: 状態遷移の分割

ところが，R_iを表す特徴関数を BDD で表現した場合には，このような処理の途中で得られる R_iの BDD サイズは比較的大きくなる傾向があるため，たとえステップ数が対数的に少なくなっても計算時間がかえって増大してしまうことがある．このため，この方法は BDD を用いた実装ではあまり効果的ではない．この点については実験結果の章でもう一度触れる．

2.3.3 再帰的な方法

もう一つのアルゴリズムは隣接行列の形で表された状態遷移関係から再帰的に推移的閉包を表す行列を構築するものである．

以下に示すアルゴリズムは基本的には A.V.Aho, J.E.Hopcroft, J.D.Ullman の古典的なアルゴリズム [4] に基づいたものであるが，もとの文献では反射的な推移的閉包（$\mathcal{R}^*$と表す，常に$s\mathcal{R}^*s$が成り立つ）を求めているので本稿では，前述の定義に従った推移的閉包（$\mathcal{R}^+$と表す，sを含む閉路がなければ$s\mathcal{R}^+s$は成り立たない）を求めるアルゴリズムに修正している．

Xを$n\times n$行列とする．ここでnが2のベキ乗，すなわち2^kであると仮定する．まず，Xを4つの$2^{k-1}\times 2^{k-1}$行列に分割する．

$$X=\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

この分割で，もととなる状態は2^{k-1}の集合S_1, S_2に分けられる．部分行列AはS_1に属する状態間の遷移を表し，部分行列DはS_2に属する状態間の遷移を表す．行列BはS_1の状態からS_2の状態への遷移を表し，行列CはS_2の状態からS_1の状態への遷移を表す．この様子が図 1に示されている．

求める推移的閉包を

$$X^+=\begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

とすると，EはS_1に属する状態間の，またHはS_2に属する状態間の推移的閉包関係となる．FはS_1を始点として，S_2を終点とする推移的閉包関係を表し，GはS_2を始点としてS_1を終点とする推移的閉包関係を表している．

まず，Eを計算するものとする．状態uから状態vへの長さ1以上の道で，u,vともS_1に属するものは以下の3通りが考えられる．

(1) 行列Aに含まれる遷移

(2) 行列Bに含まれる遷移からはじまり，行列Dに含まれる遷移を0回以上の繰り返した後で，行列Cに含まれる遷移で終わる道

(3) 上記(1)又は(2)の1回以上の繰り返しからなる道

このような遷移を表すと

$$E=(A+B\cdot D^*\cdot C)^+$$

となる．$D^*=D^++\mathcal{I}$ ($\mathcal{I}$は単位行列）であるので，

$$E=(A+B\cdot(D^++\mathcal{I})\cdot C)^+$$

と書き表せる．同様にHも

$$H=(D+C\cdot(A^++\mathcal{I})\cdot B)^+$$

と書き表せるが，Eの計算に用いた部分式を再利用した方が効率的なので，次のような式を用いる．

$$H=D^++(D^++\mathcal{I})\cdot C\cdot(E+\mathcal{I})\cdot B\cdot(D^++\mathcal{I})$$

この場合，新たに閉包演算を必要としない．

F,Gも

$$F=(E+\mathcal{I})\cdot B\cdot(D^++\mathcal{I})$$

$$G=(D^++\mathcal{I})\cdot C\cdot(E+\mathcal{I})$$

と書き表せる．

結局，

$$Tmp1=D^+$$

$$Tmp2=B\cdot(Tmp1+\mathcal{I})=B\cdot Tmp1+B$$

$$Tmp3=(Tmp1+\mathcal{I})\cdot C=Tmp1\cdot C+C$$

を予め計算しておけば，

$$E=(A+Tmp2\cdot C)^+$$

$$F=(E+\mathcal{I})\cdot Tmp2=E\cdot Tmp2+Tmp2$$

$$G=Tmp3\cdot(E+\mathcal{I})=Tmp3\cdot E+Tmp3$$

$$H=Tmp1+Tmp3\cdot F$$

という一連の手順でXの推移的閉包を計算することができる．この場合，$2^k\times 2^k$行列の閉包演算は，$2^{k-1}\times 2^{k-1}$行列の閉包演算2回と6回の和演算および6回の積演算で実行することができる．

これにより，行列の推移的閉包を求める手間はその行列と同じサイズの行列どうしの積の演算の手間と同程度（$O(n^3)$: nは行列の行／列の数）で実行可能なことが示されている．

3 BDDを用いた推移的閉包の計算アルゴリズム

ここでは，BDDを用いて状態遷移関係を表現した場合に適用可能な推移的閉包の計算アルゴリズムを提案する．このアルゴリズムは前述の再帰的アルゴリズムに基づくものである．

3.1 BDDによる行列表現

各要素が0又は1であるような$n \times n$ブール行列は見方を変えると定義域$\{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$から値域$\{0,1\}$への写像とみなすことができる．そこで，$n = 2^k$とすると，$2k$入力1出力の論理関数で行列を表現することができる．

例えば，図2で示される有効グラフの隣接行列Aは

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

であるが，これを論理関数として表現することを考える．まず，節点v_1, v_2, v_3, v_4を表1のように符号化する．ここで，行に対する変数a, bと列に対する変数c, dの合わせて4つの変数が導入されている．このような符号化を行うと，図2の隣接行列は図3に示す様なカルノー図とみなすことができる．これより，図2の隣接行列を表す論理関数は，$\overline{a}\overline{b}d + abd + acd + \overline{a}bc\overline{d}$であることがわかる[1]．

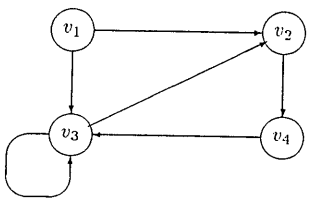

図2: 有向グラフの例

隣接行列に対する関数表現が得られたら，入力変数の間に適当な全順序関係を与えてその関数を表すBDDを作ることは容易に行える．例えば図2の例で変数順を$a \prec c \prec b \prec d$とした場合のBDDを図4に示す．

3.2 BDDによって表現された行列の演算

次に，推移的閉包を求めるための行列演算について，BDDを用いた表現でどのように対応するか述べる．扱う演算は，1) 行列の分割，2) 行列の和，3) 行列の積　の3つがある．

[1] このような関数表現は唯一ではなく，節点をどのように符号化するかによって異なったものとなる．

表1: 節点の符号化

節点	a	b	c	d
v_1	0	0	0	0
v_2	0	1	0	1
v_3	1	1	1	1
v_4	1	0	1	0

		cd			
		00	01	11	10
ab	00	0	1	1	0
	01	0	0	0	1
	11	0	1	1	0
	10	0	0	1	0

図3: カルノー図

3.2.1 行列の分割

まず，最初に必要となるのが行列を4つに分割する処理であるが，これはBDDの変数順を工夫することで簡単に行える．図2の例で節点を$\{v1, v2\}$と$\{v3, v4\}$に分けることによって隣接行列を分割することを考える．この場合，表1に示す符号化を行ったとすると，変数aおよびcでcofactorをとる操作（a, cの値を0又は1に固定する操作）が隣接行列の分割に相当する．例えば，$\{v1, v2\}$内のみの隣接行列を求めるためには$a = 0$および$c = 0$でcofactorを取ればよい．結果の論理関数は

$$(\overline{a}\overline{b}d + abd + \overline{a}bc\overline{d})|_{a=0,c=0} = \overline{b}d$$

となる．これは$v1$から$v2$への遷移に相当している．残りの3つの部分行列も同様なcofactor演算で求めることができる．

このように，隣接行列の分割は，行，列を表す変数をそれぞれ1つづつ取ってきてcofactor演算を施すことで行える．BDDに対するcofactor演算はBDDのノード数に比例した手間で行うことができるが，もしもcofactorをとる変数が与えられたBDDのなかでもっとも根に近い変数の場合には，その根の節点の指す2つの部分グラフがそれぞれ0および1でcofactorを取った結果になっているので，この処理は定数時間で行える．そこで，分割の順に従って変数を並べておけば，すべてのcofactor演算を定数時間で行うことができる．具体的には（行の第一変数）$\prec$（列の第一変数）$\prec$（行の第二変数）$\prec$（列の第二変数）...という順序を用いればよい．図4のBDDはそのような規則に基づいて順序づけされている．例えば，このBDDを4つに分割するためにはaの節点とその子供の2つのcの節点を取

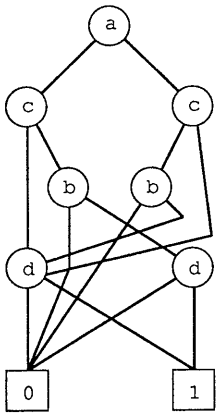

図 4: 行列を表す BDD

り除き，それらの子供の節点をそれぞれ根とする BDD を作ればよい．

与えられた 4 つの部分行列をマージして 1 つの行列にする処理も BDD 上で簡単に行える．具体的には分割の処理の逆を行えばよい．

3.2.2 行列の和

$n \times n$ 行列 $A = a_{ij}(i = 1,\ldots,n, j = 1,\ldots,n), B = b_{ij}(i = 1,\ldots,n, j = 1,\ldots,n)$ に対する和演算の結果 $C = A + B$は次のように定義される．

$$C = c_{ij}(i = 1,\ldots,n, j = 1,\ldots,n),$$
$$\text{ただし,} c_{ij} = a_{ij} + b_{ij}$$

そこで，この場合は単純に，行列 A を表す関数 f_Aと行列 Bを表す関数 f_Bの論理和を計算すればよい．つまり，2 つの行列の和を表す BDD，f_C は

$$f_C = f_A \vee f_B$$

で与えられる．2 つの BDD の論理和を求める演算は結果の BDD のサイズに比例した手間で実行することができる [5]．

3.2.3 行列の積

$n \times n$ 行列 $A = a_{ij}(i = 1,\ldots,n, j = 1,\ldots,n), B = b_{ij}(i = 1,\ldots,n, j = 1,\ldots,n)$ に対する積演算の結果 $C = A \cdot B$は次のように定義される．

$$C = c_{ij}(i = 1,\ldots,n, j = 1,\ldots,n),$$
$$\text{ただし,} c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

これは言い換えると，

$$c_{ij} = \begin{cases} 1 & \text{ある } k\text{が存在して } a_{ik} = 1 \\ & \text{かつ } b_{kj} = 1 \\ 0 & \text{それ以外} \end{cases}$$

となる．そこで，m を $n = 2^m$ を満たす整数として，行列 A を表す関数を $f_A(x_1,\ldots,x_m,y_1,\ldots,y_m)$，行列 Bを表す関数を $f_B(x_1,\ldots,x_m,y_1,\ldots,y_m)$ とすると（x_iは行に対する変数，y_jは列に対する変数），行列の積を表す関数 f_Cは，

$$f_C(x_1,\ldots,x_m,y_1,\ldots,y_m) =$$
$$\exists z_1,\ldots,z_m(f_A(x_1,\ldots,x_m,z_1,\ldots,z_m)$$
$$\wedge f_B(z_1,\ldots,z_m,y_1,\ldots,y_m))$$

で与えられる．つまり，新たな変数群 $z_1,\ldots,z_m$を導入し，変数の入れ替えを行ったあとで，2 つの関数の論理積を計算し，最後に変数群 $z_1,\ldots,z_m$を消去すればよい．存在作用素（existential quantifier）がついた変数を消去する処理は，該当する変数で 0,1 両方の cofactor を取り，その 2 つの結果の論理和を計算することで行える．

このようにして 2 つの行列の積を計算することができるが，変数の置き換え，論理積，そして変数の消去と一連の処理を別々に行わずに 1 回の処理で積を計算することも可能であり，より効率的である．そこでそのようなアルゴリズムを提案する．アルゴリズムを図 5に示す．

なお，図 5中の加算記号「+」は BDD で表現された行列に対する加算である．アルゴリズムは最初に終端条件かどうかのチェックを行う．どちらか一方が 0 の場合は結果は常に 0 になる．また，どちらも 1 の場合は結果は常に 1 となる．図 4の例でも明らかなように，部分行列が 0 行列（又はすべての要素が 1 の行列）になってしまった場合はたとえそれが終端のレベルでもなくても 0（又は 1）を表す BDD が返されるので，このチェックは再帰のレベルによらず必要である．

次に，同じ計算を過去にしていないかを調べ，計算結果がハッシュ表に格納されている場合にはその結果を返す．そうでなければ，2 つの行列を分割する．

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

とすると

$$A \cdot B = \begin{pmatrix} A_1 \cdot B_1 + A_2 \cdot B_3 & A_1 \cdot B_2 + A_2 \cdot B_4 \\ A_3 \cdot B_1 + A_4 \cdot B_3 & A_3 \cdot B_2 + A_4 \cdot B_4 \end{pmatrix}$$

で個々の部分行列を計算することができる．この $A_1 \cdot B_1$ のような分割された行列に対する積は再帰的に計算する．

最後に結果を格納し，その値を返す．

おおまかな流れは以上であるが，効率化のために次のような処理を行っている．例えば A_1と A_2が等しい場合，

$$A_1 \cdot B_1 + A_2 \cdot B_3 = A_1 \cdot B_1 + A_1 \cdot B_3 = A_1 \cdot (B_1 + B_3)$$

```
mat_mul(bdd A, bdd B) {
    if (A = 0 ∨ B = 0)
        return 0;
    if (A = 1 ∧ B = 1)
        return 1;
    if (A × Bの計算結果がハッシュ表に入っている)
        return ハッシュ表の結果;

    /* 行列の分割 */
    (A1, A2, A3, A4) ← split(A);
    (B1, B2, B3, B4) ← split(B);

    C1 = mat_mul(A1, B1) + mat_mul(A2, B3);
    C2 = mat_mul(A1, B2) + mat_mul(A2, B4);
    C3 = mat_mul(A3, B1) + mat_mul(A4, B3);
    C4 = mat_mul(A3, B2) + mat_mul(A4, B4);

    /* 部分行列のマージ */
    C ← merge(C1, C2, C3, C4);
    ハッシュ表に A × B → Cを登録;
    return C;
}
```

図 5: 行列の積のアルゴリズム

```
t_closure(bdd X) {
    if (X = 0)
        return 0;
    if (X = 1)
        return 1;
    if (Xの計算結果がハッシュ表に入っている)
        return ハッシュ表の結果;

    (A, B, C, D) ← split(X);
    t1 ← t_closure(D);
    t2 ← B · t1 + B;
    t3 ← t1 · C + C;

    E ← t_closure(A + t2 · C);
    F ← E · t2 + t2;
    G ← t3 · E + t3;
    H ← t1 + t3 · F;

    Y ← merge(E, F, G, H);
    t_closure(X) → Yをハッシュ表に登録;
    return Y;
}
```

図 6: 推移的閉包の計算アルゴリズム

と式を変形することが可能である．この場合，元の式では2回の積と1回の和の計算が必要だったのに対して，式変形の後では1回の積と1回の和の計算で同様の計算が行える．行列の積の計算は比較的高価であるのでこのような式変形は場合によっては効率化に大きく役立つ．

3.3 推移的閉包の計算アルゴリズム

3.2節で説明した演算を用いれば2.3節の推移的閉包を計算するアルゴリズムをBDDを用いたアルゴリズムに変更することは比較的容易である．特に最初の2つの反復法によるアルゴリズムはBDDの論理和，論理積，および存在作用素のついた変数の消去の演算を用いて直接，実装可能である．3つめの再帰的なアルゴリズムも行列の積，および和の演算と分割を用いて実装できるが，その際に他のBDDの再帰的アルゴリズムと同様に，演算結果を格納するハッシュ表を併用することによってアルゴリズムの効率を上げることができる．アルゴリズムを図6に示す．

この表引きの技法は，等しい部分行列は同一のBDDによって表されるというBDDの特性を利用している．もし，この表引きをおこなわないとすると，この再帰的アルゴリズムは常に2回の再帰呼び出しを行うため，再帰のレベルに対して指数的な計算の手間を持つことになる．しかし，実際には表引きのため，同じ部分行列に対する推移的閉包の計算は1度しか行われない．このため，平均的にはこの再帰的アルゴリズムが再帰のレベルに対して必ずしも指数的な計算複雑度を持つものでないことが期待される．

もとの行列を表すBDDのノード数が実際の行列の要素数に比べてとても小さいのならば（状態遷移関係の場合，実験的にそうであるといえる），それは多くの部分行列が等しいために共有されてコンパクトな表現になっているといえるので，多くの場合，この再帰的アルゴリズムがうまく動くであろうという傍証になっている．

3.4 推移的閉包に関する実験結果

上述のBDDを用いた推移的閉包の計算アルゴリズムを実装し，ISCAS'89[6]のベンチマーク回路に対して推移的閉包を求める実験を行った．プログラムはC++で記述されており，実行はDECsystem5500（約38MIPS）上で行われた．結果を表2に示す．

最初の列が回路名である．例題の回路のうち，状態遷移関係が同一のもの（例えばs344とs349）は同じ結果なので一つのみ記している．次の2つの列はそれぞれ状態遷移関係(TR)と推移的閉包(TC)を表すBDDのノード数を表している．残りの3つの列が計算時間で，それぞれ単純な反復法（linear），反復二乗法（square）および再帰法（recur）を表している．また，「>80M」は使用メモリ量が80メガバ

表 2: 推移的閉包の実験結果

回路名	BDD のノード数		CPU 時間 (秒)		
	TR	TC	linear	square	recur
s208	50	1	0.22	0.02	0.00
s298	453	688	0.63	0.80	0.17
s344	586	132639	91.40	>80M	41.70
s382	765	1143	46.00	13.35	1.90
s386	79	11	0.03	0.02	0.00
s420	116	1	103.92	0.42	0.00
s444	765	2055	39.05	122.02	2.88
s510	148	56	0.92	0.23	0.02
s526	1334	5743	106.18	>80M	4.18
s641	4112	1212	29.87	29.92	4.13
s820	94	9	0.07	0.03	0.00
s838	>80M	-	-	-	-
s953	864	579	0.38	0.87	0.22
s1196	4851	1176	2.48	2.37	4.88
s1488	177	10	0.73	0.40	0.02

イトを越えたために実行を打ち切ったことを示している.

この結果からわかるように, 再帰的なアルゴリズムの計算時間は他の反復法に比べて極めて短い. また, 反復二乗法は単純な反復法よりも遅くなることがあり, また, メモリ使用量の制限で結果が求められない例もあることがわかる. 総じて再帰的なアルゴリズムが抜群の性能を持っていると言える.

ただし, 通常の場合は推移的閉包の計算にかかる時間は, Coudert や Filkorn のアルゴリズム [1, 3] よりも長いことが多い. これは推移的閉包のほうがより一般的, かつ広範囲な情報を持つためであり, 簡単な例では推移的閉包の持つ多くの情報を用いなくても等価性検証ができてしまうためである.

しかし, 最初で述べたように非常に長い状態遷移を持つ順序機械の場合には計算複雑度の面からも実際の計算時間の面からも推移的閉包を用いた手法のほうが有利となる. 例えば, 表 2中の回路 s420 は 16 個のフリップフロップを持つカウンタであるが, これを従来の手法で到達可能な全状態を列挙するためには表 2と同じ条件で約 22 秒も必要とする. しかるに推移的閉包は 100 ミリ秒もかからずに計算可能である. もしフリップフロップの数が 32 個になればこの差はさらに顕著なものとなることが予想される.

このように, 各々のアルゴリズムにはそれぞれ得手, 不得手があるので場合に応じて効率的なアルゴリズムを使い分けることが必要であろう. そのための一つのオルタナティブとしてこの推移的閉包を用いたアルゴリズムは重要な意味を持つものと思われる.

4 おわりに

本稿では, BDD を用いて推移的閉包を計算するアルゴリズムについて述べ, ベンチマーク回路を用いた実験結果を示した. 実験結果よりここで提案している再帰的なアルゴリズムが他の手法に比べてはるかに効率的であるといえる.

ただし, いくら効率的に推移的閉包を計算しても, それよりもただ単に到達可能な状態集合を求めたほうが速い場合もあるため, 推移的閉包を用いた検証手法自体が常に効率的というわけではない. 場合に応じた使い分けが必要になるものと思われる.

また, 本稿では述べていないが, 形式的検証の手法として, ω-オートマトンの言語包含関係 (langage containmen) を用いた手法も提案されている. この手法は状態遷移図中のある状態を含む強連結成分 (strongly connected component) を計算する必要があり, 推移的閉包関係から強連結成分を求めることは容易に行えるので, 本アルゴリズムをこの手法に適用することも可能である.

参考文献

[1] O.Coudert, C.Berthet, and J.C.Madre, "Verification of sequential machines using boolean functional vectors" In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, 1989.

[2] J.R.Burch, E.M.Clarke, K.L.McMillan, and D.L.Dill, "Sequential circuit verification using symbolic model checking" In *ACM/IEEE Design Automation Conference*, 1990.

[3] T.Filkorn, "A method for symbolic verification of synchronous circuits" In *proceedings of IFIP CHDL-91*, 1991.

[4] A.V.Aho, J.E.Hopcroft, and J.D.Ullman (野崎 昭弘／野下 浩平訳), "アルゴリズムの設計と解析 I", サイエンス社, 1977.

[5] R.E.Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computer*, C-35(12), 1986.

[6] F.Brglez, D.Bryan, and K.Kozminski, "Combinational profiles of sequential benchmark circuits", *IEEE Int'l Symp. on Circuits and Systems*, 1989.