設計自動化　40－17
(1987. 12. 18)

論理関数のグラフ表現を用いた記号シミュレーション

Symbolic Simulation Using a Graphical Representation of Boolean Functions

北嶋　雅哉　　高木　直史　　矢島　脩三
Masaya KITAJIMA　Naofumi TAKAGI　Shuzo YAJIMA

京都大学工学部
Faculty of Engineering, Kyoto University

あらまし　論理回路の記号シミュレーションは、回路内の信号値を入力変数の論理関数として扱うことにより、少ない入力パタンで回路動作の理解し易いシミュレーション結果を得る有効な手法である。本稿では、信号値として論理関数のグラフ表現を用いた記号シミュレーションについて述べる。二分決定図と呼ばれるグラフ表現を採用し、さらに複数の関数でサブグラフを共有することにより使用記憶量と処理時間の削減をはかっている。また、あるグラフとその否定のグラフとをポインタで結ぶことによって論理演算に要する時間を短縮している。従来の論理式を用いる方法に対して、冗長変数の除去が容易に行え、使用記憶量と処理時間においても比較的効率の良い記号シミュレーションが行える。

Abstract　Symbolic simulation is an efficient CAD tool in the logic design. In this article,symbolic simulation using a graphical representation of boolean functions is described. A graphical representation called "Binary Decision Diagrams" is adopted. By sharing subgraphs among the different functions, the processing time and the required strage are reduced. Moreover,by coupling the graph of a function and that of its complement, the processing time for logical operations is reduced. In the simulation, simplification of the expression of logic functions such as elimination of reduntant variables can be performed efficiently.

1．はじめに

論理シミュレーションは、論理回路の動作を計算機上で模擬することによって回路の設計ミスを検出する有効な方法であり、いまや論理設計支援システムには欠くことのできないものとなっている。最近では対象回路の大規模化、複雑化にも十分対応できるようにベクトル計算機や専用ハードウェアの利用による高速化[1][2]がはかられている。一方、設計ミスの発見を効率よく行うために、マンマシン・インタフェースを向上させるというアプローチがある。論理シミュレータは設計された回路の動作を確認するための支援ツールであるため、利用者が回路の動作を容易に把握できるような機構を備えていることが望ましい。このような要求に答えるものとして記号シミュレーションが考えられている[3][4][5]。

論理回路の記号シミュレーションは、回路の入力に0，1の他に、変数（記号）を用いてシミュレーションを行うものであり、回路の出力は入力変数の論理関数（記号式）で表わされる。変数を用いることにより、入力パタンを少なくすることができ、また論理の追跡が容易であるため、回路の動作が理解し易いという利点がある。しかし、論理関数（記号式）の簡単化をいかに効率よく行うかが重要な課題となっている。本稿では、冗長変数の除去が容易であり、使用記憶量と処理時間においても比較的効率のよい論理関数のグラフ表現を用いた記号シミュレーションについて述べる。

記号シミュレーションにおいては、理解し易い

シミュレーション結果を得るために論理関数の簡単化を十分に行う必要があり、特に関数の値に影響しない変数を完全に除去することが重要である。現在、記号シミュレーションにおける論理関数の表現としては論理式が広く用いられているが、論理式の十分な簡単化を行うためには非常に多くの時間を要するため、式の値に影響しない変数の除去が完全には行われていないのが現状である。

本稿の記号シミュレーションでは、信号値の内部表現として二分決定図（Binary Decision Diagrams）[6]と呼ばれる論理関数のグラフ表現を採用している。さらに、複数の論理関数でサブグラフを共有することにより、使用記憶量を削減し、関数の等価性判定を容易にしている。また、ある関数を表わすグラフとその関数の否定を表わすグラフとを双方向のポインタで結ぶことにより、論理演算に要する時間を短縮している。以下、２章では、信号値の内部表現として論理関数のグラフ表現を用いた記号シミュレーションについて述べ、３章でその評価を行う。

２．論理関数のグラフ表現を用いた記号シミュレーション

２．１二分決定図の採用

二分決定図は図1に示すようなループのない有向グラフによる論理関数の表現方法である。サブグラフを共有することにより、比較的コンパクトな表現が可能である。グラフの葉は論理値0，1に対応し、葉でない節は変数名に対応している。節は図2に示すように３つの属性（var,low,high）を持つ。葉でない節では、x.varは変数名であり、x.low、x.highはそれぞれ0，1とラベル付けされた枝の指す節である。葉では、x.varは0または1で、x.low、x.highはともにnilである。二分決定図の表わす論理関数は、図3に示すようにして得られる。

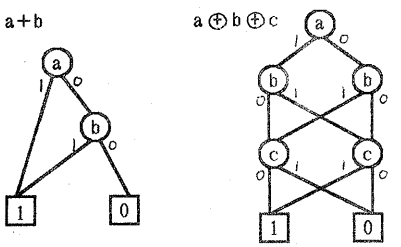


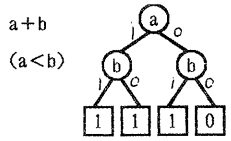
図1 二分決定図

```
type node is record
    x.var :変数名または0,1;
    x.low :node;
    x.high:node;
end record;
```

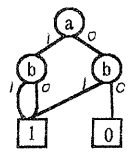
図2 節の属性

```
if x.var=0 or x.var=1 then
    f(x):=x.var;
else
    f(x):=x.var・f(x.high)+x.var・f(x.low);
end if;
```

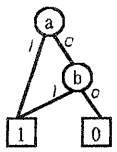
図3 二分決定図の表わす論理関数



(1) 真理値表のグラフ表現



(2) サブグラフの共有



(3) 冗長な節の削除

図4 順序付き二分決定図

ここで、すべての変数を順序付けし、グラフ中の親子の節の間に

x.var＜x.high.var ， x.var＜x.low.var

なる制約をつけると、根から葉にいたる１つのパス上には同じ変数の節は２回以上現われないことになり、基本的には真理値表を図4(1)に示すようなグラフで表現したものとなる。このような変数の順序による制約をつけた二分決定図において、図4(2)のように重複している同じグラフは１つにまとめ、図4(3)のようにx.low＝x.highであるような冗長な節xを削除する処理を行うと、関数に対応するグラフ表現は一意に決まる[7]。本稿では、信号値の内部表現としてこのグラフ表現を採用する。このグラフ表現では、関数の等価性判定がグラフの比較で行うことができる。

２．２複数の関数でのグラフの共有化

記号シミュレーションにおいては、回路内の複数の信号線に対応して、複数の論理関数を同時に表現する必要がある。本稿では、使用記憶量と処

理時間の削減のために、複数の論理関数の間においてもサブグラフの共有化を行う。複数の論理関数でのサブグラフの共有により、回路内の信号値と出力値は、複数の根を持つ一つのグラフで表現される。すなわち、ここで述べるグラフ表現は多出力関数のグラフ表現となる。

サブグラフを共有するために、「節テーブル」を用意し、すでに生成された節はすべてこのテーブルに記憶しておく。新たに節を生成するときには次に述べる処理を行い、同じサブグラフが重複して作られないようにする。

(1) 「節テーブル」を調べた結果、いま作ろうとしている節と同じ節（var,low,highがそれぞれ等しい節）が既に存在しているなら、新しい節を作らずに、既に存在している節を利用する。

(2) 作ろうとしている節と同じ節が節テーブル上に存在していなければ、新しく節を作り節テーブルに登録する。

節の生成の際に、このような処理を行うことにより、複数の関数においてもサブグラフが共有され、同じグラフが重複して生成されることはない。したがって、関数の等価性判定は節を指すポインタの比較で行うことができる。

```
function s-NOT(x:node) return node is
begin
    if x.var=0 then
        return(1);
    elsif x.var=1 then
        return(0);
    elsif x.inv<>nil then
        return(x.inv);
    else
        z.var:=x.var;
        z.high:=s-NOT(x.high);
        z.low:=s-NOT(x.low);
        z.inv:=x;
        x.inv:=z;
        return(z);
    end if;
end s-NOT;
```

図5 否定演算

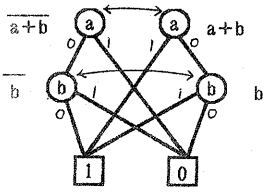


図6 否定演算の例

2.3 論理演算の方法

＜否定演算＞

ある関数fの否定を表わすグラフは、関数fを表わすグラフの0と1とを交換したグラフである。記号シミュレーションでは、fのグラフを保存したままfの否定のグラフを作る必要があるので、fのグラフの0と1とを交換したグラフを新たに作ることになる。このように否定演算では、グラフ全体にわたる処理が必要であり、またグラフを新たに作る際にはグラフ中のすべての節について節テーブルをチェックする必要がある。したがって、否定演算に要する計算時間は、節テーブルの大きさとグラフの大きさの積に比例する。本稿では、否定演算を効率良く行うために、invという節の属性を新たに追加し、元のグラフと否定のグラフとを互いにこのinvという否定のポインタでつないでおく。したがって、一度否定演算が行われると、以後そのグラフの否定が否定ポインタをたどるだけで得られる。否定演算の方法を図5に示す。この方法により、図6の例のようにすべてのサブグラフについても元のグラフとその否定のグラフが否定ポインタでつながれる。否定演算の処理は、サブグラフに対しても再帰的に適用されるので、既にサブグラフの否定が得られている場合には、否定ポインタをたどるだけでそのサブグラフの否定が得られる。

＜二項論理演算＞

二項論理演算は基本的には図7に示す方法で行う。どちらか一方のグラフが葉の場合には、論理演算の種類に応じて、

s-AND(x, 1) → x , s-AND(x, 0) → 0 ,
s-NAND(x, 1) → s-NOT(x) , ...

のような処理を行うが、そうでない場合は演算の種類にかかわらず同様の処理となる。複数の枝で共有されている節については、同じ演算が複数回適用される可能性があるが、本稿では共有節の間で行われた演算結果を「演算結果テーブル」に記憶しておくことにより、以後の演算を効率良く行う。本稿のグラフ表現では、演算が適用される2つの関数の間でもサブグラフが共有されていることもあるので、二項演算の前処理として、共有されている節の二次元テーブルを用意する。共有節の間で行われた演算結果はこのテーブルに記憶し

```
function s-OP(x,y:node) return node is
begin
    if x.var=0 or x.var=1
            or y.var=0 or y.var =1 then
        {OPの種類に依存した処理を行う}
    elsif x.var=y.var then
        z.var:=x.var;
        z.high:=s-OP(x.high,y.high);
        z.low:=s-OP(x.low,y.low);
        if z.high=z.low then
            return(z.high);
        else
            {節テーブルのチェック}
            return(z);
        end if;
    elsif x.var<y.var then
        z.var:=x.var;
        z.high:=s-OP(x.high,y);
        z.low:=s-OP(x.low,y);
        if z.high=z.low then
            return(z.high);
        else
            {節テーブルのチェック}
            return(z);
        end if;
    elsif y.var<x.var then
        {x.var<y.varの場合と同様の処理}
    end if;
end s-OP;
```

図7 二項論理演算

```
function s-AND(x,y:node) return node is
begin
    if x=y then
        return(x);
    elsif x=y.inv then
        return(0);
    . . .
end s-AND;
```

図8 組み込み規則

ておき、以後、同じ共有節の間で演算が行われるときには、この演算結果テーブルを参照して演算結果を得るようにする。

また、演算により新しい節を生成する際には節テーブルのチェックを行ない、同じ節を重複して生成しないようにする。これにより重複するグラフは生成されず共有されるので、同時に冗長な節の削除も行える。

本稿のグラフ表現では、関数の等価性判定はポインタの比較で行え、否定演算は否定ポインタによって得られる。よって、図8のような規則を演算の手続きに組み込むことにより、二項演算を効率良く行うことができる。

2.4 変数の順序付け

本稿のグラフ表現では、シミュレーションを行う前にあらかじめ使用する変数（記号）の順序を決定しておく必要があるが、グラフの大きさは変数の順序に依存する。例えば、図9に示すように変数の順序によってグラフの大きさが変化し、使用記憶量だけでなく演算時間の効率をも左右する要因となる。したがって、グラフが小さくなるような変数の順序が望ましい。

算術演算回路等のある種の回路に対しては、回路の性質からある程度経験的に適切な変数の順序を決めることができる。加算器では、図10に示すような変数の順序付けをすれば、比較的良い結果が得られることがわかっている。このような回路に対しては、利用者が回路の機能をシミュレータに知らせることによって、暗に変数の順序を決定することができる。制御回路等の回路においては、適切な変数の順序を容易に予測できない場合が多い。しかし、このような回路に対しては、すべての入力に異なる変数（記号）を与えてシミュレーションすることは少なく、出力値が少数の変数にしか依存しないような人間にとって理解できる程度の関数になるように、入力パタンを与える場合が多いと考えている。したがって、変数の順序の違いによる影響は少ないであろうと思われる。

回路の機能を調べるためにシミュレーションを行う場合や、異なる実現法の回路の等価性を判定するような場合には、適切な変数の順序を決定することは難かしく、今後の課題である。

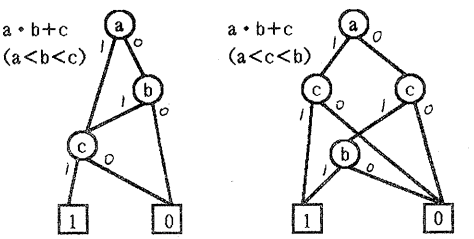


図9 変数の順序付け

An<Bn<An-1<Bn-1<…<A0<B0
A0<B0<…<An-1<Bn-1<An<Bn

図10 加算器における変数の適切な順序

２．５論理式への変換

本稿の論理関数のグラフ表現は、計算機で処理する場合には効率の良い表現であるが、人間にとって理解し易い表現であるとは言い難い。対話的な環境でシミュレーションを行い、シミュレーション結果を利用者が解析して回路の動作を調べるようなシステムでは、内部表現である論理関数のグラフ表現を外部表現としての論理式に変換することが必要であると考えられる。

論理式への変換方法は、基本的には図11(1)のように二分決定図の定義通りに変換して行く。ただし、a･1=a,a･0=0,a･f+0=a･f等の簡単化は当然行う。信号値の内部表現として論理関数のグラフ表現を用いることの利点は、冗長変数の除去が容易に行えることである。グラフ表現においては、冗長な節は除去されているので、変換された論理式においても冗長な変数は含まれず、a･b+aなどのような冗長な変数を含む論理式が出力されることはない。同様に、恒真式や恒偽式は出力されることはなく、1または0と出力されるので利用者は回路の動作を適確にとらえることができる。

また、否定ポインタを利用することによって、図11(2)に示すように排他的論理和（バリティ）演算子を用いた表現も可能であり、式の長さの短いわかりやすい論理式で表わせる。a･(b+c)+$\bar{a}$･c=a･b+cなどの簡単化についてはヒューリスティクな方法で対処する。したがって、式が複雑になる場合には対処しきれないこともあるが、対話的な環境では人間が理解できる程度の比較的簡単な、あるいは規則的な形の論理式になるようにシミュレーションが行われると考えられる。

図12に本稿のグラフ表現を用いた記号シミュレーションの例を示す。否定ポインタが有効に働いていることがわかる。

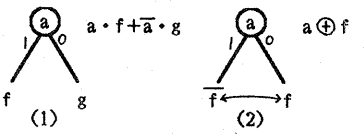


図11 論理式への変換

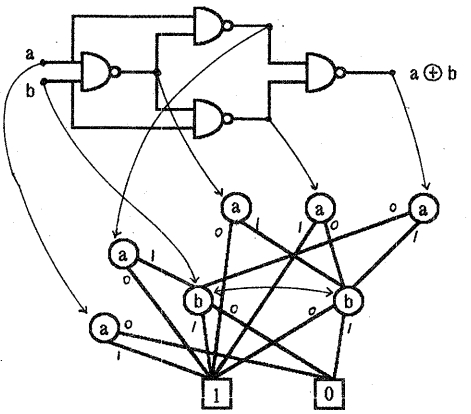


図12 シミュレーション例

３．評価及び考察

本稿で述べた論理関数のグラフ表現を信号値の内部表現とした、コンパイル法によるシミュレータをワークステーションＮＥＷＳ（NWS-830）上のFranz Lispで作成した。ただし、今回作成したシミュレータでは、二項論理演算において共有節の演算結果テーブルを利用していない。表1に順次桁上げ加算器とＡＬＵ(74181)に対するシミュレーション結果を示す。表中の論理演算の回数はシミュレーション中に適用された否定演算と二項論理演算の回数である。否定ポインタを用いる場合の方が実行時間が短かくなっており、その効果

表1 シミュレーション結果

	論理演算の回数	否定ポインタなし		否定ポインタあり		最大のグラフ	
		実行時間（秒）	節テーブルのサイズ	実行時間（秒）	節テーブルのサイズ	出力線名	節の数
4bit加算器	28	0.2	61	0.05	41	Ｆ３	25
8bit加算器	56	3.4	121	0.1	81	Ｆ７	49
１×ＡＬＵ	95	2.9	1419	1.9	1175	Ａ＝Ｂ	246
２×ＡＬＵ	191	29	3358	6.5	2789	Ａ＝Ｂ	490

ただし、変数の順序は　M<S0<S1<S2<S3<An<Bn<…<A0<B0<Cin　である。

が現われている。短縮された実行時間の大部分は、節テーブルのチェック回数の減少によるものであると考えられる。また、否定ポインタを用いる場合の方が節テーブルのサイズが小さいのは、ある関数の否定を表わすグラフは否定ポインタをたどって得ることができるので、節テーブルには登録しなくてもすむような処理を行っているからである。

表4.1は、比較的良い変数の順序でのシミュレーション結果であるが、表4.2では8bit加算器に対して、異なる変数の順序付けによるシミュレーション結果を示す。出力値を表わすグラフが小さくても、節テーブルのサイズが大きくなっている場合(order3)がある。これは回路内の信号値を表わすグラフが大きかったり、複数の関数でのグラフの共有があまり行われていないためであると思われる。したがって、効率の良いシミュレーションを行うには、回路の機能（出力関数）だけでなく回路の構造をも考慮して、変数の順序を決定する必要がある。

4．おわりに

論理関数のグラフ表現を用いた記号シミュレーションについて述べ、シミュレータを作成して評価を行った。複数の関数でのサブグラフの共有化によって使用記憶量を削減し、否定ポインタを用いることによって論理演算を効率良く行えた。しかし、大規模な回路では節テーブルが大きくなりそのチェックに要する時間が多くなるので、チェックに要する時間の短縮とチェック回数を少なくする工夫が必要である。また、効率の良いシミュレーションを行うために、変数の順序をいかにして決定するかが今後の課題である。

表2　8bit加算器のシミュレーション結果

変数の順序	節テーブルのサイズ	出力F7のグラフの節の数	実行時間（秒）
order1	81	49	0.1
order2	173	87	0.2
order3	593	28	25
order4	977	50	25
order5	1563	1023	4.4

ただし、変数の順序は

order1: An<Bn<…<A0<B0<Cin
order2: Cin<An<Bn<…<A0<B0
order3: Cin<A0<B0<…<An<Bn
order4: A0<B0<…<An<Bn<Cin
order5: An<…<A0<Bn<…<B0<Cin

謝辞

本学、安浦寛人助教授及び石浦菜岐佐助手を始め、御討論頂いた矢島研究室の諸氏に感謝いたします。なお、本研究は一部文部省科学研究費補助金による。

参考文献

[1] 石浦，安浦，矢島：ベクトル計算機による高速論理シミュレーション，設計自動化，25-2(1985).

[2] T.Blank:A survey of hardware accelators used in computer-aided design,IEEE Design & Test of Comput.,1-3(1984),21-39.

[3] 斎藤隆夫，上原貴夫：論理回路の記号シミュレーション，情報処理，26-1(1985),7-16.

[4] W.C.Cater,W.H.Joyner,and D.Brand:Symbolic Simulation for Correct Machine Design, Proc. 16th DA Conf.,(1979),280-286.

[5] W.E.Cory:Simbolic Simulation for Functional Verification with ADLIB and SDL,Proc. 18th DA Conf.,(1981)82-89.

[6] S.B.Akers：Binary Decision Diagrams, IEEE Tran. on Comput.,C.27-6,(1978),509-516.

[7] R.E.Bryant：Graph-Based Algorithms for Boolean Function Manipulation, IEEE Tran. on Comput.,C.35-8,(1986),677-691.