(9)

情報処理学会
DA 34－1
設計自動化 34－1
（1986. 10. 21）

Tokioに基づく論理回路の検証

中村 宏・河野 眞治・＊藤田 昌宏・田中 英彦

東京大学工学部・＊富士通研究所

我々は、ディジタルシステムの内、同期部を対象とした時相論理に基づく検証系をPrologを用いて作成してきた。しかし、実行速度、メモリ使用量の点で満足のいくものではなかった。そこで、論理式をカバーで表現することにより、手続き型言語での実装を容易にし、C言語を用いて実際に実装した。

C言語による今回のシステムは、Prolog版に比べて、2桁程度実行速度が向上し、回路の規模の大きさにも対応できる。

LOGIC DESIGN VERIFIER BASED ON TOKIO

Hiroshi NAKAMURA, Shinji KONO, ＊Masahiro FUJITA, and Hidehiko TANAKA

University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, 113 Japan
＊FUJITSU LABORATORIES LTD.
1015 Kamikodanaka, Nakahara-ku, Kawasaki, 211 JAPAN

(英称住所)

We have developed a Logic Design Verifier for synchronization part of digital systems. It is based on Temporal Logic reasoning and implemented on Prolog. As it is implemented on Prolog, the speed and memory usage are not so efficient. So, we express logic formula by cover (sum-of-product form), which makes it easy to implement the Logic Design Verifier by procedural languages. We have implemented it by C language. This system is about 100 times faster than the Prolog system and can handle much larger circuits.

1．はじめに

従来、論理設計の確認はシミュレーションのみで行なわれていた。しかし、適切なシミュレーションデータの作成は難しく、もれも多かった。そこで形式的な検証手法が種々提案・研究されてきたが、そこで最も問題となったのは、”厳密に仕様を記述するのは容易ではない”ということであった。仕様記述が難しいというこの問題は、レジスタトランスファレベルより高位レベルの仕様記述言語がなかったことによる。従って従来は仕様全部を検証しようとはせずに、仕様のうち設計者が特に確認をしたいことのみチェックする方式が実用的であった。

そこで我々はレジスタトランスファレベル以上のレベルを形式的に記述する言語としてTokio を提案し、その処理系を作成してきた［１］。Tokio を用いれば、仕様記述のレベルから柔軟に記述することができる。一方自動合成の研究もさかんに行なわれており、自動合成できれば手間のかかる検証は不要であるが、現状ではなんらかの人手の介入が必要であり、検証は重要である。

我々は、時相論理で仕様記述したものに対して検証や合成を行なうシステムを、ディジタルシステムの内、同期部を対象にして、Prologを用いて開発してきたが、実行速度やメモリ使用量の点で問題があった［２］。そこで今回、検証システムをＣ言語で実装することにし、組合せ回路の部分は、積和形（カバー）に変換して処理する検証プログラムを作成し、従来のPrologによる検証プログラムと比較したので、その結果について報告する。

2．時相論理と状態遷移図

時相論理（Temporal Logic）とは、通常の古典論理に幾つかの時相演算子（Temporal Operator ）を付け加えたものである。

時相論理にも幾つかの種類があるが、ここではLinear Time Temporal Logic（ＬＴＴＬ）［３］を使用する。ＬＴＴＬは連続時間ではなく離散時間上に定義されている。

ＬＴＴＬには４つの時相演算子○，□，◇，Ｕがあり、各々以下のような意味を持っている。

○Ｐ：次の時刻にＰが成り立つ。

□Ｐ：現在から考えて全ての時刻でＰが成り立つ。

◇Ｐ：現在から考えていつかはＰが成り立つ。

ＰＵＱ：現在から考えてＱが成り立つまではＰであり続ける。

これらの時相演算子を用いてハードウェアの仕様記述に必要な様々な性質を表現できる。

たとえば”信号Ｐがactiveになると必ず信号Ｑが次の時刻（クロック）にactiveになる”は

□（Ｐ→○Ｑ）

と表現できる。さらに、信号Ｑがactiveになるのが次の時刻とかぎらない場合には◇演算子を用いて

□（　Ｐ→◇Ｑ）

と表現できる。

次にＬＴＴＬの式を状態遷移図に展開することを考える。

任意の時相論理式は現在と次の時刻に関する条件に展開できるため状態遷移図に展開することができる［４］。各時相演算子にはＬＴＴＬの公理から導かれる次のような性質がある。

①□Ｆ→Ｆ∧○□Ｆ

②◇Ｆ→Ｆ∨（～Ｆ∧○◇Ｆ｛Ｆ｝）

③～□Ｆ＝◇～Ｆ

｛Ｆ｝はeventuality と呼ばれるもので｛｝内の条件がいつか成立することを要求する。

Ｐ，Ｑ，Ｒが時相演算子を含まないとして

（Ａ）□Ｐ

（Ｂ）～（（Ｐ∧○□Ｑ）→○□Ｒ）

を状態遷移図に展開する。

（Ａ）は①を用いると□Ｐの展開から得られる次の時刻に関する条件は□Ｐとなりもとと同じなので状態遷移図は図１のようになる。

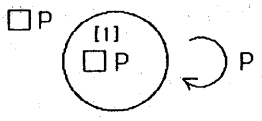


図１．状態遷移図（１）

次に（Ｂ）を展開する。

～（（Ｐ∧○□Ｑ）→○□Ｒ）

＝Ｐ∧○□Ｑ∧～（○□Ｒ）

＝Ｐ∧○□Ｑ∧○◇（～Ｒ）

＝Ｐ∧○（□Ｑ∧◇（～Ｒ））

下線部が次の時刻の状態になり、それはさらに以下のように展開される。

□Ｑ∧◇（～Ｒ）

＝Ｑ∧○□Ｑ∧（～Ｒ∨（Ｒ∧○◇（～Ｒ){～Ｒ}））

＝（Ｑ∧～Ｒ∧○□Ｑ）∨

（Ｑ∧Ｒ∧○（□Ｑ∧◇（～Ｒ){～Ｒ}））

下線部が次の時刻の状態であり、結局状態遷移図は図２のようになる。

時相論理における論理式の充足可能とは、論理式を状態遷移図に展開した時、無限長の状態遷移列がとれることをいう。

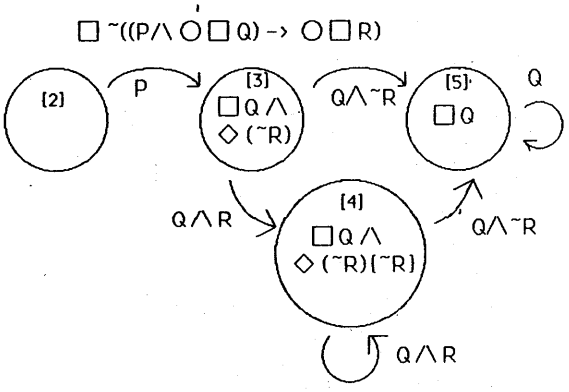


図２．状態遷移図（２）

（Ｂ）の論理式は、[5],[5],[5],... という無限の状態遷移列があるので充足可能である。この場合[4],[4],[4],... という遷移列も無限長に見えるがeventuality を満たしていないので充足可能の理由にはならない。

複数の論理式の積の充足可能チェックも各式の対応する状態遷移図を同時に状態遷移させることによりできる。

例えば、

□（Q∧R）∧～（（P∧○□Q）→○□R）

は図２と図３を同時に状態遷移させればよく、図４のようになる。[4,6],[4,6],[4,6],... の遷移列もeventuality を満たしておらず、この論理式は充足不可能である。

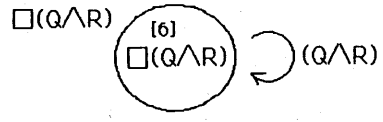


図３．状態遷移図（３）

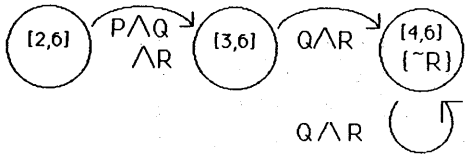


図４．状態遷移図（４）

論理設計検証では以上で述べたことを次のように利用する。

今Ｄを検証すべき設計に対応する時相論理式としＳをそれに対する仕様とすると、検証とは

Ｄ→Ｓ

という論理式が恒真であることを示すことである。背理法を用いると上式の否定

Ｄ∧～Ｓ

が充足不可能であることを示せばよい。

従ってＤ，～Ｓそれぞれに対応する状態遷移表現を求めておきそれらを共に満たす無限長の状態遷移列の有無を調べればよい。無限長の状態遷移列があれば誤設計であり、反例となる。

３．カバー表現を用いた論理設計検証

ここでは、検証対象回路を図５のように分けた組み合わせ回路部分のシミュレーションをカバーを用いて計算し、内部入力変数と内部出力変数との対応（フリップフロップの接続情報）とを用いて高速に検証を行う手法［５］を述べる。

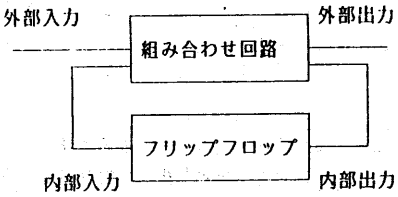


図５．検証対象回路の構成図

まず、キューブとカバーについて説明する。

・キューブとカバー

キューブは１つの積項を、カバーは積和形で表された論理式を表現する。n 入力m 出力の１つの積項p に対し、大きさが２ビットの要素n 個と大きさ１ビットの要素をm 個を持つベクタがキューブであり入力変数の部分をinput part、出力変数の部分をoutput part という。そして与えられた論理式を積和形に変形しキューブの集合として表現したものがカバーである。

例　$f1 = AB + \overline{B}C + AC$

$f2 = \overline{B}C + \overline{C}\overline{D}$

という論理式は、

A	B	C	D	f1	f2	
01	01	11	11	1	0	AB
11	10	01	11	1	1	$\overline{B}C$
01	11	01	11	1	0	AC
11	11	10	10	0	1	$\overline{C}\overline{D}$

という４つのキューブの集合からなるカバーで表現される。A，B，C，Dが入力変数でありf1，f2が出力変数である。input partには01,10,11のみが許されており00は許されない。

・intersection

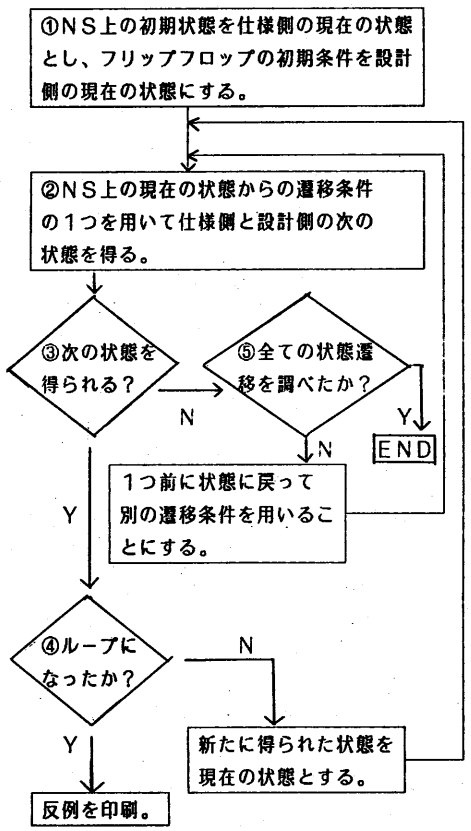
２つのキューブ間の論理積演算としてintersection

が定義される。2つのキューブPとQのintersectionP・Qはキューブであり各要素のビット積をとることにより求まる。

		[A	B	C	f1]
例	f1=AB	[01	01	11	1]
	f1=BC	[11	01	01	1]
	f1=ABC	[01	01	01	1]

・onカバーとoff カバー

ある出力変数に対してそれを1とするような入力変数の論理式をonカバーといい、0とするような入力変数の論理式をoff カバーという。



※NS（Negation of Specification）は仕様の否定を状態遷移図に展開したものを表す。

図6．検証のフローチャート

カバー表現を用いた検証のフローチャートは図6のようになる。

このフローチャートを例を用いて説明する。使用する例はハンドシェイクを用いてデータ転送を行うシステムの受信側の状態遷移制御部である。この回路に対して2つのフリップフロップが初期状態でリセットされているという条件で、□（Call→◇Hear）という仕様を満たすかを検証する。

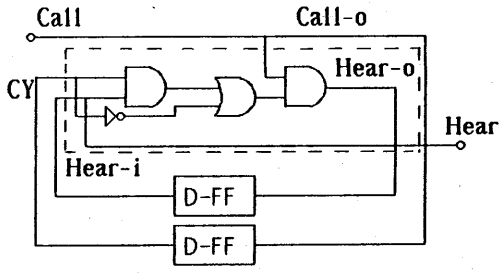


図7．データ転送システムの受信側の状態遷移制御部

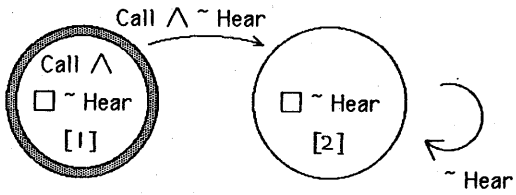


図8．仕様の否定の状態遷移図

（準備）（Call,CY,Hear-i,Call-o,Hear-o,Hear ）の形でキューブを表現すると、図7の組合せ回路の部分（枠で囲ってある）のonカバーとoff カバーはそれぞれ

$$Con=\begin{bmatrix}01&11&11&1&0&0\\01&10&11&0&1&0\\01&01&01&0&1&0\\11&11&01&0&0&1\end{bmatrix}$$

$$Coff=\begin{bmatrix}10&11&11&1&0&0\\10&11&11&0&1&0\\11&01&01&0&1&0\\11&11&10&0&0&1\end{bmatrix}$$

となる。

（例えば、onカバーの2番目と3番目のキューブより、Hear-oがonになるのはCallがonでCYがoff の場合とCall,CY,Hear-i全部がoff の場合であることが分かる。）

フリップフロップの接続情報は、○Hear-i=Hear-o,○CY=Call-oである。

また仕様の否定、~□（Call→◇Hear）を状態遷移図に展開すると（即ちNS）図8 のようになる。

①仕様側の初期状態は図８の[1] である。
またフリップフロップがリセットである条件は
Ccond=[11　10　10　　1　　1　　1]である。
②ＮＳ上の[1] からの遷移は（Call∧～Hear）である。Callは入力変数なので、キューブ
[01　11　11　　1　　1　　1]
が得られ、Hearは出力変数なのでHearに対するoffカバーから
[11　11　10　　1　　1　　1]
が得られる。
　従って（Call∧～Hear）に対するカバーは
Ct =[01　11　10　　1　　1　　1]となる。仕様側の次の状態は[2] であり、設計側の次の状態は
Cnext-on =Ccond・Con・Ct 及び
Cnext-off=Ccond・Coff ・Ct から求まる。
③ＮＳ上では次の状態[2] が得られている。設計側の次の状態は、Cnext-on とCnext-offから求まる。もしある出力変数に対する要素の値が１となるキューブがどちらのカバーにもない時は設計側の次の状態が得られなかったことを表わす。Cnext-on のみに１となるキューブがない時はその出力変数の値を０とし、Cnext-offのみにない時は１とする。そして、フリップフロップの接続情報から次の設計側の状態を求める。この場合
Cnext-on =[01　10　10　　1　　1　　0]
Cnext-off=[01　10　10　　0　　0　　1]
であるからCall-oとHear-oの値は１でありHearの値は０である。従って接続情報から
Cnext=[11　01　01　　1　　1　　1]
という設計側の次の状態が得られ、④にいく。
④仕様側と設計側で両方共ループになっているかを見る。両方共ループでしかもeventuality を満たしていれば、それは反例である。この場合ループではないので、仕様側の状態を[2] 、設計側はCnextを新たな状態Ccondとして②へいく。
②今度のCt は[11　11　10　　1　　1　　] でありCt ・Ccond=nil なので、Cnext-on とCnext-offは共にnil である。従って設計側の次の状態が得られないので、③から⑤へいく。
⑤ＮＳ上の全ての遷移を調べているので検証は終了する。

４．検証システム

この検証システムは図９のような構成をしている。

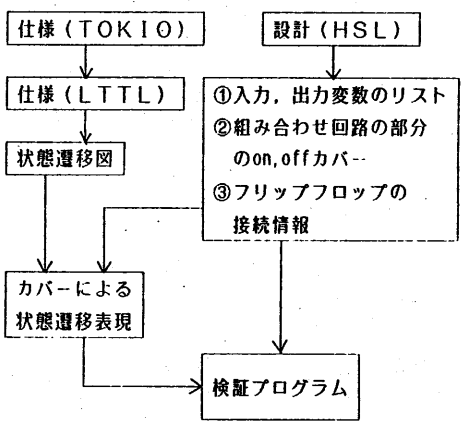


図９　検証系の構成

仕様の状態遷移図をカバーに変換する時は、変数の並びを揃えなければならないので、①が必要になる。
　現在実装されているのは右半分であり、左半分は、Prolog版［１］を用いる。

５．検証システムの評価

ハンドシェイクを用いてデータ転送を行うReceiver及び富士通のミニコンピュータＵ－３００のＤＭＡ制御回路の２つの例題に付いて検証を行い以前に作成したPrologによるシステムと今回実装したＣ言語によるシステムを比較・検討する。使用計算機はＶＡＸ11／730 （0.2 ～0.3MIPS)であり、Prologはその上に載っているC-prologを用いた。今回作成したシステムでは、仕様の否定の状態遷移図はProlog版が作成したものを用いている。
（１）Receiver
　図１０のような回路であり、これについて仕様
　□（Reset →□（Call→◇Hear））
を検証した。
　検証時間は表１、表２の通りである。

状態の記憶

図５のフローチャートで、一度調べた状態を記憶しておくと、バックトラックして別の遷移状態に来たときそれが以前調べた状態と同じであればその後の状態は一度求めてあるので再び求めなくても済む。これが状態の記憶である。

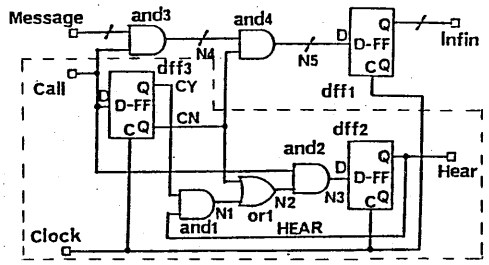


図１０．Receiverの回路図

	状態の記憶有	状態の記憶無
演算部データ幅１bit	4.28	9.88
演算部データ幅２bit	28.25	204.80
演算部データ幅３bit	253.30	5645.80
絞り込んだ回路	0.93	0.87

表１．　Prologによるシステムで検証にかかるCPU時間[sec]（Receiver）

	カバー生成のみ	検証
演算部データ幅１bit	1.3	1.9
演算部データ幅４bit	3.3	3.9
演算部データ幅８bit	5.4	6.1
演算部データ幅８bit で絞り込みを実施	4.2	4.5

表２．　C言語によるシステムで検証にかかるCPU時間[sec]（Receiver）

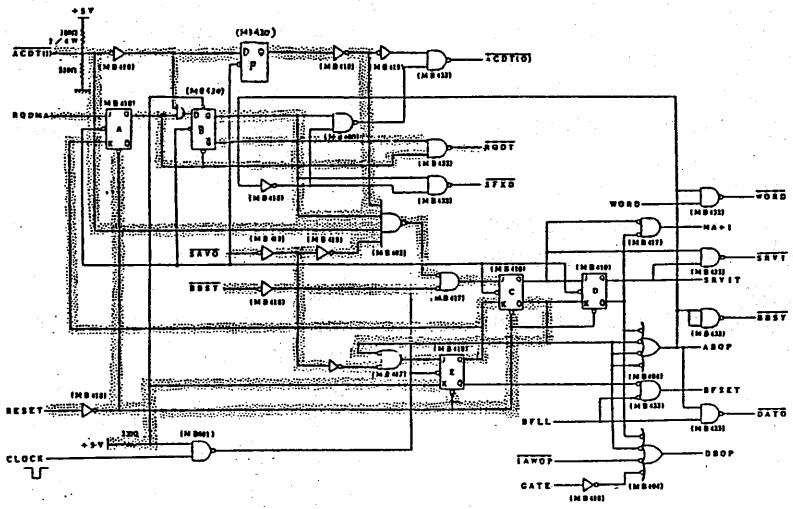


図１１．DMA制御部の回路図

C言語によるシステムではカバー同士のintersectionという簡単な操作で次の状態が得られるのに対し、Prologによるシステムでは、次の状態を得るためには回路内の全てのゲートに関してunificationをするので時間がかかる。従ってPrologによるシステムでは状態の記憶が有効であったが、今回実装したC言語によるシステムでは状態の記憶はしていない。

絞り込み

この仕様のように出力変数がHearしかない場合には、回路をHearについて絞り込んだ回路（図１０の点線内、実は図７と同じ）について検証すれば良く、２つのシステムとも絞り込みの機能を持っている。

Prologによる実装では、ＨＳＬの記述をホーン節に変換したものを検証プログラムの入力としており、この変換時に絞り込みを行う。表１の”絞り込んだ回路”の欄のデータは、この絞り込まれたデータを入力としたときの測定時間である。

C言語による実装では、ＨＳＬの記述をカバーに変換する部分と検証の部分が１つのプログラムになっている。従って、表２では絞り込まれたデータが入力ではなく、カバーに変換する際に絞り込んだ後に検証した測定時間である。

（２）DMA制御回路

回路図は図１１であり、この回路に対して

①□((Reset ∧○□～Reset ∧□～Acdt)
　→○□(Rqdma→○Rqdt)

②□((Reset ∧○□～Reset ∧○□～Rqdma)
　→○□(Acdt →○～Rqdt)

の２つの仕様を検証した。

検証時間は表３、表４の通りであった。Prologによるシステムで絞り込まれている部分は図１１内の点線部である。

	状態の記憶無	状態の記憶有
①絞り込み無	＞60000	＞60000
①絞り込み有	＞60000	2672.05
②絞り込み無	＞60000	＞60000
②絞り込み有	＞60000	1923.35

表３．　Prologによるシステムで検証にかかるCPU時間[sec]（DMA制御回路）

	カバー生成	検証
①	16.1	26.4
②	16.1	25.5

表４．　C言語によるシステムで検証にかかるCPU時間[sec]（DMA制御回路）

評価

２つの例について、Prologによるシステム（状態の記憶がある場合）とC言語によるシステムとを比較する。

回路の規模が小さい場合（Receiverでデータ幅１bit）は、２つのシステムの検証時間のオーダーは同じであるが、回路規模が大きい場合にはProlog版では検証時間は飛躍的に増大する。それに比べ、C言語版は回路の規模の増大に十分に対応しており、DMA制御回路の例ではProlog版より２桁程度速くなっている。

６．中間変数を用いた検証法

これまでの例は比較的規模が小さく、DMA制御回路でゲート数約３０、外部入力変数１２であった。

しかし、回路の規模が大きくなり入力変数の数が増えると検証対象回路の組み合せ回路の部分を一度にカバーに変換するのは容易ではなくなる。一般にカバーのキューブの数は入力変数の数をｎとしたとき最悪2^{n-1}になる。組み合わせ回路の論理が複雑な時に最悪に近づく。同期回路の組み合わせ回路の部分の論理は演算回路に比べそれほど複雑ではないが論理が複雑であっても対応できる手法として、中間変数の導入を考えた。中間変数を用いると、出力変数に対する論理が簡単になるためカバーの生成が簡単になる。

中間変数を用いると検証対象回路の構成は図１２のようになる。但し中間出力と中間入力は１対１に対応している。

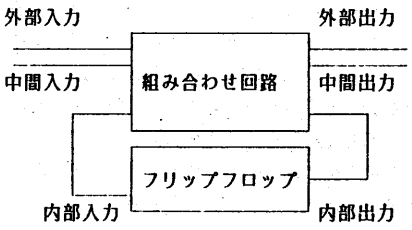


図１２．中間変数を用いた検証対象回路の構成図

検証のフローチャートは３節で述べた図５と同じだが実際の操作に少し違いがある。それは、組み合わせ回路のon,offカバーを用いる部分であり、

（１）仕様側の遷移条件をカバーに変換する部分

（２）次の時刻のフリップフロップの初期条件を求める部分

である。

（準備）組み合わせ回路の部分のonカバーCon, off カバーCoff をもとめる。このときキューブの表現は

[Ｉ，ＦＩ，ＴＩ，Ｏ，ＦＯ，ＴＯ]になる。

但し、Ｉ：外部入力，ＦＩ：内部入力，ＴＩ：中間入力，Ｏ：外部出力，ＦＯ：内部出力，ＴＯ：中間出力，をあらわし、input partはＩ，ＦＩ，ＴＩであり、output part はＯ，ＦＯ，ＴＯである。

（１）遷移条件のカバーCt を求める。遷移条件に外部出力変数が含まれているとすると、その外部出力変数に対するonカバー(あるいはoff カバー）を用いるので、

Ct ＝[Ｉ，ＦＩ，ＴＩ，Ｏ，ＦＯ，ＴＯ]

の形になる。Ct のoutput part は全て１である。ここで、Ct のinout partから中間変数を除去する。

中間変数の除去

Ct の中間入力変数TI_iが01(10)であればTI_iに対応する中間変数TO_iのonカバー(offカバー)のinput partとCt のinput partのintersectionを新たなCt のinput partとし、TI_iの項は11にする。TO_iのonカバー(offカバー)はCon(Coff)から得る。これを、Ct の中間入力変数が全て11になるまでする。

この操作の間に、I，FI，TIの中のある変数の項が00になったら、そのキューブはnil であり消去する。もし、操作の結果Ct のキューブが全て消去されてCt がnil になったら、その遷移条件が満たされないことを表すので、別の遷移条件を探す。(2) 3節と同様にして求められたCnext-on，Cnext-offのinput partから、今述べた方法で再び中間入力変数を除去する。その後の処理は同じである。

7．おわりに

論理式の表現にカバーを用いることで、論理設計検証が以前のProlog版によるシステムよりもオーダーの単位で速くなり、また扱う回路の規模が大きいほどその効果が大きいことを示した。さらに、中間変数を導入することで扱える回路の規模がさらに大きい場合にも対応できることを示した。

今後は、仕様の否定を高速に状態遷移図に変換する部分、及び中間変数の扱える検証系を実現していきたい。

参考文献

[1] M. Fujita :" Logic Design Assistance with Temporal Logic", IFIP 7th CHDL (1985)

[2] 河野：" 時相論理型言語Tokioの実装", Logic Programming Conference 8.1 (1985)

[3] Z. Manna :" Verification of Concurrent Programs . Part1. The Temporal Framework", Stanford Univ.Report STAN-CS-81-836 (1981)

[4] 河野：" 時相論理型言語Tokioの検証", Logic Programming Conference 10.2 (1986)

[5] 藤田：" 時相論理を用いた論理設計検証およびテスト生成のカバー表現に基づく高速化", 電子通信学会 FTS研究会 Nov. (1985)