時相論理に基づく論理回路検証システム†

中村 宏†† 藤田昌宏†††
河野真治†††† 田中英彦††

近年のディジタルシステムの大規模化，複雑化に伴い，論理設計段階での設計支援が強く望まれている．論理設計段階での検証に関しては，従来シミュレーションによるものがほとんどであったが，適切なシミュレーションデータの作成は難しく，必ずしもすべての誤りを検出できるわけではない．そこで形式的な検証手法が種々提案・検討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．これはハードウェアシステムに内在する並列性・順序性を簡単に記述する言語がないことによる．我々は，並列性・順序性を容易かつ厳密に記述できる時相論理で記述した仕様に対して検証を行うシステムを，ディジタルシステムの制御部 (control part) の同期回路を対象にして作成した．本論文では，その検証システムで用いている検証手法，および検証結果を述べる．実際に例題を検証した結果，本システムは実行速度，メモリ使用量の点で実用的であるといえる．制御部の設計は，演算部 (function part) の設計と比較して，回路の規模としては小さいがタイミングが複雑であり，人間にとっては誤設計を起こしやすい部分であり，本システムの利用価値は大きいと考える．

1. はじめに

近年のディジタルシステムの大規模化，複雑化に伴い，論理設計段階での設計支援が強く望まれている．論理設計段階での検証に関しては，従来レジスタトランスファレベルやゲートレベルにおけるシミュレーションによるのがほとんどであった．しかし，適切なシミュレーションデータの作成は難しく，必ずしもすべての誤りを検出できるわけではない．そこで形式的な検証手法が種々提案・検討[1)~3)]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．これはハードウェアシステムに内在する並列性・順序性を簡単に記述する言語がないことによる．そこで，我々は，並列性・順序性を容易かつ厳密に記述できる時相論理で記述した仕様に対して検証や合成を行うシステムを，ディジタルシステムの制御部 (control part) を対象にして，Prolog を用いて開発したが[4)]，実行速度やメモリ使用量の点で問題があった．そこで今回新たに，検証対象回路の組合せ回路部分を積和形（カバー）に変換して処理することにより速度を向上させた検証システムをC言語で実装し[5)]，さらに文献4）で用いた“状態の記憶”手法を，積和形を用いたC言語によるシステムでも実装した．2章では時相論理に基づく検証手法を示し，3章で本システムで実装した，カバー表現を用いた検証法を示す．4章では，検証結果を示し，5章で処理能力について評価・検討する．

2. 時相論理に基づく検証手法

2.1 時相論理と状態遷移図

時相論理 (Temporal Logic) とは，通常の古典論理にいくつかの時相演算子 (Temporal Operator) を付け加えたものである．時相論理にもいくつかの種類がある[6),7)]が，ここでは Linear Time Temporal Logic (LTTL)[8)] を用いる．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：現在から考えてすべての時刻で 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．

例えば，“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)

† Logic Design Verification System Based on Temporal Logic by HIROSHI NAKAMURA (Department of Electrical Engineering, Faculty of Engineering, University of Tokyo), MASAHIRO FUJITA (Fujitsu Loboratories Ltd.), SHINJI KONO (Sony Computer Science Laboratory Inc.) and HIDEHIKO TANAKA (Department of Electrical Engineering, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部電気工学科
††† (株)富士通研究所
†††† (株)ソニーコンピュータサイエンス研究所

と表現できるし，さらに，信号Qが active になるのが次の時刻と限らない場合には，◇演算子を用いて

$$\Box(P\rightarrow\Diamond Q)$$

と表現できる．

次に，LTTL の式を状態遷移図に変換することを考える．LTTL の公理から導かれる以下の性質を用いると，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できる[9]．

① $\Box F\rightarrow F\cap\bigcirc\Box F$

② $\Diamond F\rightarrow F\cup(\sim F\cap\bigcirc\Diamond F\{F\})$ (以降 "～" は否定を表すことにする)

③ $F1\,UF2\rightarrow$
$F2\cup(F1\cap\sim F2\cap\bigcirc(F1\,UF2))$

④ $\sim\Box F=\Diamond\sim F$

例えば②は，"いつかFが成り立つということは，現在Fであるか，あるいは，現在はFではなく次の時刻からみていつかFが成り立つ" を意味する．ただし，いつも "$\sim F\cap\bigcirc\Diamond F$" を選択しているとずっと$\sim F$となり，◇$F$を満たさない．したがって，"$\sim F\cap\bigcirc\Diamond F$" は，いつかは必ず$F$が成立するという条件付きで選択しなければならない．これは，eventuality と呼ばれるもので，"$\sim F\cap\bigcirc\Diamond F$" の後ろの $\{F\}$ は，この eventuality を表す．

例として，P, Q が時相演算子を含まないとし，

(A) $\Box P$

(B) $\sim((P\cap\bigcirc\Box Q)\rightarrow\bigcirc\Box R)$

を状態遷移図に展開するとそれぞれ図 1(a)，(b)のようになる．

時相論理において論理式が充足可能であるとは，その論理式を状態遷移図に変換したときに無限長の状態遷移列が存在することをいう．

(B)の論理式は，〈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〉,…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，充足可能の理由とはならない．

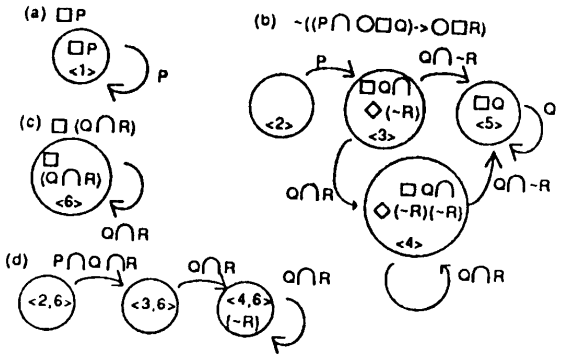


図 1 状態遷移図
Fig. 1 State diagrams.

複数の論理式の積の充足可能性チェックも各式に対応する状態遷移図を同時に遷移させることにより可能である．例えば，

$$\Box(Q\cap R)\cap\sim((P\cap\bigcirc\Box Q)\rightarrow\bigcirc\Box R)$$

は，図1(b)，(c)を同時に遷移させればよく，図1(d)のようになる．〈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．

2.2 時相論理に基づく検証手法

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を仕様に対する時相論理式とすると，検証とは

$$D\rightarrow S$$

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

$$\sim S\cap D$$

が充足不可能であることを示せばよい．

したがって，$\sim S$, D 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ともに満たす無限長の状態遷移列の有無を調べればよい．実際の検証システムでは，D（設計）に対応する状態遷移図は大きくなるので，まず$\sim S$（仕様の否定）をあらかじめ状態遷移図に展開し，$\sim S$を満たすように設計側の状態を遷移させていく．そこで無限長の遷移列があれば誤設計であり，その遷移列自身が反例となる．

3. カバー表現を用いた論理設計検証

図2が，前節で述べた検証手法を実装した検証システムの構成図である．一般にディジタルシステムは，各端子間のデータ転送のタイミングを扱う制御部と，ALU のように実際に計算を行う演算部 (function part) に分けられるが，本システムはディジタルシステムの制御部の同期回路を対象にしている．LTTL は前節で述べた時相論理であり，HSL[10] はゲート間の接続情報のみを記述するハードウェア記述言語である．LTTL から状態遷移図に変換する部分は Prolog で実装し，その他の部分はC言語で実装した．

仕様に対する状態遷移図は前節で述べた手法で LTTL の式を展開して求める．一方設計側の状態は

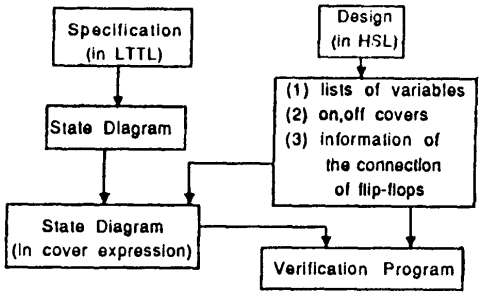


図 2 検証システムの構成図
Fig. 2 Structure of the verification system.

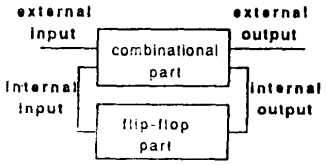


図 3 同期回路の構成
Fig. 3 Structure of synchronous circuits.

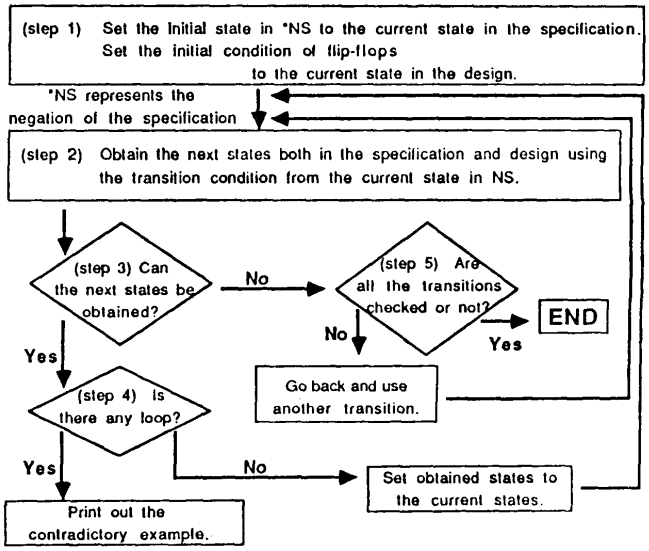


図 4 検証の処理の流れ
Fig. 4 Flowchart of the verification.

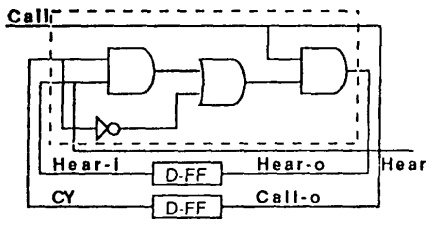


図 5 検証例
Fig. 5 Example of design.

フリップフロップの状態である．このシステムでは，検証対象回路を図3のように分けた時の組合せ回路部分をカバー表現[11]で扱う．このカバー表現を用いて，順次フリップフロップの状態を求めることにより，設計側の状態遷移図を求める．カバー表現は，PLA の簡単化等に使用する論理式簡単化に利用されるなど研究が進んでおり，複雑な論理式に対してはかなりコンパクトな表現法である．まずカバー表現について説明する．

3.1 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2bit の要素 n 個と1bit の要素 m 個を持つベクタがキューブであり，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. $f1=AB+\bar{B}C+AC$

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

という論理式は，

A	B	C	D	$f1$	$f2$
01	01	11	11	1	0
11	10	01	11	1	1
01	11	01	11	1	0
11	11	10	10	0	1

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．入力変数部には 00 は許されない．

• intersection

2つのキューブ (P,Q) 間の論理積演算として，intersection $(P\cdot Q)$ が定義され，キューブの各要素のビット積をとることによって求まる．

• 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．

3.2 カバー表現を用いた検証アルゴリズム

カバー表現を用いた検証のフローチャートは図4のようになる．図4において NS は仕様の否定に対応している．このフローチャートを，例を用いて説明する．図5の回路に対し，フリップフロップの初期状態がリセットの時，仕様 □(Call→◇Hear) を満たすかを検証する．

(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5の組合せ回路の部分(点線内) の on, off カバーは

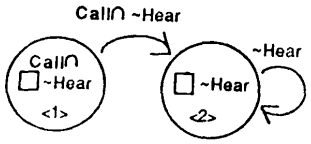


図 6 仕様の否定の状態遷移図
Fig. 6 State diagram for the negation of the specification.

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

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

となる．また，フリップフロップの接続情報は○Hear-i=Hear-o, ○CY=Call-o である．

仕様の否定，～□ (Call→◇Hear) を展開すると (すなわち NS) **図 6** になる．

(step 1) 仕様側の初期状態は図 6 の〈1〉であり，フリップフロップのリセット条件は

Ccond=[11 10 10 1 1 1].

である．

(step 2) NS 上の遷移条件 (Call∩～Hear) に対応するカバーは

Ct=[01 11 10 1 1 1]

である．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

および

Cnext-off=Ccond·Coff·Ct

から求める．

(step 3) NS 上の状態はすでに〈2〉として得られている．もしある出力変数に対する要素が 1 となるキューブが Cnext-on と Cnext-off の両方にないときは，設計側の次の状態が得られなかったことを表す．Cnext-on のみに 1 となるキューブがないときはその出力変数の値を 0 とし，Cnext-off にのみないときは 1 とする．そして，フリップフロップの接続情報より，次に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, Hear-o の値は 1 で，Hear の値は 0 である．したがって，接続情報より

Cnext=[11 01 01 1 1 1]

という状態が得られる．

(step 4) この場合まだループ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 (step 2) へいく．

(step 2) 今度の Ct は [11 11 10 1 1 1] であり，Ct·Ccond=nil なので，Cnext-on, Cnext-off とも nil である．設計側の次の状態が得られないので (step 5) にいく．

(step 5) NS 上のすべての遷移を調べているので，検証は終了する．

3.3 検証の速度向上の手法

検証する際の速度を向上させるためにとった手法についてここでは述べる．

(1) 絞り込み (filtering)

これは，回路全体を調べなくても仕様を検証できる場合に，仕様に関連する部分だけを，検証対象回路から絞り込む手法である．仕様にでてくる出力変数に注目し，その出力から入力側へとさかのぼれば絞り込める．この絞り込みは，HSL の記述をカバー表現に変換するときに，ユーザがシステムに対し出力変数を指定することにより自動的に行われる．

(2) 状態の記憶 (memorizing states)

図 4 のフローチャートで一度調べた状態を記憶しておくと，別の状態遷移が得られたとき，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調べなくても済む．これが状態の記憶である．

4. 検証結果

ハンドシェイクを用いてデータ転送を行う Receiver 回路，および富士通のミニコンピュータ U-300 の DMA 制御回路[12]の 2 つの例題の検証を行った．検証結果をここでは示す．データの測定は SUN 3/260 (約 4 MIPS) を用いて行った．

(1) Receiver 回路

図 7 の回路に対し仕様□(Reset→□(Call→◇Hear)) を検証した．

この仕様では，出力変数が Hear しかないため，絞り込むと図 7 の点線内 (実は図 5 と同じ) になる．

したがって，絞り込めばデータパス部のビット幅は検証時間に影響を与えない．

① 検証時間

各部分で要した CPU 時間は**表 1** のようであった．

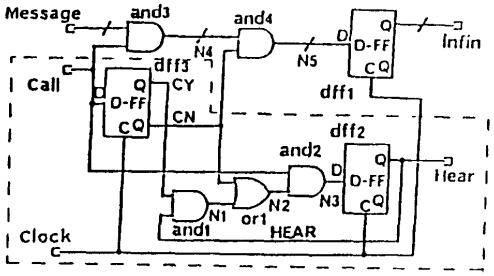


図 7 Receiver 回路
Fig. 7 Receiver by handshaking.

表 1 Receiver 回路の検証結果
Table 1 Results of verifying the receiver.

CPU time SUN 3-260 [sec] 4 MIPS		HSL ↓ cover	State diagram ↓ cover	Verification part Memorizing states without	with
Bit width of data path (not filtered)	1bit	0.14 (0.12)	0.07	0.07	0.07
	4bits	0.29 (0.22)	0.12	0.13	0.13
	8bits	0.56 (0.30)	0.27	0.27	0.29
	16bits	1.49 (0.64)	0.84	0.84	0.94
Filtered		*	0.06	0.06	0.06

* Since the design is filtered in making covers, the time depends on the bit width of data path. The time for each width is shown above in (parentheses).

LTTL の式の展開は 1 秒以内で済む.

② トレースした状態数

(〜仕様∩設計) の状態遷移図上で, トレースした状態の数は 2 個である. これは状態の記憶の有無およびデータパスのビット幅に関わらない.

③ メモリ使用量

検証のみに要するメモリ使用量は, データパスのビット幅が 16, 絞り込みなしで, 状態の記憶の有無に関係なく約 30 KB であり, 十分少ない. 最もメモリを使うのは, HSL の記述をカバー表現に変換するときであり, 絞り込みなし, ビット幅 16 の時に約 130 KB である. 入力変数の数に対する on, off カバーの大きさを表 2 に示す.

(2) DMA 制御回路

図 8 の回路に対し

(1) □((Reset∩○□〜Reset∩□〜Acdt)→○□(Rqdma→○Rqdt))

(2) □((Reset∩○□〜Reset∩○□〜Rqdma)→○□(Acdt→○〜Rqdt))

の 2 つの仕様を検証した.

出力変数 Rqdt について絞り込まれている部分は, 図 8 の点線部である.

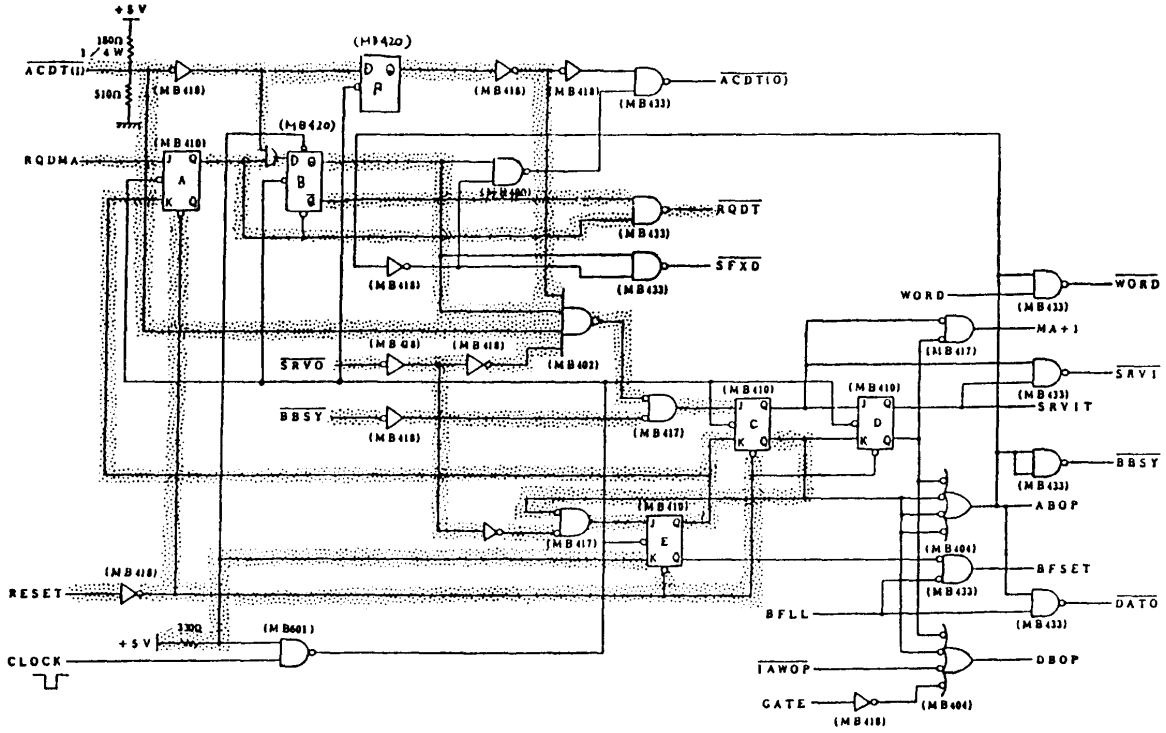


図 8 DMA 回路
Fig. 8 DMA controller.

表 2 Receiver 回路のカバーの大きさ
Table 2 Size of covers for receiver.

		Sum of inputs	Sum of outputs	Sum of cubes (on cover)	Sum of cubes (off cover)
Bit width of data path (not filtered)	1 bit	5	5	6	8
	4 bits	11	11	12	20
	8 bits	19	19	20	36
	16bits	35	35	36	68
Filtered		3	3	4	4

表 3 DMA 回路の検証結果
Table 3 Results of verifying DMA.

CPU time SUN 3-260 4MIPS [sec]		HSL ↓ cover	State diagram ↓ cover	Verification part: Memorizing states without	Verification part: Memorizing states with
Not filtered	(1)	1.36	0.44	0.66	0.58
	(2)	1.36	0.44	0.65	0.54
Filtered	(1)	1.13	0.21	0.27	0.23
	(2)	1.13	0.21	0.26	0.22

表 4 DMA 回路のカバーの大きさ
Table 4 Size of covers for DMA.

	Sum of inputs	Sum of outputs	Sum of cubes (on cover)	Sum of cubes (off cover)
Not filtering	16	18	46	59
Filtering	11	6	11	32

① 検証時間

各部分で要した CPU 時間は**表 3** のようであった．LTTL の式の展開は2つの仕様とも約2秒である．

② トレースした状態数

(〜仕様∩設計) の状態遷移図上でトレースした状態の数は，絞り込みの有無には影響を受けない．状態の記憶をしない場合は，仕様(1)で7個，仕様(2)で6個であり，状態の記憶をする場合，仕様(1)で6個，仕様(2)で5個である．

③ メモリ使用量

検証のみに要するメモリは絞り込みなしの時で，状態の記憶の有無に関係なく，両方の仕様の場合も約 50 KB である．HSL の記述をカバーに変換する際に要するメモリは絞り込まない場合で約 110 KB である．カバーの大きさは**表 4** のとおりである．

5. 評価・検討

5.1 メモリ使用量

2つの例題については，メモリ使用は十分小さいといえる．最もメモリの使用量が多いのは HSL の記述をカバーに変換するときである．入力変数 n に対し，on, off カバーのキューブの大きさは，論理がパリティ関数の時が最悪で 2^{n-1} になる．しかし，本システムが対象にしている制御部は，演算部と比較した場合，ビット幅を持つパスがなく入力変数が少なく，また論理も単純であることが多いため，問題はあまり起こらない．実際，表2と表4ではキューブの数は2の (入力変数の数−1) 乗よりはるかに小さくなっている．

5.2 実行速度

表1，表3で示すとおり，2つの例題に対しては，この検証システムは十分に高速である．

すでに開発したカバー表現を用いないシステム[4]で，図8の例題の検証のみに要する CPU 時間を示す．絞り込みあり・状態の記憶ありで，仕様(1)に対して 6.7 秒，仕様(2)に対して 4.8 秒である．絞り込みまたは状態の記憶を行わない場合は，いずれの場合も 150 秒以上である．(ただし，文献 4)のシステムは VAX 11/730 の C-Prolog 上に実装されているため，上で述べた数値は SUN 3/260 の Quintus-Prolog に LIPS 値換算したものである．) したがって，カバー表現による本論文のシステムは，これを用いなかったシステム[4]に比べ，絞り込み・状態の記憶をともに行う場合で約 20〜30 倍の速度向上を実現している．絞り込みまたは状態の記憶を行わない場合にはこの速度比はさらに向上している．

ここで，検証速度の向上のためにとった2つの手法について考察する．

(1) 絞り込み

検証に要する時間は，(〜仕様∩設計) の状態遷移図上でトレースすべき状態遷移数とカバーの大きさの積に比例する．絞り込みの手法は検証対象回路を検証すべき仕様に関係する部分のみに絞り込むことによって，カバーの大きさを抑えるものである．前節で示した実行結果では，2つの例題とも絞り込みが効果的に行われている．一般に，制御部の回路では，ある出力変数に対しすべての入力変数が影響を与えることは少ないので，その場合にはこの手法は大変有効になる．

(2) 状態の記憶

状態の記憶は，以前に調べた状態を記憶することにより実際にトレースする状態遷移数を減らして，速度の向上を計るものである．その反面，状態を覚えるため，メモリの使用量は増加する．前節で示した実行結果では，調べるべき状態遷移図が小さかったため，この手法は，さほど速度の向上をもたらさなかった．

(～仕様∩設計) の状態遷移図では，～仕様をみたしながら設計側の状態を遷移させるので～仕様の状態数を抑えることができれば，全体の状態遷移数を減らすことができる．実際に，検証すべき仕様の1つ1つは一般にそれほど複雑ではないので，それぞれを別々に検証することにすれば，～仕様に対応する状態遷移図を小さくできる．したがってこの方法が効果を発揮するのは，ある程度仕様が複雑で状態遷移図が大きくなった場合であろう．

5.3 大規模回路の検証

制御部に対象を限定すれば最悪の場合はあまり起こらないと述べたが，論理の複雑な制御部も存在し，また回路の規模が大きくなれば，やはりカバーの大きさは入力変数の数に対し指数的に増大する．検証対象回路の組合せ回路の部分を一度にカバーに変換できない場合の対処法として以下の2つが考えられる．いずれの方法も検証手法自身を変更する必要はない．

① 中間変数の導入

これは，検証対象回路をいくつかに分割し，それぞれの部分についての出力変数に対する入力変数の論理を簡単にすることによって，カバーの大きさを抑えるものである．この場合，検証する際のオーバヘッドが増大する．

② カバー表現以外の論理式の表現法の導入

例えば，文献 13)で提案されているグラフに基づく表現法では，カバー表現では苦手なパリティ関数を入力変数 n に対し $2n+1$ 個の頂点を持つグラフで表現できる．しかし，カバー表現でコンパクトに表現できる論理式を必ずしもこの表現法はコンパクトに表現できない．したがってそれぞれの表現法を用いた検証システムを両方実装し相補的に用いるのが有効であろう．

6. おわりに

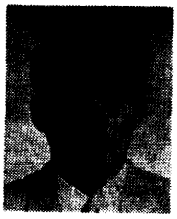
ディジタルシステムの制御部の同期回路を対象に，時相論理で記述した仕様に対して，高速に検証を行うシステムの検証手法および検証結果を示した．実際に例題を検証した結果，本システムは実行速度，メモリ使用量の点で実用的であり，回路の規模の増大に対しても回路を絞り込むことによって，ある程度耐えられる．制御部の設計は，演算部の設計と比較して，回路の規模としては小さいがタイミングが複雑であり，人間にとっては誤設計を起こしやすい部分であり，本システムの利用価値は大きいと考える．

謝辞 有益な御意見をいただいた大森匡氏，および田中研究室の諸氏に感謝します．

参考文献

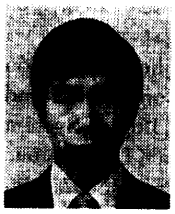
1) Gordon, M. J. C.: LCF-LSM, Univ. of Cambridge Computer Laboratory Technical Report, No. 41 (1983).
2) Barrow, H. G.: VERIFY: A Program for Proving Correctness of Digital Hardware Design, *Artif. Intel.*, Vol. 24, No. 1-3, pp. 437-492 (1984).
3) Kimura, S. and Yajima, Y.: The Description and Verification of Input Constraints and Input-Output Specification of Logic System Using a New Extended Regular Expression, *Proc. Int. Conf. on Very Large Scale Integration '87*, pp. 113-120 (1987).
4) 藤田，田中，元岡：時相論理によるハードウェア記述と Prolog を用いたゲート回路の検証，情報処理学会論文誌，Vol. 25, No. 2, pp. 173-179 (1984).
5) Nakamura, H., Fujita, M., Kono, S. and Tanaka, H.: Temporal Logic Based Fast Verification System Using Cover Expressions, *Proc. Int. Conf. on Very Large Scale Integration '87*, pp. 99-111 (1987).
6) Moszcowski, B. C.: Reasoning about Digital Circuits, STAN-CS-83-970 (1983).
7) 平石，矢嶋：正則集合と表現等価な正則時相論理 RTL，情報処理学会論文誌，Vol. 28, No. 2, pp. 117-123 (1987).
8) Manna, Z. and Pnuili, A.: Verification of Concurrent Programs, Part 1: The Temporal Framework, STAN-CS-81-836 (1981).
9) Wolper, P.: Synthesis of Communicating Process from Temporal Logic Specifications, STAN-CS-82-925 (1982).
10) 杉山，須藤，唐津：VLSI 設計システム，情報処理学会電子装置設計技術研究会資料，7-2 (1980).
11) Grass, W. and Schielow, N.: VERENA: A Program for Automated Verifications of the Refinement of a Register Transfer Description into a Logic Description, *Proc. 7th Int. Symp. on Computer Hardware Description Language and their Applications*, pp. 115-128 (1985).
12) PANAFACOM·U シリーズ ユーザ装置設計手引書 09-HS-0080-1，富士通．
13) Bryant, R. E.: Graph-Based Algorithms for Boolean Funtion Manipulation, *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 667-691 (1986).

(昭和63年6月29日受付)
(平成元年4月11日採録)



中村　宏（正会員）

昭和38年生．昭和60年東京大学工学部電子工学科卒業．昭和62年同大学大学院工学系研究科電気工学専攻修士課程修了．現在同大学院後期博士課程在学中．時相論理，論理設計支援に関する研究に従事．日本学術振興会特別研究員．

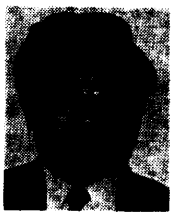


藤田　昌宏（正会員）

昭和31年生．昭和55年東京大学工学部電気工学科卒業．昭和60年同大学院工学系研究科情報工学専門課程博士課程修了．工学博士．同年(株)富士通研究所入社．以来，論理設計CADシステムの研究開発に従事．並列論理プログラミング言語や並列処理技術，特にその論理設計CADに興味を持つ．

河野　真治（正会員）

昭和34年生．昭和59年東京工業大学理学部化学科卒業．平成元年東京大学大学院工学系研究科情報工学専攻博士課程修了．工学博士．同年ソニーコンピュータサイエンス研究所入社．時相論理型言語，並列オブジェクト指向言語に関する研究に従事．



田中　英彦（正会員）

昭和18年生．昭和40年東京大学工学部電子工学科卒業，昭和45年同大学院博士課程修了．工学博士．同年東京大学工学部講師，昭和46年助教授，昭和62年教授．昭和53年～54年ニューヨーク市立大学客員教授，現在に至る．計算機アーキテクチャ，並列推論マシン，知識ベースマシン，オブジェクト指向計算システム，分散処理，CADなどの研究を行っている．「計算機アーキテクチャ」，「VLSIコンピュータI，II」（いずれも共著），「情報通信システム」著．電子情報通信学会，人工知能学会，日本ソフトウェア科学会，IEEE，ACM各会員．