Vol. 25 No. 2 情報処理学会論文誌 Mar. 1984

時相論理によるハードウェア仕様記述と Prolog を用いたゲート回路の検証†

藤田昌宏†† 田中英彦††† 元岡達†††

従来，論理装置設計者は，おもに自然言語を用いて仕様を記述して機能設計を行い，それに基づいて論理設計を行っていた．論理設計以降は，設計支援ソフトウェアもある程度そろっているが，機能設計においては，自然言語を中心に順序線図（タイミングチャート）等の付加情報を加えて記述しているため，設計者間の意思の疎通がむずかしく，階層設計を円滑に行いにくい．設計の検証についても，レジスタ転送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る．そこでわれわれは，仕様記述からゲートによる記述まで一貫して階層設計を支援する検証システムを提案する．一般にシステムは，同期部 (synchronization part) と，演算部 (function part) の二つに分けることができる．ここでは，演算部を入出力の表として記述し，同期部の仕様記述には，時相論理 (temporal logic) を用いる．時相論理は，古典論理に時相演算子を加えたもので，時間軸上の順序関係を記述することができる．本論文では，時相論理による仕様記述を用いた階層設計法，および，ゲートによる設計に対する検証を，論理型プログラミング言語 Prolog を用いて自動的に行うシステムについて述べる．Prolog のもつ自動バックトラック機構や強力なパターン照合能力により，検証プログラムが非常に作成しやすく，かつ，簡単になっており，将来知識工学的手法を取り入れる場合にもつながりがよい．

1. まえがき

近年，計算機技術や，VLSI 技術に代表される素子技術の進歩に伴い，ますます大規模・複雑なシステムを短期間に設計する手法・手段が必要となってきている．また，計算機が社会に対して重要な位置を占めるようになり，その高信頼性がとくに要求されている．

従来，論理装置設計者は，おもに自然言語を用いて仕様を記述し，ブロックダイアグラム，状態遷移図，順序線図（タイミングチャート），あるいは，レジスタ転送レベル（RTL）のハードウェア記述言語を用いて，機能設計を行い，これらに基づいて論理設計を行っていた[1]．

論理設計以降は，ハードウェア記述言語を中心として仕様記述を行うため，各設計者間の情報伝達のミスは比較的少なく，設計支援ソフトウェアもある程度そろっている．しかし，機能設計においては，自然言語を中心に順序線図等の付加情報を加えて記述しているため，意思の疎通がむずかしい．とくに問題となるのは，各モジュール間のインタフェース部分であり，通常順序線図として表されているような情報を取り扱えるツールが望まれる．

また，設計の検証に関しては，RTL 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り，もれなくすべての場合について検証を行うことは不可能である．とくに，大規模システムで問題となるユニット間のインタフェースに関するチェックを行うことはむずかしい．このため，ハードウェアの検証に関して，ソフトウェアの検証手法の応用[2]や DDL ベリファイア[3]等が研究されている．前者は，検証を行う際の仕様の与え方がむずかしいこと，および，インタフェース部分の検証を行いにくい等の問題がある．また後者は，比較的大規模なシステムにも応用ができ，インタフェース部分の検証を行うこともできるが，現在のところ，システムの安全性のみしか検証できないこと，DDL[4]による記述に限られるという問題がある．一方，大規模・複雑なシステムの設計には，徐々に設計を詳細化でき，仕事の分担をうまく行うことができる階層設計が向いている．

以上のことから，本論文では，仕様記述からゲートによる記述まで一貫して階層設計を支援する検証システムを提案する．一般に，ハードウェアシステムは，同期部 (synchronization part) と，演算部 (function part) の二つに分けることができる．ここでは，演算部を入出力の表として記述し，同期部の仕様記述には，時相論理 (temporal logic)[5),6)] を用いる．時相論

† Hardware Specification in Temporal Logic and Verification of Gate Level Designs with Prolog by MASAHIRO FUJITA (Information Engineering Course, Graduate School of Engineering, University of Tokyo), HIDEHIKO TANAKA and TOHRU MOTO-OKA (Department of Electrical Engineering, Faculty of Engineering, University of Tokyo).

†† 東京大学大学院工学系研究科情報工学専門課程
††† 東京大学工学部電気工学科

理は，古典論理に時相演算子を加えたもので，時間軸上の順序関係を記述することができる．このため，順序線図と同様に時間関係を表すことができ，ハードウェアのインタフェース条件を仕様として記述することができる．

設計は階層的に行う．一つの設計サイクルは，

① 上位レベルの仕様を満たすようにいくつかのサブモジュールに分け，

② それぞれのサブモジュールに要求される仕様を時相論理で記述，または，RTL やゲートによる設計を行い，

③ それら全体が上位レベルの仕様を満たしているかを検証すること，

となる．

本論文では，時相論理による仕様記述を用いた階層設計法，および，ゲートによる設計に対する検証法について述べる．時相論理をハードウェアの仕様記述に用い人手による検証を行った例[7]が報告されているが，本論文では，ハードウェアモジュールとしてより一般的な仕様記述および，階層設計を考え，論理型プログラミング言語 Prolog[8] を用いて，それらの検証を自動的に行うシステムについて述べる．検証を行う際には，すべての場合について考慮する必要があるが，これは，Prolog のもつ自動バックトラック機構により，簡単に実現できる．Prolog のもつ強力なパターン照合能力により，検証プログラムが非常に作成しやすく，また，簡単になっている．より大きなシステムを検証する際には，ある程度，知識工学的手法を取り入れる必要があるが，Prolog を用いておくとつながりもよい．

2章では，時相論理によるハードウェア仕様記述の手法を示し，3章で，例を用いて本システムにおける階層設計を説明し，4章で，Prolog によるゲート回路の表現法を述べ，5章では，検証アルゴリズムを具体的に説明し，検証例を示す．そして，6章では，検証時間や能力について，検討を行う．

2. 時相論理によるハードウェアの仕様記述

2.1 時相論理[5),6)]

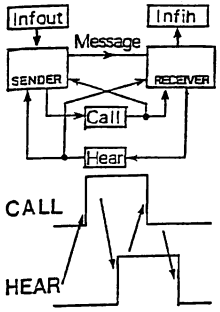
本節では，時相論理について簡単に説明する．詳細については，参考文献を参照されたい．

古典論理では，∧，∨，→，～等の演算子を用いるが，時相論理では，これらに四つの時相演算子を付け加える．システムについて記述する際に，古典論理ではある特定の時間（通常，現在）についてのみしか記述できないのに対し，時相論理では，現在から将来のすべての実行順序について記述することができる．時相演算子には，□(always)，○(next)，▽(sometime)，U(until) の四つがある．最初の三つは，単項演算子で，最後のは2項演算子である．□Fは，現在から先，永久にFが真であることを示す．○Fは，次の時刻にFが真であることを示す．また，▽Fは，現在から考えて，将来の少なくとも1時刻にFが真であることを示す．そして，F1 U F2 は，F2 が真となるまでは，F1 が真でありつづけるということを示す．

時相演算子を組み合わせて使うこともでき，たとえば，A→▽□Bは，「もしAが真なら，そのときからずっとBが真でありつづけるような時刻が存在する」を表し，A→□▽Bは，「もしAが真なら，将来のどの時刻においても，Bがいつかは真となる．つまり，無限回Bが真となりうる」を表す．システムの性質は，安全性 (safeness) と生存性 (liveness) に分けることができる．安全性は，どのようなときにも満足しなければならない条件を表し，一般に，「悪いことが起こらない」を示すことが多い．これは，□演算子を用いて，A→□BやA→▽□Bの形で表せる．一方，生存性は，いつかは望ましい結果に到達することや，いつかは処理が終了すること等を表す．これは，▽演算子を用いて，A→▽BやA→□▽Bの形で表現できる．このように時相論理を用いて，システムのさまざまな性質を表現することができる．

2.2 順序線図の記述[7)]

時相論理を用いて，順序線図で表されるような各モジュール間の時間関係を表現することができる．信号の立ち上り↑Pは，↑P＝～P∧○Pで，立ち下り↓Pは，↓P＝P∧○～Pで表現できる．また，モジュール間の同期条件も，たとえばハンドシェイクで考えると，**図1**のように表現できる．同図(a)では，Hear が0であることを確認して Call が立ち上がり，Call が立ち上がると Hear が立ち上がり，次に Call が立ち下がり，最後に Hear が立ち下がって，1サイクルが終了することを表している．この関係を時相論理で表現すると同図(b)のようになる．□(～Hear→▽Call) で Hear が0なら，いつか Call が1になることを示す．また，U演算子は，たとえば，□(～Call→(～Call U ～Hear)) で，いったん Call が0になると，Hear が0になるまでは，0でありつづけること，つまり，いわゆる「ばたつき」がないことを表し



□(～ Hear → ▽ Call)
□(Hear → ▽ ～Call)
□(～Call → ～Call U ～Hear)
□(Call → Call U Hear)
Peter (Sender)

□(Call → ▽ Hear)
□(～Call → ▽ ～Hear)
□(～Hear → ～Hear U Call)
□(Hear → Hear U ～Call)
Paul (Receiver)

(a) Timing chart (b) Temporal logic

図 1 ハンドシェイクによるデータ転送
Fig. 1 Data transfer using handshaking sequences.

ている．このようにU演算子を用いて，Call と Hear がレジスタの性質をもつことを表現できる．

2.3 ハードウェアの仕様記述

一般に，ハードウェアシステムは，同期部 (synchronization part) と，演算部 (function part) の二つに分けて記述することができる．同期部は，各モジュール間のデータ転送等の同期を取る部分で，個々はそれほど複雑ではないが，人間にとって混乱しやすく，設計しにくい部分であり，そのため誤設計が多い．また，演算部は，ALU 等のように具体的に計算を行う部分で，すでに設計されたモジュールを使うことも多く，タイミングのことをあまり考えなくてもよいこともあって，比較的誤設計が少ない．

本論文では，演算部は入力と出力の間の関係表として記述し，同期部を時相論理で記述することによってハードウェアの仕様記述を行う場合を想定し，同期部の検証について考察することにする．演算部の設計と検証については，ソフトウェアでの手法の応用等が考えられるが，ここではふれない．

3. 階層設計支援システム

時相論理による仕様記述から，DDL やゲートによる設計まで，1章で述べた設計サイクルに従って，順次階層的に設計を進める場合を想定する．

ハンドシェイクによるデータ転送を，階層設計の例として図2に示す．送信側を Sender，受信側を Receiver とし，レジスタ Infout の内容をレジスタ Infin にターミナル Message を通し，Call と Hear とによるハンドシェイクを行って送る．図2からわかるように，1段目の仕様は，「適当な初期条件が成立すると，いつかはデータが正しく転送される」，および，「その初期条件がいつでも成立しうる」からなる．次に，2段目の仕様として，具体的にハンドシェイク

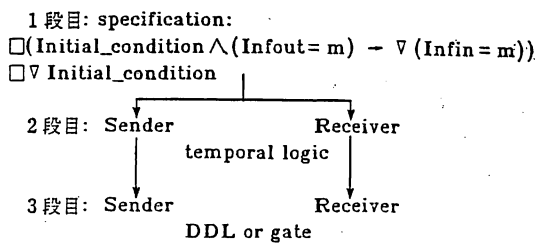


図 2 階層設計例
Fig. 2 Hierarchical design example.

によって送ることにし，図1のような知識を用いて，Sender, Receiver のおのおのに要求される仕様を時相論理で記述する．3段目では，ゲート（あるいは DDL）を用いて設計する．

検証は，2段目の Sender と Receiver の仕様を合わせたものが，1段目の仕様と整合しているかどうか，3段目の Receiver のゲートによる設計が2段目の Sender の時相論理による仕様を満たしているかどうか等を調べる．なお，DDL や時相論理で記述されたものに対する検証については，ここでは述べない．

4. Prolog によるゲート回路の表現

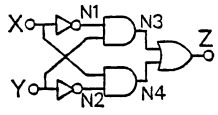
4.1 基本ゲート

本章では，制御を明確に表現することができる Prolog/KR[9] を用いた，ゲート回路の表現法について述べる．以下，組合せ回路には遅延がないとする．AND, NOT 等の基本ゲートは，入力と出力の関係を真理値表に対応する形で図3のように Prolog で表現される．同図(a)は，Prolog/KR による記述で，(b)は，通常の Prolog による記述であり，両者は対応している．Prolog/KR では，ASSERT を使ってプレディケイトを表現する．ASSERT の次にくる項がゴールで，それ以降の項が条件となる．0，1の2値によって表現してあり，一つの ASSERT のなかで最後の項が出力で，それ以外の項が入力である．基本ゲートを組み合わせた回路の表現例として，exclusive-OR の場合を図4に示す．(a)の回路を Prolog に直したものが，(b)，(c)で，直接表現すると(d)のように

```
(ASSERT (FAND 0 0 0))   fand (0 0 0).
(ASSERT (FAND 0 1 0))   fand (0 1 0).
(ASSERT (FAND 1 0 0))   fand (1 0 0).
(ASSERT (FAND 1 1 1))   fand (1 1 1).
(ASSERT (FNOT 0 1))     fnot (0 1).
(ASSERT (FNOT 1 0))     fnot (1 0).
```

(a) Prolog/KR (b) Popular Prolog

図 3 Prolog によるゲートの記述
Fig. 3 Gate descriptions in Prolog.



(a) Circuit

```
(ASSERT (FEOR *X *Y *Z)
 (FNOT *X *N1) (FNOT *Y *N2)
 (FAND *N1 *Y *N3)
 (FAND *N2 *X *N4)
 (FOR *N3 *N4 *Z))
```

(b) Prolog/KR

```
feor(X,Y,Z):-fnot(X,N1),fnot(Y,N2),
      fand(N1,Y,N3),fand(N2,X,N4),
      for(N3,N4,Z).
```

(c) Popular Prolog

```
(ASSERT (FEOR 0 0 0))
(ASSERT (FEOR 0 1 1))
(ASSERT (FEOR 1 0 1))
(ASSERT (FEOR 1 1 0))
```

(d) Prolog/KR

図 4 Exclusive OR ゲートの記述
Fig. 4 Exclusive OR gate descriptions.

なる．このように回路中の配線の表現は，同じ変数（＊で始まるもの）名を用いることによって，つながっている各素子の端子を同じ値にすることで可能である．

4.2 フリップフロップの表現

フリップフロップは，現在の内部状態と，次の内部状態の関係として表現される．Dフリップフロップの Prolog/KR による記述を図5に示す．表のうち，最初の項がD入力であり，次の2項が現在の内部状態Qと～Q，その次の2項が次の内部状態 Q_n と $\sim Q_n$，そして最後の項がクロックである．したがって最初の ASSERT は，もし現在の内部状態がリセット（Q＝0，～Q＝1）で，かつクロックが0（クロックがこない）なら，次の内部状態はD入力に関係なくリセット（Q＝0，～Q＝1）となることを示している．このように，クロックの存在を考えた内部状態の変化が表現される．同じようにして，カウンタ等の内部状態をもつものを記述できる．

```
(ASSERT (FDFF * 0 1 0 1 0))
(ASSERT (FDFF 1 0 1 1 0 1))
(ASSERT (FDFF 0 0 1 0 1 1))
(ASSERT (FDFF * 1 0 1 0 0))
(ASSERT (FDFF 1 1 0 1 0 1))
(ASSERT (FDFF 0 1 0 0 1 1))
```

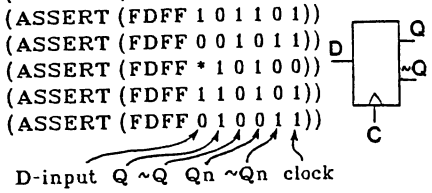
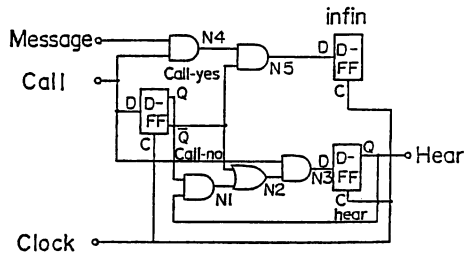


図 5 Dフリップフロップの記述
Fig. 5 D flip-flop descriptions.

4.3 モジュールの表現

前節までに示した手法を用いて，順序回路は最終的に，現在の値と次の値の表として Prolog で記述される．図1で示したデータ転送システムの受信側である Receiver をゲートで設計した例を図6（a）に，Prolog に直したものを（b）に示す．4.1 節で述べたように，回路中の配線は，同じ変数名によっており，回路図中のネット名と一致させてある．図中，＊の後に@があるものが次の時刻での値で，ないものが現在の値である．モジュール全体に対する入出力端子と，内部で使用するフリップフロップの状態を Prolog での記述における外部への引数とする．フリップフロップが現在の値と次の値を直接結びつける役割をはたす．このようなモジュール全体の記述は，回路の接続のみ与えられれば，自動的に合成することができる．



(a) Circuit

```
(ASSERT (RECEIVER
          (*MESSAGE *CALL *HEAR *INFIN *C_Y *C_N)
          (*@MESSAGE *@CALL *@HEAR *@INFIN *@C_Y *@C_N)
          *CL)
     (FAND *MESSAGE *CALL *N4) (FAND *N4 *C_N *N5)
     (FDFF *N5 *INFIN *~INFIN *@INFIN *@~INFIN *CL)
     (FDFF *CALL *C_Y *C_N *@C_Y *@C_N *CL) (FAND *C_Y *HEAR *N1)
     (FOR *N1 *C_N *N2) (FAND *CALL *N2 *N3)
     (FDFF *N3 *HEAR *~HEAR *@HEAR *@~HEAR *CL))
```

(b) Prolog/KR

図 6 Receiver のゲートによる設計例
Fig. 6 A gate level design of Receiver.

5. ゲートレベルの検証

5.1 順方向推論と逆方向推論

3.3 節で示した，デート転送システム中に現れる時相論理の記述を参考として，

□(A→▽B)，□(A→□B)，□(A→○B)，
□(A→□▽B)，□(A→▽□B)，
□(A→B U C)

(A，B，C には，時相演算子はないとする)

の六つについて，時間に関して，順方向推論で検証を行うプログラムを Prolog/KR で作成し，最初の三つについては，逆方向推論で検証するプログラムも作成した．これら6種類の論理で基本的に順序線図の情報を表せると考える．

検証ステップは，次の二つに分かれる．

① まず，回路の接続情報から，4.4 節で示したような Prolog の記述を得る．

② 背理法により，順方向推論か逆方向推論のいずれかを用いて検証する．

5.2 検証プログラム

本節では，順方向推論と逆方向推論おのおのについて，□(A→▽B) を例にとって，プログラムを具体的に説明する．どちらも，すべての場合について，(A→▽B) が成り立つことを検証することによって，□(A→▽B) を検証する．背理法で検証するために，まず否定を取り，

～(A→▽B)＝(A∧～▽B)＝(A∧□～B)

(∵ ～▽＝□～ 公理より)

を得る．次に，A∧□～Bを満たすパスがないかを調べ，もしあれば反例として印刷する．推論を進めていって，前の状態と同じになったか (ループになったか) 否かのチェックは，すべてのフリップフロップ等の内部状態をもつ素子の状態が等しくなったかどうかで行う．

- 順方向推論

① 初期状態 INIT から始め，それがA∧～Bを満たしているかを調べる．満たしていれば，次の状態Nを得る．

② ループになるまで以下を繰り返す．

NがBを満たしているか調べる．もし満たしていれば，A∧□～Bではありえないので，強制的にバックトラックをかけ別のパスを調べる．もうパスが他になければ検証を終了する．Bを満たしていなければ，その次の状態を求める．ループになったら③へ行く．

③ このパスは A∧□～B を満たし，反例なので

```
(ASSERT (VER *INIT)
      (LOGICA *INIT) (LOGIC~B *INIT) (STTRAN *INIT *N) (LOGIC~B *N)
      (VER1 (*INIT) *N))
(ASSERT (VER1 *H *P)
      (IF (STEQU *H *P)
          (AND (PRINT (*H "type<>" *P)) (FALSE))
          (AND (STTRAN *P *N) (LOGIC~B *N) (VER1 (*P . *H) *N))))
```

(a) Forward reasoning program

```
(ASSERT (VERBK *INIT)
      (LOGIC~B *INIT) (STTRAN *PAST *INIT) (LOGIC~B *PAST)
      (VERBK1 (*INIT) *PAST))
(ASSERT (VERBK1 *H *P)
      (IF (STEQU *H *P)
          (IF (LOGICA *P)
              (AND (PRINT (*H "typebk<>" *P)) (FALSE))
              (AND (STTRAN *PAST *P)
                   (LOGIC~B *PAST)
                   (VERBK2 (*P . *H) *PAST))))
      (STTRAN *PAST *P) (LOGIC~B *PAST) (VERBK1 (*P . *H) *PAST))
(ASSERT (VERBK2 *H *P)
      (IF (STEQU *H *P)
          (FALSE)
          (IF (LOGICA *P)
              (AND (PRINT (*H "typebk<>" *P)) (FALSE))
              (AND (STTRAN *PAST *P)
                   (LOGIC~B *PAST)
                   (VERBK2 (*P . *H) *PAST)))))
```

(b) Backward reasoning program

図 7 □(A→▽B) に対する検証プログラム
Fig. 7 Verifier program for □ (A→▽B).

そのパスを印刷し，また別のパスを調べるために強制的にバックトラック (Prolog/KR では，(FALSE)) をかけ，②にもどる．

以上を Prolog/KR で記述したものが図7(a)である．(STTRAN ＊P ＊N) は，現在の状態 (＊P) と次の状態 (＊N) の関係を表したものであり，具体的には，たとえば図6(b)の RECEIVER が対応する．(STEQU ＊H ＊P) は，現在の状態がそのパスの過去の状態 (＊H) と等しいか調べるシステムプログラムであり，パターン照合機構により簡潔に記述されている．また，(LOGICA ＊INIT) は，初期状態 INIT が条件Aを満たしているか調べるもので，(LOGIC~B ＊P) は，状態Nが条件～Bを満たしているか調べるものであり，ユーザが別に与える (5.3節参照)．このように，Prolog のもつ自動バックトラック機構と，強力なパターン照合能力により，プログラムが非常に簡単になっている．

• 逆方向推論

この場合は時間を溯っていくので，A∧□～Bを調べるのに，～Bを満たす状態から始める．詳細は省略するが，～Bを満たしながらループとなり，そのループに入る状態でAを満たすかどうかをチェックする．Prolog/KR によるプログラムを図7(b)に示す．一般に，Aが true のとき，つまり▽Bを検証しようとするときは，逆方向推論のほうが速く処理できるが，A→▽Bのときは，どちらが速いともいえない．

以上のような方法で，前に述べた六つのアサーションを処理するプログラムを作成した．より複雑なアサーションに対する処理プログラムも同様に作成可能であるが，時相論理の決定手続きをもとに作成したほうが取り扱いやすい．

5.3 検 証 例

本節では，4.3 節に示した Receiver のゲートレベルの設計に対して，□(Call → ▽Hear) の検証を行った例を図8に示す．図6(b)の Prolog の記述から現在の状態と次の状態の関係が得られ，これを前節のプログラム中の (STTRAN ＊P ＊N) として用いる．Call → ▽ Hear の否定をとると，Call∧□～Hear が得られる．したがって，前節のプログラムの条件Aは Call=1 となり，条件～B は Hear=1 となるため，それぞれ図のように LOGICA, LOGIC~B の対応する変数をセットする．

(a)，(b)は，順方向推論によるものであり，(a)のように初期条件として，Call=1 のみでは反例があることがわかる．そこで(b)のようにフリップフロップQ3のリセット (Call-No=1) を初期条件に加えると反例がなくなることがわかる．また，(c)は，(b)を逆方向推論で処理している．

6. 考察・検討

6.1 順方向推論と逆方向推論の差

一般に，初期状態における条件 (たとえば，□(A→

```
(ASSERT (STTRAN *P *N) (RECEIVER *P *N 1))
                          Call=1
(ASSERT (LOGICA (*MESSAGE 1 *HEAR *INFIN *C_Y *C_N)))
(ASSERT (LOGIC~B (*MESSAGE *CALL 0 *INFIN *C_Y *C_N)))
          Call=1                Hear=0
:(ver (*message 1 *hear *infin *cy *cn))
(((0 1 0 0 1 0)) "type<>" (*@MESSAGE_0054 *@CALL_0054 0 0 1 0))
(((0 1 0 0 1 0) (0 1 0 1 1 0)) "type<>" (*@MESSAGE_0102 *@CALL_0102 0 0 1 0))
.....
```

(a) Forward reasoning

```
                          ~CY  CN
:(ver (*message 1 *hear *infin 0 1))
nil
```

(b) Forward reasoning with ~CY∧CN

```
                      Hear=0
:(verbk (*message *call 0 *infin *cy *cn))
(((0 0 0 0 1 0) (0 0 0 0 0 1) (*MESSAGE_0525 *CALL_0525 0 0 0 1))
 "typebk<>" (0 1 0 0 1 0))
(((0 0 0 0 1 0) (0 0 0 0 0 1) (*MESSAGE_0525 *CALL_0525 0 0 0 1))
 "typebk<>" (0 1 0 1 1 0))
.....
```

(c) Backward reasoning

図8 Receiver に対する □(Call → ▽ Hear) の検証例
Fig. 8 Verification example of □(Call → ▽ Hear) on Receiver.

▽B）におけるA）がないとき（true のとき）には，逆方向推論のほうが順方向推論よりも速い．これは，逆方向推論は矛盾を起こした状態から推論を進めていくので，設計にもよるが一般に調べる範囲が順方向推論よりも小さくてすむからである．しかし，初期条件が指定されているときは，逆方向推論も初期条件を満たすかを調べる必要があり，条件しだいでどちらが速いともいえない．システム側では，両方サポートしユーザが使い分けるのがよいといえる．

6.2 より大きな回路への対応

図8に示した検証例では，大型計算機 M-280H で，だいたい 0.5〜1 秒程度かかっている．現在の手法では，回路規模に対して，単純には指数的に処理時間が増大すると考えられる．したがって回路規模が大きくなると，発見的な手法も取り入れる必要があるが，たとえば，過去に検証したことがらを定理として用いる，あるいは，途中必ず通過しなければならない状態を指定してやる等の方法で，処理時間の増大はかなりおさえられると考える．途中の状態が指定されれば，そこの前と後で二つに区切って処理してやれば，扱う場合の数をかなり減らすことができる．また，より大きなモジュール（たとえば，adder, counter）も基本素子として入出力の関係を Prolog で記述してやることにより，かなり高いレベルまで扱え，また処理時間の減少も期待できる．

7. むすび

時相論理による同期部の仕様記述を中心とした階層設計および，Prolog/KR を用いたゲートレベルの検証について述べた．

Prolog の自動バックトラック機構と強力なパターン照合能力により処理プログラムはきわめて簡単になっている．また，処理時間についても推論のパスをユーザがある程度指定してやることにより，かなりおさえられると期待できる．

時相論理で記述されたものや DDL で記述されたものに対する検証と組み合わせて用いることにより，システムの仕様記述からゲート回路まで，一貫して扱うことができる検証システムを作成することができる．

参考文献

1) 樹下行三(編)：論理装置の CAD，情報処理学会叢書 5，オーム社，東京 (1981).
2) Hill, F. J., Chu, Y., Dietmeyer, D. L. and Siewiorek, D.: Introducing Computer Hardware Description Languages, *IEEE Comput.*, Vol. 7, No. 12, pp. 18-67 (1974).
3) 上原，丸山，斎藤，川戸：DDL Verifier, 5th Computer Hardware Description Language and Their Applications, pp. 51-61, North-Holland (1981).
4) Duley, J. R. and Dietmeyer, D. L.: A Digital System Design Language (DDL), *IEEE Trans. Comput.*, Vol. C-17, No. 9, pp. 850-861 (1968).
5) Wolper, P.: Temporal Logic Can Be More Expressive, 22nd Annual Symposium on Foundation of Computer Science (1981).
6) Pnueli, A., Shelah, S. and Stavi, J.: On the Temporal Analysis of Fairness, 7th Annual Symposium on Principle of Programming Language (1980).
7) Bochmann, G. V.: Hardware Specification with Temporal Logic, *IEEE Trans. Comput.*, Vol. C-31, No. 3, pp. 223-231 (1982).
8) Clocksin, W. F. and Mellish, C. S.: *Programming in Prolog*, Springer-Verlag, New York (1981).
9) 中島：Prolog/KR User's Manual, 東大工学部テクニカルレポート METR 82-4 (1982).

(昭和58年6月 9 日受付)
(昭和58年7月19日採録)