テンポラル・ロジックとProlog による論理設計の検証

藤田　昌宏、　田中　英彦、　元岡　達
（東京大学　工学部）

1.　はじめに

システムの大規模化・複雑化により、ハードウェア（システム）設計の初期における、設計検証が重要になってきている［1］。　ここでは、機能レベル以上のレベルから設計を支援するCADシステムを提案する。　特徴として、

①階層設計を支援する。
②仕様記述にテンポラルロジックを用いる。
③上位レベルと下位レベル間の検証を行なう。
④比較的小さなモジュールに対しては、自動合成を行なう。
⑤Prolog ／KRを用いて作成する。

が、あげられる。

2.　テンポラルロジック

ここでは、テンポラルロジックの各オペレータについて、ごく簡単に説明する。詳しいことについては、参考文献＜2、5、10＞参照。

テンポラルロジックは、古典論理に、テンポラルオペレータを付け加えたものであり、各オペレータは、

・□P　always：　いつもP
・◇P　somtime ：　いつかP
・○P　next：　次の時刻にP
・P　U　Q　until ：　QになるまでPでありつづける

を表わす。　古典論理では、今の時刻に対してassertion を与えられるのに対し、テンポラルロジックでは、時間のシーケンスに対してassertion を与えることができ、通常、ハードウェア設計で用いられるタイミングチャートの時間関係を表わすことができる。

テンポラルオペレータをつづけて用いると、例えば、
B⊃◇□A　は、『Bならば、ある時からずっとAとなる。』を表わし、　B⊃□◇A　は、『Bならば、いつでもAがおこる、つまり、無限回がおこる。』を表わす。

□を用いて、safeness property を、また、◇を用いて、liveness property を表わすことができる。　また、until オペレータは、storage の表現に有効であり、例えば、フリップフロップで、

out ＝1　⊃　out ＝1　U　clear ＝1

によって、一旦、out ＝1になると、clear が1になるまで out＝1でありつづけることを表わせる。

3.　CADシステムの構成

全体の設計の流れを、ハンドシェィクによるデータ転送を例にとって、図1に示す。設計は階層的に行なわれ、1つの設計サイクルは、

① テンポラルロジックによる仕様記述
② サブモジュールにモジュール分け、または、DDL［3］やprimitive gates による設計
③ ②のサブモジュールや設計が上位レベルの仕様を満たしているかの検証

と、なる。

図1の例では、　top-level で仕様として、適当なinitial-condition が成り立てば、データが正しく伝送されること、及び、そのinitial-condition がいつでも成立し得ることを記述する。　2nd-levelの設計を行なう時にハンドシェイクについての知識を用いて、テンポラルロジックによる仕様（設計）を得る。　3rd-levelでは、DDLによる設計を行なっており、図11に示す。また、もし必要なら、4th-levelの設計として、gateによる設計を行なう。

このサイクルは、全てのモジュールがDDLやprimitive gates で記述される（自動合成も含む［4］。）まで続けられる。　検証は、アサーションとその答え、という形で行なわれる。すなわち、1つ上のレベルの仕様（設計）が、1つ下のレベルに対するアサーションとなる（図2）。

4.　Prolog ／KR［6］、［8］

Prolog ／KR［6］は、中嶋氏により開発されたものであり、deterministic なcontrol が付け加えられている。また、シンタックスは、UTILISP［7］に従う。Prolog ／KRを用いて、gateの記述を図3に示す。簡単のため、0、1の2値とし、かつ、組み合せ回路には遅延がないとする。図に示すように、table の形で入力・出力の関係を記述していく。また、“＊”で始まるsymbolは変数を表わし、配線は、ネットに同じ変数を用いて、同じ値にさせることにより行なう。

1つ解が求まっても、強制的にbacktrack （false ）をかけることによって、特定の条件を満たすような解を全て求めることができ、verificationのプログラムが作りやすくなっている。

現在の状態の値と、次の状態の値は、D-flipflop （Prolog ／KRでは、FDFF）のtable によって互いに結びつけられている。table の第一項は、D－入力を示し、次の二項は、現在の内部状態を示し、その次の二項は、次の内部状態を示し、最後の項は、クロックの存在を示している。すなわち、クロックが1でなければ、内部状態は変化しないように定義されている。

registerの値は、実際には、（＊value　＊x ）　の形で扱われる。＊value は、そのregisterの実際の値を示し、＊x は、付加情報で判例を分かりやすくするためのものである。

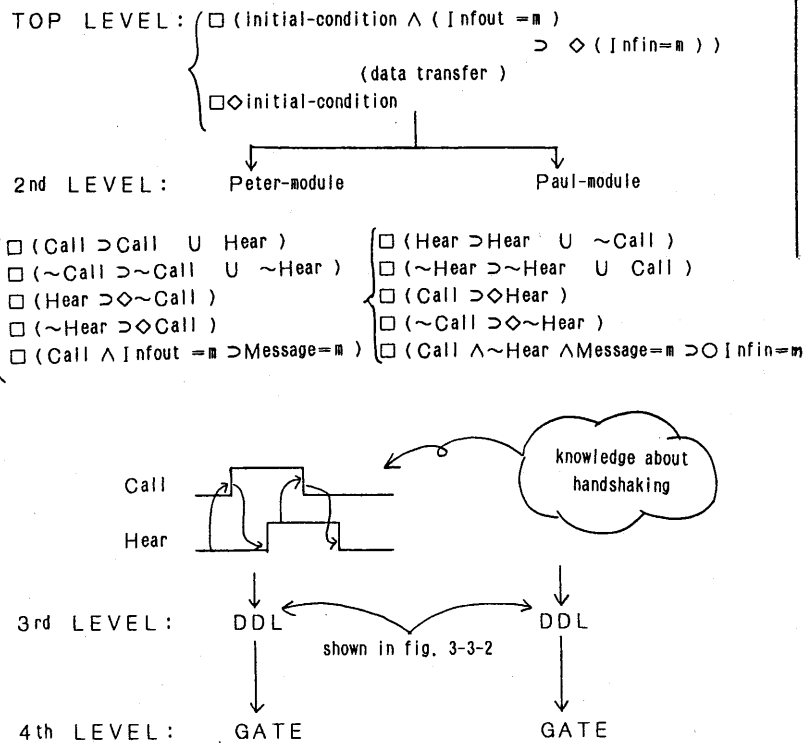

図１　ハンドシェイクによるデータ転送

5.　Prolog を用いた、DDL、及び、ゲートレベルのベリフィケーション

5－1　6つのタイプのアサーション

次の6つのタイプのアサーションについて、実験的に検証を行なうプログラムを作成した。

□(A ⊃ ○B)、□(A ⊃ ◇B)、
□(A ⊃ □B)、
□(A ⊃ □◇B)、□(A ⊃ ◇□B)、
□(A ⊃ B U C)

これらを用いて様々なタイミングチャートを表わすことができる。

また、もつと複雑なタイプのアサーションにも拡張することはできるが、6章のtemporal logicのdecision procedureを用いた方が扱いやすい。（6章参照）

5－2　forward reasoning とbackward reasoning

assertion は、forward reasoning でも、backward reasoningでもどちらでも処理できる。

①DDL記述やprimitive gates による記述をProlog／KRのstate transition tableに直す。つまり、現在の状態と次の状態との関係に直す。

②①のtable を用いてシステム全体の状態遷移を可能にする。

③背理法で証明する。

以上を図に示すと、図6のようになる。次に例を用いてforward reasoning とbackward reasoningについて、具体的に説明する。

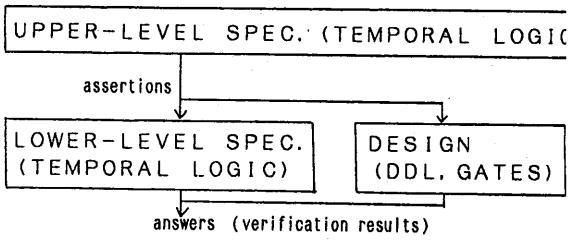

図２　設計検証サイクル

```
(ASSERT (FAND 0 0 0))
(ASSERT (FAND 0 1 0))
(ASSERT (FAND 1 0 0))
(ASSERT (FAND 1 1 1))
(ASSERT (FOR 0 0 0))
(ASSERT (FOR 0 1 1))
(ASSERT (FOR 1 0 1))
(ASSERT (FOR 1 1 1))
(ASSERT (FNOT 0 1))
(ASSERT (FNOT 1 0))
(ASSERT (FDFF 1 0 1 1 0 1))
(ASSERT (FDFF 0 1 0 1 0 0))
(ASSERT (FDFF 0 1 0 0 1 1))
(ASSERT (FDFF 0 0 1 0 1 *))
(ASSERT (FDFF 1 1 0 1 0 *))
```

D入力　Q　～Q　Qn　～Qn　clock

図３　Prolog ／KRによる基本ゲートの記述

```
(ASSERT (VERI11 *0 *1) (STTRAN *0 *1) (VERI111 *0 *1))
(ASSERT (VERI111 *B *P) (STTRAN *P *A) (IF (LOGICB *P) (FALSE))
    (IF (STEQU *B *P) (AND (PRINT (*B TYPE<> *P)) (FALSE))) (VERI111 (*P *B) *A))
```

図4　forward reasoning による検証

```
(ASSERT (VERIBK11 *0 *1) (STTRAN *1 *0) (VERIBK1 *0 *1))
(ASSERT (VERIBK1 *H *P) (STTRAN *PAST *P) (IF (LOGICB *P) (FALSE))
    (IF (STEQU *H *P)
        (IF (LOGICA *P) (AND (PRINT (*H TYPE<>BK *P)) (FALSE)) (VERIBK2 (*P *H) *PAST)))
    (VERIBK1 (*P *H) *PAST))
(ASSERT (VERIBK2 *H *P) (STTRAN *PAST *P) (IF (LOGICB *P) (FALSE))
    (IF (LOGICA *P) (AND (PRINT (*H TYPE<>BK *P)) (FALSE)))
    (IF (STEQU *H *P) (FALSE) (VERIBK2 (*P *H) *PAST)))
```

図5　backward reasoningによる検証

＊forward reasoning

初期状態から始め、ループか、エラーが起こるまで次の状態を求めていくことを繰り返す。もしループか、エラーが起こったら、強制的にbacktrack をかけ、全ての状態遷移について調べる。

<例>　A　⊃　◇B

背理法で証明するため、まず否定をとって、

　～（A　⊃　◇B）＝A∧□～B

を得る。そこで、Aから始めループか、エラーが起きるまで以下を繰り返す。

①次の状態Nを得る。

②もしNがBを満たせば、このパスはOKなので強制的にbackyrack をかけ、別のパスを調べる。

もしループになり、ループ中の全ての状態でBが満たされないなら、このパスは反例なので、印刷し、別のパスを調べる。

＊backward reasoning

<例>　A　⊃　◇B　－－－>　A∧□～B（否定）

～Bから始め、ループになるまで以下を繰り返す。

1つ前の状態の1つをProlog のプログラムから得、Bを満たすかどうか調べる。もしBを満たせば、このパスはOKなので強制的にbacktrack をかけ別のパスを調べる。

ループになったら、ループに入る前の最後の状態がAを満たすかどうか調べる。もし満たせば、A∧□～Bが成り立つことになり、反例となる。もし満たさなければ、今度はループに入る以前の状態でAが成り立っていないか調べる。すなわち、ループになるまで　A∧～Bが成り立たないか調べる。もしあれば、エラーであり、なければOKである。

Prolog ／KRによるプログラムを図4、5に示す。

一般に、Aがないとき、つまり、◇Bを証明する時には、backward reasoningの方が速いが、そうでない時には、どちらがよいとも言えない。

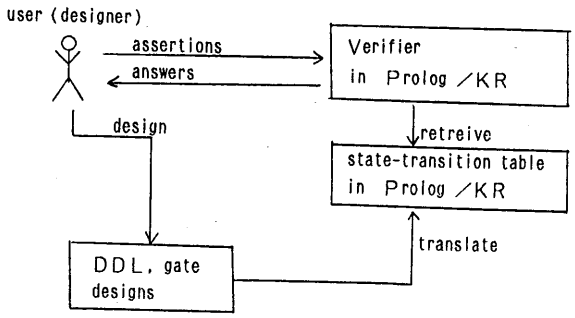

図6　検証システム

5－3　ゲートレベルのベリフィケーション

data transfer systemのPaul-moduleについて、ベリフィケーション例を示す。図7（a）が、ゲートでのインプリメンテーション例であり、同図（b）がそのProlog ／KRでの表現である。ここで、組み合せ回路にはディレイがないとする。図7（b）中の変数のうち、“@”で始まるものは次の状態における各変数の値であり、その以外のものは今の状態における変数の値である。

これで、現在の状態と次の状態の関係がProlog ／KRの記述として得られた。次にPaul-moduleのspecification をassertion として用いた、ベリフィケーション例を示す。

＊　Call　⊃　◇Hear　（図11）

～（Call　⊃　◇Hear）＝（Call ∧□～Hear）だから、Call ＝1で始める。すると、NIL（解がない）を得る。これは、～（Call　⊃　◇Hear）が満たされないことを示し、従って、（Call　⊃　◇Hear）が証明された。

ここで、初期状態として、Call-Yes∧～Call-No であることに注意。もしそうでないと、反例が存在する。

```
(ASSERT (PAULC ((*MESSAGE *X1) (*CALL *X2) (*HEAR *X3) (*INFIN *X4) *CALL_YES *CALL_NO)
               ((*@MESSAGE *X@1)
                (*@CALL *X@2)
                (*@HEAR *X@3)
                (*@INFIN *X@4)
                *@CALL_YES
                *@CALL_NO)
               *CL)
      (FAND *MESSAGE *CALL *N4) (FAND *N4 *CALL_NO *N5)
      (FDFF *N5 *INFIN *~INFIN *@INFIN *@~INFIN *CL)
      (FDFF *CALL *CALL_YES *CALL_NO *@CALL_YES *@CALL_NO *CL)
      (FAND *CALL_YES *HEAR *N1) (FOR *N1 *CALL_NO *N2) (FAND *CALL *N2 *N3)
      (FDFF *N3 *HEAR *~HEAR *@HEAR *@~HEAR *CL))

(ASSERT (LOGICB (*M *CALL *HEAR *INFIN *CALL_YES *CALL_NO)) (= *HEAR (*V *X))
      (EQ *V 1))
:(VERI11 (*M (1 I) (0 I) *INFIN 0 1) *1)
:NIL
:
```

Call　～Hear　～Call-Yes　Call-No　□（Call ⊃ ◇Hear の検証例）

図７（b）　（a）のProlog／KRによる記述
及び、検証例

5－4　DDLレベルのベリフィケーション

ゲートレベルの時と同じように、まず、現在の状態と次の状態の関係を表わすProlog のプログラムを作る。これは、DDLの場合、forward reasoning の方はすぐに得られ、backward reasoningの方は、DDL translator［9］を用いるとすぐに得られる。図11のDDLの記述をProlog に直したものを図9に示す。ベリフィケーションの方法は、ゲートレベルの時と同じである。

ベリフィケーション例を、

Data-transfer-system by-handshaking＝Peter-module ＋Paul-module

について示す。

・クロックの取り扱いについて

異なるクロックで動作する複数のモジュールからなるシステムの検証は次のようにして行なう。今、Peter-module のクロックをclpeter 、Paul-moduleのクロックをclpaulとすると、　1／n ｜clpaul｜≦｜clpeter ｜≦n ｜clpaul｜の時は、連続してn 回まで続けて片方にのみクロックがくることを許すように処理する。例えば、1／2｜clpaul｜≦｜clpeter ｜≦2｜clpaul｜の時は、Peter、Paul に連続してクロックがくることがないとして、図8のように各モジュールへのクロックの供給を決めればよい。

＊　Hear　⊃　◇～Call

図10は、forward reasoning によるもので、反例が存在し、適当な初期状態が満たされなければ、仕様は満足されないことを表わしている。

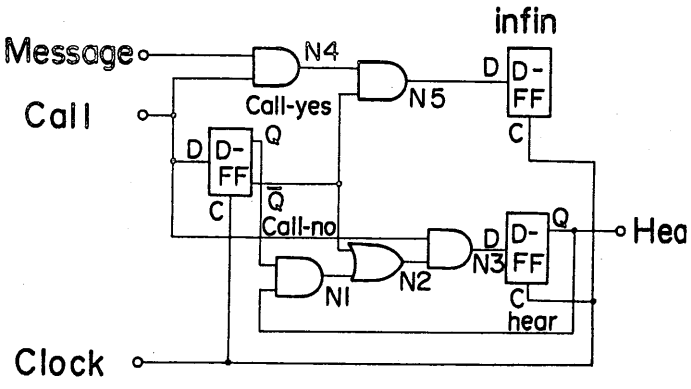

図７（a）　Paul-moduleの回路

```
(ASSERT (CLSTATE S11 1 1))
(ASSERT (CLSTATE S10 1 0))
(ASSERT (CLSTATE S01 0 1))
(ASSERT (CLOCK S11 S10))
(ASSERT (CLOCK S11 S01))
(ASSERT (CLOCK S10 S11))
(ASSERT (CLOCK S10 S01))
(ASSERT (CLOCK S01 S11))
(ASSERT (CLOCK S01 S10))
```

Peterに
クロックを送る
Paul に
クロックを送る

図８　クロックの供給　…Peter-Paul-moduleの片方に続けてクロックを送らないように制御している。

```
(ASSERT (TRAN *@A *B)
    (IF (AND (= *@A (*VAR *C)) (ATOM *C))
        (AND (PRINT (ERROR-IN-REGISTER-TRANSFER *@A *B)) (FALSE))
        (IF (ATOM *B) (= *@A (*B C)) (AND (= *B (*VB *X)) (= *@A (*VB C))))))
(ASSERT (FEQ (*VAR *C) *VAL) (IF (ATOM *C) (= *VAR *VAL) (= (*VAR *C) *VAL D))))
(ASSERT (STTRAN (*PETER *PAUL *CALL *HEAR *INFOUT *INFIN *CLSTATE *M)
             (*@PETER *@PAUL *@CALL *@HEAR *@INFOUT *@INFIN *@CLSTATE *@M))
    (CLOCK *CLSTATE *@CLSTATE) (CLSTATE *CLSTATE *CL1 *CL2)
    (PETER (*PETER *CALL *HEAR *INFOUT *M)
           (*@PETER *@CALL *@HEAR *@INFOUT *@M)
           *CL1)
    (PAUL (*PAUL *CALL *HEAR *INFIN *M) (*@PAUL *@CALL *@HEAR *@INFIN *@M) *CL2)
    (REGUNCHANGED
       (*CALL *HEAR *INFOUT *INFIN *M)
       (*@CALL *@HEAR *@INFOUT *@INFIN *@M))
    (STUNCHANGED (*PETER *PAUL) (*@PETER *@PAUL)))
(ASSERT (PETER (*STATE *CALL *HEAR *INFOUT *M)
             (*@STATE *@CALL *@HEAR *@INFOUT *@M)
             *CL)
    (IF (EQ *CL 0)
        (TRUE)
        (AND (OR (AND (= *STATE HY)
                      (OR (AND (FEQ *HEAR 1) (= *@STATE HY))
                          (AND (FEQ *HEAR 0) (TRAN *@CALL 1) (= *@STATE HN))))
                 (AND (= *STATE HN)
                      (OR (AND (FEQ *HEAR 0) (= *@STATE HN))
                          (AND (FEQ *HEAR 1) (TRAN *@CALL 0) (= *@STATE HY)))))
             (OR (AND (FEQ *CALL 1) (TRAN *@M *INFOUT)) (FEQ *CALL 0))))))
(ASSERT (PAUL (*STATE *CALL *HEAR *INFIN *M) (*@STATE *@CALL *@HEAR *@IFIN *@M) *CL)
    (IF (EQ *CL 0)
        (TRUE)
        (OR (AND (= *STATE CN)
                 (OR (AND (FEQ *CALL 0) (AND (TRAN *@HEAR 0) (= *@STATE CN)))
                     (AND (FEQ *CALL 1) (TRAN *@HEAR 1) (TRAN *@INFIN *M) (= *@STATE CY))))
            (AND (= *STATE CY)
                 (OR (AND (FEQ *CALL 1) (= *@STATE CY))
                     (AND (FEQ *CALL 0) (TRAN *@HEAR 0) (= *@STATE CN))))))))
```

十レジスタ転送の記述

等しいかのチェック

← Perer+Paul の記述

← Peterの記述

← Paul の記述

図9　図11のDDL記述をProlog／KRに変換したもの

Call　Hear

□(Call ⊃ ◇～Hear)　の検証

反例

```
:(VERI11 (*PETER *PAUL (1 I) (1 I) *INFOUT *INFIN *CLSTATE *M) *1)
 (((HY CY (1 U) (1 U) (*VA0_7652 U) (*VA0_7660 U) S11 (*VA0_7652 C))
  ((HY CY (1 U) (1 C) (*VA0_7652 U) (*VA0_7660 C) S10 (*VA0_7652 C))
   (HY CN (1 I) (1 I) (*VA0_7652 *X_7061) *INFIN_6977 S11 (*VA0_7660 *X_7037))))
 TYPE<>
 (HY CY (1 U) (1 U) (*VA0_7652 U) (*VA0_7660 U) S10 (*VA0_7652 C)))
 (((HY CY (1 U) (1 U) (*VA0_8056 U) (*VA0_8064 U) S11 (*VA0_8056 C))
  ((HY CY (1 U) (1 C) (*VA0_8056 U) (*VA0_8064 C) S10 (*VA0_8056 C))
   (HY CN (1 I) (1 I) (*VA0_8056 *X_7061) *INFIN_6977 S11 (*VA0_8064 *X_7037))))
 TYPE<>
 (HY CY (1 U) (1 U) (*VA0_8056 U) (*VA0_8064 U) S10 (*VA0_8056 C)))
```

図10　DDLレベルの検証例

```
<system> Hand-Shake;
  <register> CALL,HEAR,Message;
  <automaton> PETER;
    <register> Infout;
    <state>
    Hear-Yes:  if HEAR=1 then goto Hear-Yes
                         else co-bigin
                                CALL <- 1,
                                goto Hear-No
                              end;
    Hear-No:  if HEAR=0 then goto Hear-No
                        else co-bigin
                                CALL <- 0,
                                goto Hear-Yes
                              end;
    <logic>
              if CALL=1 then Message <- Infout;
  <end> PETER;

  <automaton> PAUL;
    <register> Infin;
    <state>
    Call-No:  if CALL=0 then co-begin
                                HEAR <- 0,
                                goto Call-NO
                             end;
                        else co-begin
                                HEAR <- 1,
                                Infin <- Message,
                                goto Call-Yes
                             end;
    Call-Yes:  if CALL=1 then goto Call-Yes
                         else co-begin
                                HEAR <- 0,
                                goto Call-No
                              end;
  <end> PAUL;
<end> Hand-Shake;
```

図１１　ハンドシェイクのＤＤＬレベルの記述

```
(ASSERT (STTRAN (*PETER *CALL_YES *CALL_NO *CALL *HEAR *INFOUT *INFIN *CLSTATE *M)
                (*@PETER *@CALL_YES *@CALL_NO *@CALL *@HEAR *@INFOUT *@INFIN *@CLSTATE *@M))
    (CLOCK *CLSTATE *@CLSTATE) (CLSTATE *CLSTATE *CL1 *CL2)
    (PETER (*PETER *CALL *HEAR *INFOUT *M)
           (*@PETER *@CALL *@HEAR *@INFOUT *@M)
           *CL1)
    (PAULC (*M *CALL *HEAR *INFIN *CALL_YES *CALL_NO)
           (*@M *@CALL *@HEAR *@INFIN *@CALL_YES *@CALL_NO)
           *CL2)
    (REGUNCHANGED
       (*CALL *HEAR *INFOUT *INFIN *M)
       (*@CALL *@HEAR *@INFOUT *@INFIN *@M))
    (STUNCHANGED (*PETER) (*@PETER)))
```

図１２　混合レベル…Peter（ＤＤＬ）＋Paul （gate）

５－５　混合レベルのベリフィケーション

ＤＤＬ記述と、gateによる設計の両方を持ったモジュールの検証も可能である。

date-trnsferシステムにおいて、Peter-module をＤＤＬで記述し、Paul-moduleをgateで設計したものを考える。クロックの取り扱いは、ＤＤＬの時と同じである。

図１２がそのProlog 記述である。このように、現在と次の関係を容易に得ることができる。ベリフィケーションの方法は今までと同じである。

６．　テンポラルロジック・レベルの検証

６－１　テンポラルロジックのdecision procedure

テンポラルロジックは、tableau method［２］によって、そのsatisfiabilityを判定できる。図１３にdecision procedureで用いるtableau を示す。基本的な考え方は、テンポラルロジックによる仕様記述を『今、要求されること』と、『次の時刻から後に要求されること』に分けて、判断していくということである。例えば、図１３の⑥は、『□Ｆは、今Ｆであり、次の時刻も□Ｆである』ということを示し、○は、『Ｆ１　Ｕ　Ｆ２は、今Ｆ２であるか、又は、今Ｆ１でありかつ次の時刻にＦ１　Ｕ　Ｆ２である』ということを示している。

次に、ベリフィケーション例として、ハンドシェイクのデータ伝送について、このprocedure を用いて証明する。図１４がテンポラルロジックによるPeter、Paul 各モジュールの仕様記述である。（簡単のために、until operatorを除いてある。）

証明しようとすることは、

$$Infout = M \supset \Diamond (Infin == m)$$

であり、背理法で証明する。すなわち、$Infout == m \wedge \Box (Infin \neq m)$ がunsatisfiable であることを示す。

① F1∧F2 → {{F1, F2}}
② ~(F1∨F2) → {{~F1, ~F2}}
③ ~(F1⊃F2) → {{F1, ~F2}}
④ ~~F → {{F}}
⑤ ~○F → {{○~F}}
⑥ □F → {{F, ○□F}}
⑦ ~(F1until F2) → {{~F2, ~F1∨○~(F1until F2)}}
⑧ ~◇F → {{~F}{○~◇F}}
⑨ F1∨F2 → {{F1}{F2}}
⑩ ~(F1∧F2) → {{~F1}{~F2}}
⑪ F1⊃F2 → {{~F1}{F2}}
⑫ ◇F → {{F}{○◇F}}
⑬ (F1until F2) → {{F2}{F1, ○(F1until F2)}}
⑭ ~□F → {{~F}{○~□F}}

図１３　テンポラルロジックのTABLEAU RULE

```
Peter:
  □(Hear ⊃◇~Call )
  □(~Hear ⊃◇Call )
  □(Call ∧Message=x ⊃ Infout =x )
Paul :
  □(Call ⊃◇Hear )
  □(~Call ⊃◇~Hear )
  □(Call ∧Message=x ⊃○( Infin=x ))

specification     Infout =m ⊃◇( Infin=m )
its negation      Infout =m ∧□( Infin≠m )
```

図１４　テンポラルロジックによるデザインの検証

```
・decomposition of  HS, S
  Hear⊃◇~Call , ~Hear⊃◇Call , Call∧ Message=x ⊃Infout =x ,
  Call⊃◇Hear , ~Call⊃◇~Hear , Call∧ Message=x ⊃○( Infin=x ),
  Infout =m⊃□( Infin≠m ), OHS, OS


・case division
Hear  Call
 0     0   |  ○◇Call , Infout =m ⊃Infin≠m          ……①
 0     1   |  ○◇Hear , Message=x ⊃○( Infout =x ),
           |  Infout =x⊃Message=x , Infout =m⊃ Infin≠m   ……②
 1     0   |  ○◇~Hear , Infout =m ⊃Infin≠m         ……③
 1     1   |  ○◇~Call , Message=x ⊃○( Infout =x ),
           |  Infout =x⊃Message=x , Infout =m⊃ Infin≠m   ……④

①:  ◇Call , HS, S
Hear  Call
 0     0   |  ①
 0     1   |  ②
 1     0   |  ○◇Call , ○◇~Hear , Infout =m ⊃Infin≠m   ……③´
 1     1   |  ④

③:  ◇~Hear , HS, S
Hear  Call
 0     0   |  ①
 1     0   |  ②
 0     1   |  ③
 1     1   |  ○◇~Call , ○◇~Hear , Message=x ⊃○( Infout =x ),
           |  Infout =x⊃Message=x , Infout =m ⊃Infin≠m   ……④´

③´:  ◇Call , ◇~Hear , HS, S
Hear  Call
 0     0   |  ①
 0     1   |  ②
 1     0   |  ③´
 1     1   |  ④´
②, ④, ④´ are unsatisfiable  (absurdity )
```

図１５　検証過程

証明過程は、
①ループか、矛盾になるこで②を繰り返す。
②図２２の各記述を図２１を用いて分割し、
　場合分けをし　　て、次の状態を求める。
③全てのパスが矛盾におちいれば、OK。ループになるものがあったら、その脱出可能性を調べ、なければ、エラーとする。
となる。以上の方法での証明過程を図１５に示す。そして、最終的に得られた状態遷移図を図１６に示す。状態、②、④、④´は、矛盾であり、状態、①、③、③´では、ループになっているが、ここからは、必ず脱出する。（◇Call があるから）

図１６は、状態遷移図となっているので、decision procedureを少し変更することにより、ＤＤＬ記述の自動合成を行なうことも可能である。

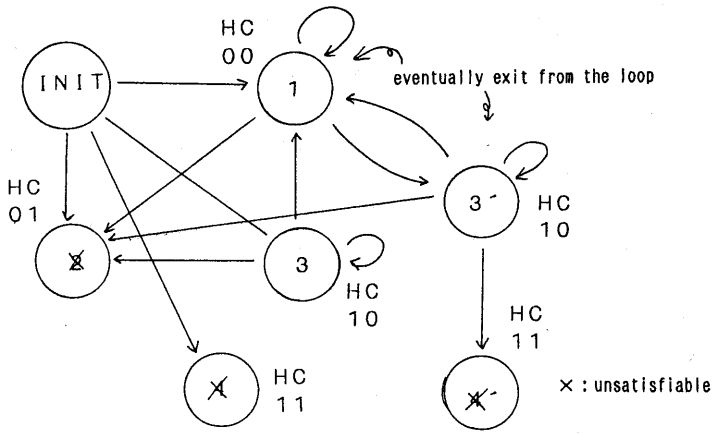

図１６　図１５から得られる状態遷移図

6－2　DDLとテンポラルロジックの混合レベルのベリフィケーション

前節のdecision procedureでは、各オペレータを次の時刻で展開していた。従って、次の時刻の単位をクロックとすると、DDLとテンポラルロジックの共存する設計に対して、検証を行なうことができる。（システム全体が同一クロックで動いているという仮定のもとで考える。）

decision procedureにおける一回の展開をDDL記述における一回の状態遷移に対応させる。つまり、decision procedureによる展開をDDLの記述を参照しながら行なうことにする。これは、constraints を持った展開と考えられる。

このように、テンポラルロジックと、DDL記述の混合した設計に対して検証を行なえることは、階層設計の時、非常に便利であり、先程と同様、少しの変更でDDLの自動合成も行なえる。

7.　最後に

Prolog／KRとテンポラルロジックを用いた、論理設計の検証について、具体例を用いて説明を行なった。ここで示した方法により、仕様記述のレベルから、ゲートレベルまで、かなり統一的に検証を行なっていくことができる。

今後、ここで述べた手法を用いた論理設計検証システムを作成し、評価、及び、高能率化を進めていくとともに、テンポラルロジックを土台［１０］として、よりformalなハードウェア仕様記述について検討していく。また、DDLの自動合成も検討しており、適当に機会に報告したい。

8.　謝辞

本研究を行なう上で、様々な貴重な助言をいただいた、富士通研究所の上原氏、川戸氏、斎藤氏、加藤氏に深く感謝いたします。

＊　参考文献

[1] T.Uehara, F.Maruyama, T.Saito, N.Kawato: "DDL Verifier", 5th Computer Hardware Description Language and their Applications, September 1981.

[2] P.Wolper: "Temporal Logic Can Be More Expressive", 22nd Annual Symposium on Foundation of Computer Science, October 1981.

[3] J.R.Duley, D.L.Dietmeyer: "A Digital System Design Language (DDL)", IEEE Trans. Computer, Vol.C-17, pp850-861 1968.

[4] N. Kawato, et al. : "An Interactive Logic Synthesis System Based Upon AI Technique", 19th Design Automation Conference, June 1982.

[5] A.Pnueli, S.Shelah, J.Stavi: "On the Temporal Analysis of Fairness", 7th Annual Symposium on Principle of Programming Language.

[6] H.Nakashima: "Prolog/KR User's Manual", Faculty of Engneering, Univ. of Tokyo, Tech. Rep. METR82-4, March 1982.

[7] T.Chikayama: "UTILISP Manual", Faculty of Engneering, Univ. of Tokyo, Tech. Rep. METR81-6, September 1981.

[8] W.F.Clocksin, C.S.Mellish: "Programming in Prolog", Springer-Verlag, 1981.

[9] N.Kawato, T.Saito, F.Maruyama, T.Uehara: "Design and Verification of Large-Scale Computers by using DDL", 16th Design Automation Conference, June 1979.

[10] B.T.Hailpern: "Verifying Concurrent Processes Using Temporal Logic", Dep. Computer Science, Stanford Univ., August 1980.