ハードウェア状態遷移表現の Prolog による検証†

藤田昌宏†† 田中英彦††† 元岡達†††

近年，素子技術の進歩に伴い，大規模・複雑なシステムの設計を短期間に正確に行う必要が増している．従来のゲートレベルのシミュレーション[1]のみでなく，ハードウェアの仕様記述からゲートによる記述まで一貫して設計・検証を支援するシステムが望まれる[1]．そこで，われわれは時相論理[2),3)]を用いて仕様記述を行い，階層設計を支援する論理設計検証システムを Prolog[4] を用いて作成することを提案し，ゲート回路の検証について報告した[5]．本論文では，状態遷移レベルのハードウェア記述言語である DDL[7] の記述に対する検証について考える．一般にシステムは，各端子間のデータ転送のタイミングを扱う同期部 (synchronization part) と，ALU のように実際に計算を行う演算部 (function part) に分けて考えることができる．ここでは，モジュール間のインタフェースを考えるときなどにとくに問題となる同期部を中心に，DDL 等で状態遷移表現されたものに対する，時相論理による仕様の検証手法について述べる．DDL の記述は，現在と次の時刻の関係の表として Prolog に変換され，ゲートのときと同じようにして検証される．このため，DDL，ゲートの混在するシステムも検証でき，階層設計を円滑に支援できる．同期部の設計は人間の不得意な分野であり，誤設計も起こりやすく，本システムの実用的価値は大きいと考える．

1. まえがき

近年，素子技術の進歩に伴い，大規模・複雑なシステムの設計を短期間に正確に行う必要が増している．従来のゲートレベルのシミュレーション[1]のみではなく，ハードウェアの仕様記述からゲートによる記述まで一貫して設計・検証を支援するシステムが望まれる[1]．そこで，われわれは時相論理[2),3)]を用いて仕様記述を行い，階層設計を支援する論理設計検証システムを Prolog[4] を用いて作成することを提案し，ゲート回路の検証について報告した[5]．本論文では，状態遷移レベルのハードウェア記述言語 (HDL)[6] の一つである DDL[7] による記述に対する検証について考える．

一般にシステムは，各端子間のデータ転送のタイミングを扱う同期部 (synchronization part) 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．ここでは，モジュール間のインタフェースを考えるときなどにとくに問題となる同期部を中心に，DDL 等で状態遷移表現されたものに対する，時相論理による仕様の検証手法について述べる．DDL の記述は，現在と次の時刻の関係の表として Prolog/KR[8] に変換され，ゲートの時と同じようにして検証される．なお，処理系はすべて Prolog/KR で作成されているが以下ではたんに Prolog と書く．

2章では，DDL のサブセット DDL-S を設定し，本システムが扱う範囲を示す．3章で DDL-S の Prolog への変換手順を示し，4章で一般に状態遷移表現されたものの Prolog での表現法を述べるとともに，DDL-S による記述とゲートによる記述の混在するシステムも同じように検証できることを示す．5章では，異なるクロックをもつ複数のモジュール全体に対する検証手法を示す．そして，6章で検証例を示し，7章で処理能力について考察する．

2. 検証対象

一般にシステムは，演算部と同期部に分けて考えることができる．たとえば通常の計算機では，演算部は ALU に，また，同期部はメモリ，アキュムレータ，ALU 等の間のデータ転送が対応する．**図1**では，演算部は ALU 自体の記述に，同期部は各ゲート G1，G2，G3 を開閉する時刻の記述に対応する．演算部では，おもに与えられた入力データから指定された時間で指定された計算を終了しうるか否かが問題となるのに対し，同期部ではおもに指定されたデータが送られてきているか否かが問題となる．ここでは，システムが大規模・複雑になってくるととくに問題となる同期部について考える．われわれは，すでに仕様記述からゲートによる記述まで一貫して支援する検証システムを提案し，ゲート回路に対する検証手法を示した[5]．本論文で取り扱う検証対象の範囲は，状態遷移レベルの HDL の一つである DDL のサブセット DDL-S

† Verification of State-Transition Based Hardware Descriptions 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

Design, 16 th Design Automation Conf. (1979).
10) 丸山, 川戸, 上原：DDL ベリファイアによる論理設計の検証とその評価, 第23回情報処理学会全国大会 (1981).
11) Fujita, M., Tanaka, H. and Moto-oka T.: Verification with Prolog and Temporal Logic, IFIP 6 th Computer Hardware Description Languages and their Applications (1983).

(昭和58年 9 月 12 日受付)
(昭和58年12月 13 日採録)

付録 DDL-S syntax

```
ddl-des ::= (system-def declare-list descriptions)
system-def ::= (SYSTEM system-name)
declare-list ::= (DCL ({declare}*))
declare ::= (CONTROL-REGISTER control-register-list) |
            (DATA-REGISTER data-register-list) |
            (CONTROL-TERMINAL control-terminal-list) |
            (DATA-TERMINAL data-terminal-list)
descriptions ::= {automaton-des}*
automaton-des ::= (automaton-def declare-list-in-automaton
                    automaton-des-body)
automaton-def ::= (AUTOMATON automaton-name)
declare-list-in-automaton ::= (DCL ({declare-in-automaton}*))
declare-in-automaton ::= (CONTROL-REGISTER control-register-list) |
                         (DATA-REGISTER data-register-list) |
                         (CONTROL-TERMINAL control-terminal-list) |
                         (DATA-TERMINAL data-terminal-list) |
                         (STATE-NAME state-name-list)
automaton-des-body ::= ({logic-des}** {state-des}**)
logic-des ::= (LOGIC action)
state-des ::= (state-name action)
action ::= (:<= data-register source2) |
           (:= data-terminal source2) |
           (:<- control-register sourcel) |
           (:- control-terminal sourcel) |
           (IF condition action-then action-else) |
           (DO {action}*) |
           (:-> state-name)
action-then ::= action
action-else ::= action
sourcel ::= bit | control-register | control-terminal
source2 ::= const | data-register | data-terminal
condition ::= (NOT condition) |
              (AND {condition}*) |
              (OR {condition}*) |
              (== control-var bit) |
              (== control-var control-var)
control-var ::= control-register | control-terminal
bit ::= 0 | 1
control-register-list ::= ({control-register}*)
control-terminal-list ::= ({control-terminal}*)
data-register-list ::= ({data-terminal}*)
data-terminal-list ::= ({data-terminal}*)
state-name-list ::= ({state-name}*)
control-register ::= name
control-terminal ::= name
data-terminal ::= name
data-register ::= name
state-name ::= name                    { }* : 1回以上のくり返し
const ::= name                         { }** : 0回以上のくり返し
system-name ::= name
automaton-name ::= name
name ::= atom
```

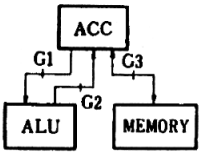


図 1 簡単な計算機
Fig. 1 A simple computer.

で設計されたものと，ゲートで設計されたものとが混在するシステムである．

DDL-S を BNF 記法で記述したものを付録に示す．おもな制限は，変数（レジスタ，ターミナル）を control と data の2種類に分け，実行の条件分岐の条件に使えるものは control 変数のみとし，かつ data 変数は制限しないが control 変数としては1ビット変数しか認めないこと，および，OR 等の論理演算を control 変数には認めるが data 変数には認めないことである．

これらは，検証範囲をおもに同期部にしぼることを意味する．演算部の検証は，別の手段を用いるか，本システムの記号シミュレーション[9]の能力を強化すること等が考えられる．しかし一般に，演算部は細かいタイミングはあまり問題とならず，また ALU 等のようにすでにいままでに設計されてきたものを一部手直しして使うことも多く，したがって誤設計も比較的少ない．

一方同期部は，各モジュール間のインタフェースのように細かいタイミングが問題となることも多く，また人間が不得意な領域でもあり，計算機による検証支援が強く望まれる．したがって，本システムの実用的価値は十分高いと考えられる．

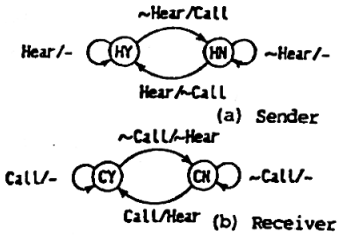
ハンドシェイクによるデータ転送システム[5]を状態遷移レベルで記状したものを図2に，それを DDL-S に書き直したものを図3に示す．本論文では，これらを具体的例として説明していく．

また，DDL のみでなく，文献5)の手法と併用することにより，DDL，ゲートの混在したモジュールも検証対象にでき，異なるクロックにより動かされる複数のモジュールにまたがる検証も取り扱うことができる．

3. DDL-S の Prolog/KR への変換

DDL-S で記述されたものを，各オートマトンの各変数（レジスタ，ターミナル）ごとに，現在と次の時刻の関係として Prolog に変換する．条件分岐の条件に使える変数を1ビットのみに制限したため，変換は容易に行える．

図3のような DDL-S による記述のうち，Sender については図4のように Prolog に変換される．図




```
LOGIC (in all states):
if Call=1 then Message := Infout
```

```
LOGIC:
if Call=1 and Hear=1
        then Infin <- Message
```

図 2 Sender と Receiver の DDL レベルの設計
Fig. 2 DDL level descriptions of Sender and Receiver.

```
((SYSTEM DATA-TRANSFER)
  (DCL ((CONTROL-REGISTER (CALL HEAR)) (DATA-TERMINAL (MESSAGE))))
  (((AUTOMATON SENDER)
    (DCL ((STATE-NAME (HY HN)) (DATA-REGISTER (INFOUT))))
    ((LOGIC (IF (== CALL 1) (:= MESSAGE INFOUT)))
     (HY (IF (== HEAR 1) (:-> HY) (DO (:<- CALL 1) (:-> HN))))
     (HN (IF (== HEAR 0) (:-> HN) (DO (:<- CALL 0) (:-> HY))))))
   ((AUTOMATON RECEIVER)
    (DCL ((STATE-NAME (CY CN)) (DATA-REGISTER (INFIN))))
    ((LOGIC (IF (AND (== CALL 1) (== HEAR 1)) (:<= INFIN MESSAGE)))
     (CN (IF (== CALL 0) (:-> CN) (DO (:<- HEAR 1) (:-> CY))))
     (CY (IF (== CALL 1) (:-> CY) (DO (:<- HEAR 0) (:-> CN))))))))
```

図 3 データ転送システムの DDL-S による記述
Fig. 3 The data transfer system in DDL-S.

し，状態 S_i 中の IF 文の数を N_i とすると，状態 S_i では，$2N_i$ だけ control 変数のとる値の組合せがある．IF 文に現れるのは control 変数のみであることから，状態遷移に関係するのは，状態名と control 変数の値のみであり，その組合せは，$(2\sum_{i=1}^{m} N_i)$ だけである．したがってどの状態から始めても，$(2\sum_{i=1}^{m} N_i+1)$ 回状態遷移すると，状態名と control 変数の値が，いままでの値のどれかと同じになることより，任意の状態から始めてループになるまでの状態遷移の最大値は，$2\sum_{i=1}^{m} N_i$ となる．したがって，これで検証の深さ（状態遷移の数）をおさえることができる．しかし，与えられる仕様にもよるが，通常の設計において，かなり前の状態に依存するような記述は現実にはないと考えられ，検証に必要な状態遷移の数はそれほど大きくならないと考えられる．したがって，同期部の検証に関する限り，かなり大きなモジュールに対しても使用できると考える．

条件文に現れる変数は1ビットであるため，IF 文において，then 側と else 側を明確に分けることができる．このため，すべてのパスのチェックが容易に行えるようになっているが，反面，記述能力はおちる．検証ではなく，たんなるシミュレーションのみであれば2章で述べた手法とほぼ同じ手法で通常の DDL[7] を Prolog に変換し，実行することができる．しかし，検証ではループチェックや IF 文の条件の否定の生成等があり，DDL を一般に扱うことはかなりむずかしく，本格的な記号シミュレーション[9]が要求される．したがって，処理時間や，処理系作成の容易さ等問題点も多く今後の課題である．しかし，その場合でも Prolog 等のパターン照合能力の強力な言語は有効である．現在でも DDL-S レベルでは，data 変数として任意のデータを扱えるので抽象度が上がっており，すべて1ビットずつ扱うゲートレベルと比べて，検証時間も速い．

簡単な，しかし強力な拡張として，Pascal のスカラ型のようなものを control 変数に許すことが考えられる．このようにすると，たとえば，CPU のインストラクションレジスタに対して，

type : IR=(nop, add, bra, lda, sta)

のようにして，インストラクションコードを与えることができ，各インストラクションに対応した処理を記述することができる．これは，ビット表現の組合せの記述と同じことであるが，みやすさにはかなりの差があり，また誤設計の減少も期待できる．

この程度の拡張を行うことで，システムのインタフェースのチェック等には，非常に有効な検証システムとすることができる．

8. むすび

DDL-S で記述されたシステムに対する，時相論理による仕様の検証手法について述べた．本手法により，DDL だけでなくゲートによる設計が混在したシステムも検証でき，また，多種のクロックがあるシステム全体を検証することもできる．したがって，階層設計が円滑に支援できる検証システムとすることができ，時相論理レベルの検証と組み合わせることにより，今後ますます大規模・複雑化していくハードウェアシステムの設計に十分対応できると考える．今後は，記号シミュレーションの強化による処理能力の拡大と，実際の使用結果をもとにした大きなモジュールに対する検証時間の検討を進めていく必要がある．

参考文献

1) Bening, L.: Development in Computer Simulation of Gate Level Physical Logic, Proc. 16-th DA Conf., pp. 561-567 (1978).
2) Wolper, P.: Temporal Logic Can Be More Expressive, 22 nd Annual Symposium on Foundation of Computer Science (1981).
3) Manna, Z.: Verification of Sequential Programs: Temporal Axiomatization, Technical Report No. STAN-CS-81-877, Dept. of Computer Science, Stanford University (1981).
4) Clocksin, W. F. and Mellish, C. S.: *Programming in Prolog*, Springer-Verlag, Berlin (1981).
5) 藤田，田中，元岡：時相論理を用いたハードウェア仕様記述と 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).
6) Hill, F. J., Chu, Y., Dietmeyer, D. L. and Siewiorek, D.: Introducing Hardware Description Language, *IEEE Comput.*, Vol. 17, No. 12, pp. 18-67 (1974).
7) 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).
8) 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).
9) Carter, W. C., Joyner, W. H. and Brand, D.: Symbolic Simulation for Correct Machine

```
(ASSERT (SENDER (#STATE #CALL #HEAR #INFOUT)
                (#@STATE #@CALL #@HEAR #@INFOUT)
                (#CL #MESSAGE))
        (IF (EQ #CL @)
            (TRUE)
            (AND (OR (AND (FEQ #CALL 1) (TRAN #MESSAGE #INFOUT)) (FEQ #CALL @))
                 (OR (AND (= #STATE HY)
                          (OR (AND (FEQ #HEAR 1) (= #@STATE HY))
                              (AND (FEQ #HEAR @)
                                   (AND (TRAN #@CALL 1) (= #@STATE HN)))))
                     (AND (= #STATE HN)
                          (OR (AND (FEQ #HEAR @) (= #@STATE HN))
                              (AND (FEQ #HEAR 1)
                                   (AND (TRAN #@CALL @) (= #@STATE HY)))))))))
```

図 4 Prolog/KR での Sender の表現
Fig. 4 Sender in Prolog/KR translated from DDL-S.

```
(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 ATOM1)) (= *@A *B))))
(ASSERT (FEQ (*VAR *C) *VAL)
      (IF (ATOM *C) (= *VAR *VAL) (= (*VAR *C) (*VAL ATOM2))))
```

図 5 システムプログラム TRAN, FEQ
Fig. 5 TRAN and FEQ programs.

```
(:<= data-reg source2)      ->  (TRAN #@data-reg #source2)
(:= data-ter source2)       ->  (TRAN #data-ter #source2)
(:<- control-reg source1)   ->  (TRAN #@control-reg #source1)
(:- control-ter source1)    ->  (TRAN #control-ter #source1)
(DO . action-list)          ->  (AND . action-list)
(:-> state-name)            ->  (= #@state state-name)
(IF condition action-then action-else)
  ->          (OR (AND condition-v action-then-v)
                  (AND not-condition-v action-else-v))
```

図 6 DDL-S の Prolog への変換法
Fig. 6 Method of translation from DDL-S to Prolog.

で # で始まるものが変数であり，@のついているものは次の時刻のもので，ないものが現在の時刻のものである．図4中の FEQ は，変数がある値と等しいかどうかを調べるシステムプレディケイトである．また，TRAN はレジスタ転送に関するシステムプレディケイトであり，**図5**に示す．各変数の内部表現は，ゲートのときとは異なり，(# var # c) の形をしており，# var に実際のその変数の値が入る．各時刻において，# c がアトムでないとき（変数のままのとき）には，この変数はまだほかからデータ転送を受けていないことを示し，アトムのときはデータ転送されたことを示す．この情報を用いて，変数がレジスタとしてふるまうよう，つまり，ほかからデータ転送されない限り前の時刻の値をもちつづけるように処理し(これは，REGUNCHANGED というプログラムで行っている)，また，同じ変数に同時に複数のところからデータ転送されていないかを調べる．

変換のおもな作業は，次のようになる．

① 各オートマトン名をプレディケイト名とし，引数として，レジスタ，内部状態，外部ターミナルを付ける．この際，次の形にする．

(automaton 名
レジスタと内部状態の現在の値のリスト
レジスタと内部状態の次の値のリスト
外部ターミナルの現在の値のリスト)

つまり，現在と次の時刻の間の各変数の関係とする．

② オートマトン内の状態の記述が現れるごとに OR で，一つの状態内の各処理を AND で結び，バックトラックによりすべての状態を調べられるようにする．P1, P2 をプレディケイトとしたとき，(OR P1 P2) は，まず P1 を実行し，もし何らかの理由で P1 が fail し，バックトラックしてきたときは，P2 を実行することを示している．したがって，各状態の記述を OR で結べば，バックトラックですべての状態を調べることができる．このように，Prolog/KR に OR があるため変換が容易になっている（通常の Prolog[4] に変換することも可能である）．

③ 各 action はおのおの**図6**の規則で変換する．レジスタへのデータ転送は，そのレジスタの次の時刻の値を表す変数（# の後に @ のついているもの）と転送元の変数をパターン照合させ，ターミナルへのデータ転送は，そのターミナルの現在の値を表す変数（@ のないもの）とパターン照合させる．状態遷移先の指定は，次の状態名を表す変数を遷移先状態名に一致させる．また，IF 文の処理は，then 側と else 側にそれぞれ，then 側の満たすべき条件(IF 文の条件)，else

```
                                    ~Call   ~Hear
(ASSERT (LOGICA ((#STATE #CALL (@ #1) #INFOUT) (#MESSAGE))))
(ASSERT (LOGIC~B ((#STATE (@ #1) #HEAR #INFOUT) (#MESSAGE))))
(ASSERT (STTRAN (#REG #TER) (#@REG #@TER)) (SENDER #REG #@REG (1 . #TER)))

                                                  Senderの検証
:(ver *)                                                        反例
((((HN (@ D) (@ D) #INFOUT_0404) (#MESSAGE_0404)))
 "type<>"
 ((HN (@ #1_0455) (*CARL_0462 . *X_0462) (*CARL_0465 . *X_0465))
  (#MESSAGE_0455)))
NIL
:
```

図 12 Sender に対する □(~Hear→▽Call) の検証
Fig. 12 Verification of □(~Hear→▽Call) to Sender.

```
                                          Call
(ASSERT (LOGICA                            ↘
          (((#SENDER #RECEIVER #CLSTATE) (1 #1) #HEAR #INFOUT #INFIN) NIL)))
(ASSERT (LOGIC~B                                  Hear
          (((#SENDER #RECEIVER #CLSTATE) #CALL (1 #1) #INFOUT #INFIN) NIL)))
(ASSERT (STTRAN (#REG #TER) (#@REG #@TER)) (DATA-TRANSFER #REG #@REG #TER))

:(ver *)                                                        反例
((((HY CN S11) (1 D) (1 D) (*VA@_2667 *X_2667) #INFIN_2569) NIL))
 "type<>"
 (((HY CY S10) (1 U) (1 C) (*CARL_2694 . *X_2694) (*VA@_2667 C)) NIL))
((((HY CY S11) (1 D) (1 D) (*VA@_2750 *X_2750) #INFIN_2569) NIL))
 "type<>"
 (((HY CY S10) (1 U) (1 U) (*CARL_2777 . *X_2777) (*VA@_2750 C)) NIL))
((((HY CY S10) (@ C) (1 C) (*VA@_3014 *X_3014) (*VA@_3022 C)) NIL)
  (((HN CN S11) (1 D) (1 D) (*VA@_3022 *X_2909) #INFIN_2569) NIL))
 "type<>"
 (((HY #@RECEIVER_2967 S11) (@ U) (1 U) (*CARL_3044 . *X_3044) (*VA@_3022 U)
  NIL))
                              ⋮      ⋮      ⋮
```

図 13 データ転送システムに対する □(Call→▽~Hear) の検証
Fig. 13 Verification of □(Call→▽~Hear) to the data transfer system.

れば反例として印刷する．反例がなければ設計が正しいことになる．時相論理の決定手続き[2]を用いれば，任意の時相論理の式に対する状態遷移列を求めることができ，したがって検証プログラムを作成することができる．

ゲートのとき[5]と同じように，時間軸に対して順方向にも逆方向にも検証できるが，ここでは順方向のみ示す．なお，時間軸に対し逆方向で検証する場合には，DDL ベリファイア[10]とほぼ同じ原理で動作する．

- DDL で記述された一つのモジュール (Sender) の検証

Sender を DDL で記述したものに対し，□(~Hear→▽Call) を検証する．検証プログラムは文献 5) のものと同じものを用いる．検証結果を図 12 に示す．□(A→▽B) を検証するプログラムを用いており，図 12 のように条件 A は LOGICA というプレディケイトで与え，条件 B は LOGIC~B というプレディケイトで与える．また，STTRAN は検証対象を指定するためのプログラムである．これは適当な初期条件がなければ反例が存在することを示している（詳細は文献 5) 参照のこと）．

- DDL で記述された複数のモジュール (Sender +Receiver) の検証

Sender, Receiver のクロックは，周期は同じで，位相のみ異なるとし，クロックの供給を図 11 のように決める．図 7 の Prolog の記述に直したものに対し，□(Call→▽~Hear)（ハンドシェイクの 1 サイクルの終了）を検証した例を図 13 に示す．この場合も適当な初期条件がなければ反例が存在することがわかる．なお，この場合と Sender, Receiver のクロックが同じであるとした場合との検証処理時間の比はだいたい，3 : 1 であった．

7. 考察・検討

DDL-S においては，変数を 1 ビットの control 変数と，一般のデータを扱う data 変数に分け，条件には control 変数のみ使えるようになっている．いま，DDL-S の記述において状態名を S_1, S_2, …, S_m と

```
(ASSERT (DATA-TRANSFER
          ((#SENDER #RECEIVER #CLSTATE) #CALL #HEAR #INFOUT #INFIN)
          ((#@SENDER #@RECEIVER #@CLSTATE) #@CALL #@HEAR #@INFOUT #@INFIN)
          NIL)
        (CLOCK #CLSTATE #@CLSTATE) (CLSTATE #CLSTATE #CLSENDER #CLRECEIVER)
        (SENDER (#SENDER #CALL #HEAR #INFOUT)
                (#@SENDER #@CALL #@HEAR #@INFOUT)
                (#CLSENDER #MESSAGE))
        (RECEIVER
           (#RECEIVER #CALL #HEAR #INFIN)
           (#@RECEIVER #@CALL #@HEAR #@INFIN)
           (#CLRECEIVER #MESSAGE))
        (REGUNCHANGED
           (#CALL #HEAR #INFOUT #INFIN)
           (#@CALL #@HEAR #@INGOUT #@INFIN)))
```

図 7 データ転送システムの Prolog/KR での記述
Fig. 7 DDL-S level descriptions of the data transfer system in Prolog/KR.

側の満たすべき条件(then 側の条件の否定) を AND で結んだものを OR で結ぶように変換し，バックトラックしたときに then 側，else 側どちらのパスもとるようにして，すべての場合を処理できるようにしている．

図6中，condition-v は IF 文の condition を Prolog/KR の記述に変換したものを表し，not-condition-v は condition の否定を Prolog/KR に変換したものを表している．このように IF 文を変換するため，IF 文の条件は，control 変数つまり1ビットの変数しか認めないようにし，条件の否定を作りやすくしている．したがって，IF の条件式に一般の変数を認めるようにするには，条件の否定等をうまく扱えるように本格的な記号シミュレーションを行う必要がある．

④ クロックに対する処理をつけ加える．

(IF (= #CL 0) (TRUE) …) をつけ加えて，そのモジュールにクロック (#CL) がこなければ (#CL=0) 何もせず，くれば (#CL=1) 状態遷移をするようにしている．

この変換プログラムも Prolog でかかれており (約300行)，処理時間は，図3を変換する場合 M-200 H で約 0.1 秒である．

複数のオートマトンやゲートで設計されたシステム全体に対する Prolog の表現は，各オートマトンやゲートに対応する記述を並べることによって作る．たとえば，図3の Sender と Receiver を合わせたデータ転送システムに対する Prolog の表現は，図7のようになる．図7の Sender や Receiver は，DDL-S からオートマトン単位に変換された図4のようなサブゴールとパターン照合される．

図中，REGUNCHANGED は，ほかからデータ転送されなかったレジスタの次の時刻の値を現在の値と一致させるシステムプレディケイトである．また，

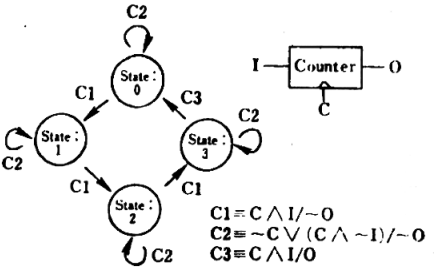


図 8 2ビットカウンタの状態遷移図
Fig. 8 State diagram for 2 bit counter.

CLOCK, CLSTATE は，各モジュールに分配されるクロックを決めるもので，5章で説明する．

4. ゲート，DDL-S の混在したシステムの検証

ゲートも DDL-S も Prolog では同じように，現在と次の時刻の関係として表現されている．したがって，ゲートに対しても DDL-S に対しても同じように検証でき，複数のモジュールのうちいくつかが DDL-S で記述され，残りがゲートで記述されているような両者が混在したシステム全体の検証も簡単に行える．複数のモジュール全体に対する記述を作るために図7のように各モジュールの記述をリストとして並べていく際に，DDL-S による記述を前にもってくるようにし，各変数のデータ構造を調整するだけで他の処理は変更する必要はない．制限される点は，ゲートレベルでは変数を 0，1 の2値で扱っているので，DDL-S においても各変数とも1ビットである必要があることである．

また，ゲートを Prolog で表現する際にも，ある程度規模の大きいもの (たとえば，MSI 等) は，必ずしも基本ゲートに展開する必要はなく，状態遷移モデルから Prolog に直すこともできる．

```
Present_state   Input      Clock
      Next_state    Output
(ASSERT (COUNTER2 3 0 (1 1 1)))
(ASSERT (COUNTER2 3 3 (0 0 1)))
(ASSERT (COUNTER2 0 1 (1 0 1)))
(ASSERT (COUNTER2 0 0 (0 0 1)))
(ASSERT (COUNTER2 1 2 (1 0 1)))
(ASSERT (COUNTER2 1 1 (0 0 1)))
(ASSERT (COUNTER2 2 3 (1 0 1)))
(ASSERT (COUNTER2 2 2 (0 0 1)))
(ASSERT (COUNTER2 0 0 (*I 0 0)))
(ASSERT (COUNTER2 1 1 (*I 0 0)))
(ASSERT (COUNTER2 2 2 (*I 0 0)))
(ASSERT (COUNTER2 3 3 (*I 0 0)))
```

図 9 図8の状態遷移図の Prolog による記述
Fig. 9 Prolog program translated from Fig. 8.

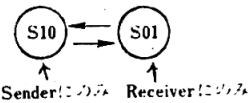


図 10 クロック供給を決める状態遷移図
Fig. 10 State diagram for clock delivery.

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

図 11 クロックの供給
Fig. 11 Prolog program for clock delivery.

たとえば，2ビット（4進）カウンタは次のような手順で Prolog に直すことができる．まず，状態遷移図で表すと**図8**のようになる．図で1が入力でCがクロックであり，1とCがともに1であることが4回起こると，出力0が1になるようになっている．これを現在の内部状態，次の内部状態，それに現在の外部ターミナルの値を引数として**図9**のように Prolog で表現できる．このように比較的大きな回路は，状態遷移モデルを用いて効率よく表現することができる．

5. クロックの異なるモジュール間の検証

本章では，クロックの異なるモジュールをもつシステムについての検証手法を示す．データ転送システムを例にとって説明する．

Sender のクロック周期を clsender, Receiver のクロック周期を clreceiver とすると，一般に，

$$1/n\ |\text{clsender}| \leqq |\text{clreceiver}| \leqq n\ |\text{clsender}|$$

のときは，Sender, Receiver の片方にのみ連続して，n 回までクロックがくる可能性がある．したがって，クロックの分配を片方に連続して n 回まで供給するように決めればよい．これは状態遷移表現で容易に記述できる．たとえば，

$$|\text{clsender}| = |\text{clreceiver}|$$

のとき，つまり両方のクロック周期は同じで，位相のみがずれているとき（一般にこのような場合が多い）には，**図10**のような状態遷移表現によってクロックの供給を決める．S10 は Sender のみに，S01 は Receiver のみにクロックを供給することを示す．これは，片方にのみ続けてクロックを供給することがないように決まっている．図10を Prolog に直したものが**図11**である．まず，CLOCK によりクロックの状態を求める．次に CLSTATE によりその状態から具体的なクロックの供給を決める．このようにして，一般的にクロックの供給を決めることができる．

6. 検 証 例

本章では，DDL で設計されたモジュールに対する検証例を示す．ハンドシェイクによるデータ転送システムにおいて，Sender, Receiver 各モジュールを DDL-S で設計したものに対し，ゲートのとき[5]と同じように上位レベルの時相論理による仕様を満たしているかどうかを検証した例について述べる．

ここで簡単に時相論理，および，検証法について説明する．詳しくは参考文献[2),3),5),11)]を参照されたい．時相論理は古典論理に，□(always)，▽(sometime)，○(next)，U(until) の四つの演算子を付け加えたものであり，各演算子は次のような意味を表す．

□P：現在から先いつも P，▽P：いつか P，

○P：次の時刻に P，

P1 U P2：P2になるまで P1 であり続ける

これらを用いて時間軸上のシーケンスを表現することができる．たとえば，「信号 P が active なら，いつかは信号 Q が active になる」は次のように表現できる．

$$\square(P \rightarrow \triangledown Q) \qquad ①$$

検証は時相論理の決定手続きに基づき，背理法で行う．また，①の検証はすべての場合について

$(P \rightarrow \triangledown Q)$ を調べることで行う．まず，検証すべき仕様の否定をとる．いまの場合，

$$\sim(P \rightarrow \triangledown Q) = P \wedge \square \sim Q \qquad ②$$

となる．②は，「現在 $P \wedge \sim Q$ であり，かつ次の時刻以降ずっと $\sim Q$ である」ような状態遷移列を表している．したがって検証すべき回路にそのような状態遷移列があるか否かをバックトラックを用いてすべての場合について調べ，もし，そのような状態遷移列があ