時相論理によるハードウェア同期部の仕様記述と Prolog によるその状態遷移表への自動合成法†

藤田昌宏†† 田中英彦††† 元岡達†††

近年の素子技術や実装技術の進歩に伴い，ハードウェアシステムはますます大規模・複雑なものとなっており，計算機支援設計が必要不可欠となっている．われわれはすでに，時相論理 (Temporal Logic)[1] を用いて仕様を記述し，処理系に Prolog[2] を用いてハードウェア論理設計を検証することを提案し，ゲート回路や状態遷移レベルのハードウェア記述言語である DDL[3] の記述に対する具体的手法を示した[4)~6)]．そこでは，検証対象をシステムの状態を制御する同期部に絞り，必要なすべての場合を調べることで検証している．同期部は並列に動作するもの全体を考えなければならず，誤設計を起こしやすい．本論文では，その同期部について，時相論理による仕様から状態遷移表を自動合成する方式について述べる．状態遷移表は，時相論理の決定手続きを用いて，仕様である時相論理の記述を現在に対する条件と次の時刻以降に対する条件に展開することによって合成することができる[7),8)]が，そのままでは時相演算子の数に対して処理時間が指数的に増大する．そこでここでは，Prolog を用いた処理時間を多項式オーダに抑える手法を示すとともに，実際的なハードウェアに応用した例を示し，人手による設計との比較等から実用性のあることを示す．また本手法を文献[4)~6)]と組み合わせることにより，ゲート回路，DDL，時相論理のいずれの記述も，また，混合した記述も検証することができ，階層設計を円滑に支援することができる．

1. はじめに

近年の素子技術や実装技術の進歩に伴い，ハードウェアシステムはますます大規模・複雑なものとなっている．このため，設計者の負担を軽くし誤りを防ぐように，定式的に仕様を記述できるようにし，従来のようなシミュレーションだけでなく，設計の早い段階から検証できたり，あるいは仕様から設計を自動合成できるような一貫して階層設計を支援できるツールが強く望まれている．

そこでわれわれは，時相論理 (Temporal Logic)[1] を用いて仕様を記述し，処理系に Prolog[2] を用いてハードウェア論理設計を検証することを提案し，ゲート回路や状態遷移レベルのハードウェア記述言語である DDL[3] の記述に対する具体的手法を示した[4)~6)]．そこでは，検証対象をシステムの状態を制御する同期部に絞り，必要なすべての場合を調べることで検証している．同期部は並列に動作するもの全体を考えなければならず，誤設計を起こしやすいところであり，検証をとくに必要としている．本論文では，同期部について，時相論理による仕様から状態遷移表を自動合成する方式について述べる．状態遷移表は，時相論理の決定手続きを用いて，仕様である時相論理の記述を現在に対する条件と次の時刻以降に対する条件に展開することによって合成することができるが，そのままでは時相演算子の数に対して処理時間が指数的に増大する[7),8)]．そこでここでは，Prolog を用いた処理時間を多項式オーダに抑える手法を示すとともに，実際的なハードウェアに応用した例を示し，人手による設計との比較等から実用性のあることを示す．また本手法により，任意の時相論理の記述を状態遷移表現に変換できるため，時相論理により記述されたハードウェアが時相論理の仕様を満たすか否かの検証を行うこともできる．したがって，文献[4)~6)]と組み合わせることにより，ゲート回路，DDL，時相論理のいずれの記述も，また，混合した記述も検証することができ，階層設計を円滑に支援することができる．

2章では時相論理を用いたハードウェア同期部の仕様記述について，例を用いて説明する．3章では時相論理の決定手続きについてその概要を述べ，自動合成のアルゴリズムについて説明する．4章で効率的なアルゴリズムを示すとともにその Prolog での実装法を示す．5章では実際的なハードウェアの自動合成を具体的に行った例を示す．そして，6章で処理能力や人手の設計との比較等の考察を行い，実用性について検討し，最後に7章で結論を述べる．

† Specification of Hardware in Temporal Logic and Automatic Synthesis of State-Diagram Using Prolog by MASAHIRO FUJITA (Information Engineering Course, Graduate School of Engineering, The University of Tokyo), HIDEHIKO TANAKA and TOHRU MOTO-OKA (Department of Electrical Engineering, Faculty of Engineering, The 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

なお，処理系には移植性を考慮して Edinburgh 大学で開発された標準的な Prolog である C-Prolog[9] を用いた．

2. 時相論理を用いたハードウェア同期部の仕様記述

2.1 時相論理とタイムチャートの記述

一般にシステムは，実際に論理・算術演算を行う演算部と，各演算間のタイミングを制御し同期をとる同期部に分けることができる[5]．同期部では並列に動作するもの全体を考えなければならず，誤設計を生じやすい．したがって，設計支援がとくに望まれる．

本章ではまず時相論理について簡単に述べ，次に例を用いてハードウェア同期部の仕様記述について説明する．なお，時相論理についての詳細は参考文献[1]を参照されたい．

時相論理は，∧, ∨, →, ～等の古典論理の演算子に，○ (next)，□ (always)，▽ (sometime)，U (until) の四つの時相演算子を加えたものであり，各演算子は次のような意味をもつ．

○P：次の時刻（同期回路では，次のクロック）にPが成り立つ

□P：現在から将来ずっとPが成り立つ

▽P：現在から将来の少なくとも1時刻でPが成り立つ

PUQ：現在から考えて，Qが成り立つまではPでありつづける（ここでは，Qが必ずしも成立することを要求しない weak until を用いる）[7]．

時相論理を用いて通常タイムチャートで表されるような時間順序関係を表現することができる．

まず，『信号Pが active になると次の時刻に信号Qが active になる』は，

$$\Box(P\rightarrow\bigcirc Q) \qquad ①$$

と表現できる．また，具体的に時刻はいえないが，将来少なくともQが active になる場合は○を▽にかえて次のように表現できる．

$$\Box(P\rightarrow\triangledown Q) \qquad ②$$

条件①や②では，『Pが active になるとQが active になる』ことは保証するが，他の場合にもQが active になるかもしれない．これを『Qが active へ変化するのは，Pが active になった次の時刻であり，かつその時に限る』とするには，次の条件を①に付け加える．

$$\Box(\sim Q\rightarrow((\bigcirc\sim Q)\cup P)) \qquad ③$$

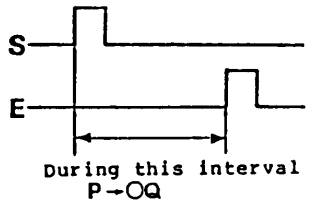

図1 タイムチャートの例
Fig. 1 A timing diagram.

(以降，時相論理の式を並べたものは，各式の積 (AND) を表すものとする)

信号の因果関係の表現は，①と③を合わせたものが基本となる．また，図1に示すタイムチャートのように，『スタート信号Sとエンド信号Eの間の時刻では信号Pが active になると次の時刻信号Qが active になる』は，次のようになる．

$$\Box(S\rightarrow((P\rightarrow\bigcirc Q)\cup E)) \qquad ④$$

(ただし，SとEは図1のようにパルス状に外部から与えられ，かつSがEより先に来るとする．このことは，次のように時相論理で表現できる．

$$\Box(S\rightarrow((\bigcirc\sim S)\cup E)),$$
$$(\bigcirc\sim E)\cup S,$$
$$\Box(E\rightarrow((\bigcirc\sim E)\cup S)) \quad)$$

このように複雑な仕様をそのまま記述すると，時相演算子が次々とネスティングされる．しかし，スタート信号Sとエンド信号Eの間の時刻 (interval)[10] のみIという信号が active になるというように記述すると簡単な条件の積で表現できる．

$$\Box(\sim I\rightarrow((\sim I)\cup S))),$$
$$\Box(S\rightarrow I),$$
$$\Box(I\rightarrow(I\cup E)), \qquad ⑤$$
$$\Box(E\rightarrow(\sim I)),$$
$$\Box((I\wedge P)\rightarrow\bigcirc Q)$$

これは，最初四つの条件で信号Iが active になる時刻をSとEの間の時刻のみにし，最後の条件でそのときにPが active ならば次の時刻Qが active になることを表現している．

一般に⑤のように適当な時間の幅 (interval) を考えることによって，複雑な仕様も簡単な条件の積で表現することができる．このIは外部には関係ない内部状態を表すものであるが，このようにすることにより仕様が記述しやすくなり，また，⑤からわかるように同じ形の条件式が多いため，後で述べるように複雑な仕様の自動合成に要する時間を低く抑えることができる．

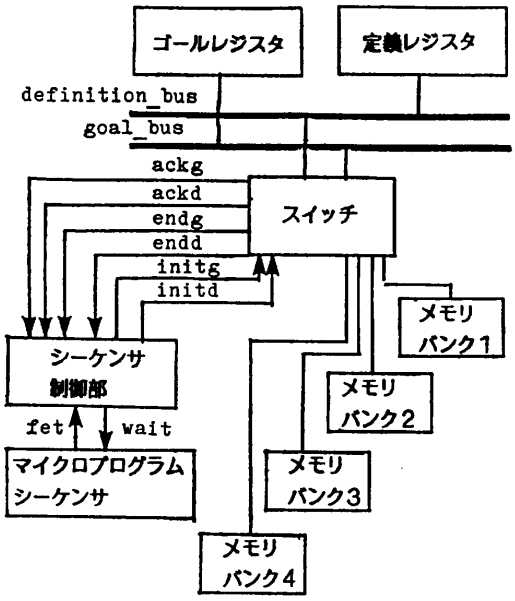

図 2 UP の内部構成[13]
Fig. 2 Internal structure of UP[13].

2.2 同期部の仕様記述例

ここでは，実際的なハードウェア同期部の仕様記述例を示す．われわれの研究室では高並列推論マシンPIE[11]の研究・開発を進めているが，その中心部をなすのがユニフィケーションを実行する Unify Processor[12]（以下 UP と略す）であり，現在試作が進められている．試作 UP[12] はマイクロプログラム制御で動作し，内部は図2のような構成になっている．ゴール (goal) レジスタ，定義 (definition) レジスタが，それぞれゴールバス(g__bus), 定義バス (d__bus) につながっており，各バスはスイッチを通して四つのバンクに分かれたメモリと接続されている．ゴールレジスタ，定義レジスタはバスを通してそれぞれ一つのメモリバンクを参照する．同一のメモリバンクを参照しようとして，アクセスが競合した場合には必ずゴール側を優先することになっている．しかし，マイクロ命令によっては，メモリ参照が1マイクロサイクルで終了せず何サイクルも続けなければならないことがある（このようなマイクロ命令を以下『メモリ連続参照命令』と呼ぶ）．もしこのような場合に，もう一方からアクセス要求が来るとそれを待たす必要がある．また，各レジスタのメモリ参照が何サイクルも続く場合には，マイクロプログラムシーケンサに対し現在のマイクロプログラムの実行を続行するよう要求する必要がある．以上のような制御を行う『シーケンサ制御部』について時相論理で記述する（ただし，ここでは説明のため多少簡単化している）．

まず，仕様記述に用いる各信号の意味を説明する（以下の信号はすべて1ビットである）．

fet：メモリに対し連続して参照を行うマイクロ命令（メモリ連続参照命令）を実行中であるか否かを示す信号

wait：シーケンサに対し，現在のマイクロ命令の実行が継続中であるか否かを示す信号

initg：そのメモリ連続参照命令中でのゴール側からの最初のアクセスであるか否かを示す信号

endg：ゴール側のメモリ連続参照が終了したか否かを示す信号

ackg：ゴール側が実際にアクセス権を得たか否かを示す信号

initg：そのメモリ連続参照命令中での定義側からの最初のアクセスであるか否かを示す信号

endg：定義側のメモリ連続参照が終了したか否かを示す信号

ackg：定義側が実際にアクセス権を得たか否かを示す信号

次に『シーケンサ制御部』を記述する．上述のように，①∧③の形の式がよく現れるので次の簡略記法を導入する．

iff-next (A, B)≡
□(A→○B)∧□(～B→((○～B)∪A))

iff-present (A, B)≡
□(A→B)∧□(～B→((～B)∪A))

シーケンサ制御部は，入力信号として ackg, ackd, endg, endd, fet を受け取り，信号 initg, initd, wait を制御する．

(S-1) ゴールレジスタが，最初のアクセス(initg)から2回目以降のアクセス (～initg) に進めるのは，現在メモリ連続参照命令を実行中 (fet) であり，かつゴール側が実際にアクセス権を得た (ackg) 場合に限る．

iff-next ((initg∧fet∧ackg), ～initg).

(S-2) 同じことが定義側についても成立する．

iff-next ((initd∧fet∧ackd), ～initd).

(S-3) ゴールレジスタが次のメモリ連続参照命令の最初のアクセス (initg) に進めるのは，メモリアクセスをしなくなる (～fet) か，または，メモリアクセスを続行中でゴール定義両レジスタともアクセスを終了 (endg∧endd) しかつ最初のメモリアクセスでない (～initg) ときに限る．

iff-next ((~fet∨(fet∧endg∧endd∧~initg)), initg).

(S-4) 同じことが定義側についても成立する.

iff-next ((~fet∨(fet∧endg∧endd∧~initd)), initd).

(S-5) シーケンサに対し wait を開始するのは, メモリ連続参照命令 (fet) の最初のサイクル (initg∨initd) のときに限る.

iff-present (((initg∨initd)∧fet), wait).

(S-6) シーケンサに対し wait を終了するのは, (S-3), (S-4)と同じ条件である.

iff-present ((~fet∨(fet∧endg∧endd ∧~initg∧~initd)), ~wait).

また, シーケンサ制御部の合成の際に, スイッチに対する次の仕様[13]を外部条件として用いる.

□(endg→(endg U (~fet∨initg)) ⑥

□(endd→(endd U (~fet∨initd)) ⑦

3. 合成アルゴリズム

ここでは, 時相論理の決定手続きに基づく状態遷移表現の自動合成アルゴリズムについて簡単に説明する. 詳細については, 参考文献[7),8)]を参照されたい. 文献[7)]では extended temporal logic について示されているが, 本論文では2章で述べた通常の時相論理 (linear time temporal logic)[1)] に適用する.

合成は次のようにして行える. まず, 時相論理の各演算子を一定の規則にしたがって, 現在に関する条件と次の時刻以降に対する条件に展開する. 次の時刻以降に対する条件についても, 一番外側の○演算子を取り除いたものを同じように展開して, 次の時刻に対する条件と次の次の時刻以降に対する条件に分ける. このような展開をすでに処理したことのある条件と一致するまで繰り返していく. 展開の結果得られるすべての条件を処理すれば, 各時刻での条件が状態に, 各展開過程が遷移に対応した状態遷移表現が得られている. 各時相演算子の展開規則は**表1**のようになっている. 表1は時相論理の公理[1)]から得られるものでたとえば, 〈1〉は『ずっとPであることは, 現在Pであり, かつ, 次の時刻からみてもずっとPである』ということを表している. また〈2〉は, 『いつかPであるということは, 現在Pであるか, または, 現在はPでなくかつ, 次の時刻からみていつかPである』ということを意味する. しかし, 〈2〉においていつも ~P∧○▽P を選択していると, ずっと ~P となってしまい ▽P を満たさない. したがって, ~P∧○▽P は『いつかはPを選択する』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, ~P∧○▽P の後ろの {P} はこの eventuality を示している. この表を用いることによって, 任意の時相論理の式を現在に対する条件と次の時刻以降に対する条件に展開できる. 展開過程に現れるすべての条件を展開した後, もし途中に eventuality がある場合には, 得られた状態遷移表現に対しても実際に eventuality が満たされているか否か調べ, もし満たされない状態遷移があればこれを削除する.

表1 時相演算子の展開規則[7)]
Table 1 Expansion rules for temporal operators.[7)]

〈1〉	□P=P∧○□P
〈2〉	▽P=P∨(~P∧○▽P{P})
〈3〉	P1 U P2= P2∨(P1∧~P2∧○(P1 U P2)
〈4〉	~□P=~P∨(P∧○(~□P) {~P})
〈5〉	~▽P=~P∧○(~▽P)
〈6〉	~(P1 U P2)=(~P1∧~P2) ∨(~P2∧○(~(P1 U P2)) {~P1})

例として, □(P→△Q) を展開してみる. まず表1の〈1〉,〈2〉を参照しながら展開する.

□(P→▽Q)
=(P→▽Q)∧○□(P→▽Q) 〈1〉
=(~P∨(P∧▽Q))∧○□(P→▽Q)
=(~P∨(P∧(Q∨(~Q∧○▽Q{Q}))))
∧○□(P→▽Q) 〈2〉
=((~P∨Q)∧○□(P→▽Q))
∨((P∧~Q)∧○(□(P→▽Q)∧▽Q{Q}))⑧

新しく □(P→▽Q)∧▽Q{Q} が現れたので, 次にこれを展開する.

▽Q∧□(P→▽Q)
=(Q∧○□(P→▽Q))
∨(~Q∧□(P→▽Q)∧▽Q{Q}) ⑨

これらはいずれもいままでに現れてきた条件と同じである. ⑧, ⑨から**図3(a)**のような状態遷移図が得られる. 図の状態2で*の遷移を取り続けると {Q} の eventuality が満たされない. そこで自動合成では遷移*を削除し, (b)のような状態遷移図を生成するようにする.

4. Prolog による実装

本章では, 3章で示した合成アルゴリズムを Prolog で実装する手法および, 効率的なアルゴリズムとその実装法について説明する. なお処理系には Edinburgh

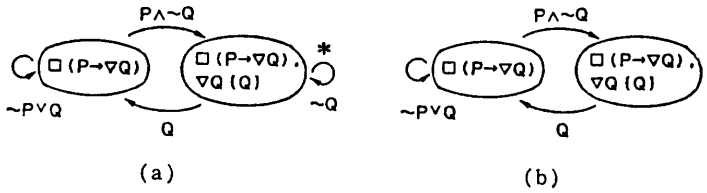

図 3 □(P→▽Q) の展開
(a) 表1による展開, (b) eventuality を満たさない遷移の削除
Fig. 3 Expansion of □(P→▽Q).
(a) Expansion by Table 1, (b) Removal of transitions which do not satisfy some eventualities.

```
   present      next          present and next
   conditions   conditions    eventualities

develop([alw,F],[[alw,F]|Fnn],E,En,P):-
        if_exist_delete(F,E,E1),      present values of variables
        develop(F,Fn,E1,En,P),
        simplify(Fn,Fnn).             if F is in E, E is deleted from F
develop([eve,F],Fn,E,En,P):-          and the rest is E1
        if_exist_delete(F,E,E1),
        develop(F,Fn,E1,En,P).        simplification of expressions
develop([eve,F],[[eve,F]|Fnn],E,[Ff|En1],P):-
        simplify(F,Ff),
        if_exist_delete([not,F],E,E1),
        develop([not,F],Fn,E1,En1,P),
        simplify(Fn,Fnn).
```

図 4 表1の〈1〉,〈2〉の Prolog による記述
Fig. 4 Prolog descriptions for 〈1〉 and 〈2〉 of Table 1.

表 2 時相論理のリスト表現
Table 2 List expressions for temporal logic.

〈1〉	□P=[alw, P]
〈2〉	▽P=[eve, P]
〈3〉	P1 U P2=[unt, P1, P2]
〈4〉	~P=[not, P]
〈5〉	P1∧P2=[and, P1, P2]
〈6〉	P1∨P2=[or, P1, P2]

大学で開発された C-Prolog[9] を用いた.

任意の時相論理の式が, 表2に示すような形のリストで与えられるとすると, 表1に示す各時相演算子の規則にしたがって任意の式を展開するプレディケイト develop の主要部は図4のようになる. develop において第1引数が展開すべき式であり, 次の引数が次の時刻以降に対する条件式, その次の二つの引数が現在と次の eventuality である. また, 最後の引数は各変数の現在の値のリストである. この develop に対し強制的にバックトラックをかけることにより, 次の時刻以降に対する条件をすべて求めることができる. したがって3章に示したアルゴリズムにしたがって処理すれば状態遷移表現を得ることができる. また eventuality については, ループになった状態遷移列に対して, 状態を順に調べてゆき各 eventuality が満たされているか否かを調べる. このように Prolog のもつ強力なパターン照合機構や自動バックトラック機構により, きわめて簡単に処理プログラムを作成することができる. しかしこの方法では, 合成に要する時間は, 時相演算子の数に対して指数的に増大する[7),8)]. したがって実用規模のハードウェアを合成するためには, 何らかの工夫が必要となる. 2章で示したように実際の仕様には同じような形の式がかなり多い. したがってよく使われる形の式はあらかじめ展開して状態遷移表現に直しておき, 実際の展開においては, その展開された状態遷移表現を組み合わせて合成することができれば, かなり効率化を図ることができる.

実際の仕様Tは, 時相論理の式 $T_1, T_2, \cdots, T_n$ に対し,

$$T = T_1 \wedge T_2 \wedge \cdots \wedge T_n \qquad ⑩$$

の形で記述できる. このことからTに対する状態遷移表現は, $T_1, \cdots, T_n$ のおのおのの状態遷移表現をまとめ, 可能なすべての状態遷移をさせることによって得ることができることがわかる. したがって, 状態遷移表現の自動合成は次の合成アルゴリズムのように効率的に行うことができる.

〈合成アルゴリズム〉

(ステップ 1) 仕様は⑩の形で記述されているも

```
procedure synthesis ;
var L: set of state-set of all Ti ;
  begin
    L := {set of initial states of all Ti } ;
    while L ≠ ∅ do begin
    choose a set of state, say s from L and delete it from L ;
      for all sets of input, possible in s do begin
        with this set of input, state-transition s for all Ti ;
        let L' be the set of new sets of states ;
        for each s' ∈ L' do begin
          s' is a successor of s ;
          if s' has not been visited then
            add s' to L ;
        end ;
      end ;
    end ;
  end .
```

図5 複数の状態遷移表現から全体に対する状態遷移表現を求める手続き
Fig. 5 A procedure to make the global state-diagram from given state-diagrams.

のとする．まず，各 T_i は図4のプレディケイト 'develop' を用いて状態遷移表現に展開しておく．

(ステップ 2) 各 T_i に対する状態遷移表現から図5にしたがってTに対する状態遷移表現を作る（なお，Prologを用いて，複数の状態遷移表現から全体に対する状態遷移表現を作る方法については参考文献[5]を参照されたい）．

(ステップ 3) 各状態遷移について eventuality を調べ，満たさないものがあればその遷移を削除する．

(ステップ 4) できた状態遷移表現を簡単化する．すなわち，同じ処理をしている状態を一つにまとめ，各遷移条件の論理式を簡単化する．

また，上のアルゴリズムにおいて，ステップ4の結果を再びステップ2で用いることもできる．したがって⑩で与えられる仕様に対して，T_1〜T_n をいくつかのグループに分け，まず各グループごとに合成してから後で全体に対する状態遷移表現を作ることもできる．このようにすることで処理時間を多項式オーダに抑えることができる（6章参照）．さらに，ステップ2で用いる状態遷移表現は，ゲートや DDL の設計記述から得られるもの[4),5)]でもよく，すでに設計されたハードウェアに新たに仕様を付け加えた新しいハードウェアに対する状態遷移表現も効率よく作成することもできる．

5. 合 成 例

本章では2.2節で示した UP の『シーケンサ制御部』を実際に合成した結果を示す．合成の手順は以下のとおりである．

(1) 仕様は(S-1)〜(S-6)および，⑥,⑦であり，いずれも *iff-next*, *iff-present*, それに □(A→(A U B) のいずれかなので，まずこれらを develop を用いて展開する．

(2) 合成アルゴリズムのステップ2を実行する．仕様を initg, wait を制御する(S-1),(S-3),(S-5),(S-6),⑥と，initd を制御する(S-2),(S-4),⑦の二つのグループに分けそれぞれ図5にしたがって合成する．

(3) (2)で得られた二つの状態遷移表現の冗長性除去を行う．

(4) (3)の二つの状態遷移表現を再び合成アルゴリズムのステップ2にしたがって全体に対する状態遷移表現を作る．

(5) (4)で得られた全体に対する状態遷移表現の冗長性除去を行う．

上記の各ステップにおける処理時間を以下に示す．

ステップ	1	2	3	4	5
処理時間	0.88	1.03	1.68	0.36	0.82

使用計算機は東京大学計算機センターの M-280 H であり，単位は秒 (CPU 時間) である．実際に自動合成した結果の詳細については参考文献[13)]を参照されたい．合成された状態遷移表現の状態数は8であり，フリップフロップ3個で表現できる．これに対し人手による設計では，『シーケンサ制御部』全体は複雑すぎて一度に考えることはできず，二つに分けて設計したため，おのおのについて3状態必要となり，全体としてフリップフロップ4個必要となっている．

6. 考察・検討

6.1 仕様記述について

2章で示したように，時相論理を用いて各信号間の時間順序関係を簡潔に表現することができる．複雑な関係をそのまま表現しようとすると，時相演算子が次次とネスティングされ，直観的に理解するのがむずか

しくなってくる．しかし，2.2 節の⑤における I のように，それが active のときのみある関係が成立するような変数を内部状態を表すものとして，設計者が定義してやることにより，簡単な条件の積に分解することができ，合成・検証時間を低く抑えることができる．外部仕様に内部の状態を表すような変数を用いるのは，不自然であるという考えもあるが，設計者が記述しやすく，理解しやすくなり，また，合成・検証しやすくなるため，このような記述法がハードウェア同期部の仕様記述には向いていると考える．

また，2 章のように実際の仕様記述に現れる時相論理の条件式の種類はそれほど多くないため，よく使われる種類の条件式はすべて前もって展開して知識として蓄えておくことができる．このようにすれば，実際の合成はすでに状態遷移表現になっているものからいくつかを選んで行うことができ，さらに新しく合成されたものも知識として付け加えていくことができるため，合成を円滑に行うことができる．

また，実際の設計では仕様自体に矛盾を含んでいることも多い．通常タイムチャートを用いて設計する場合には，まず，各信号間の立上り，立下りの関係をタイムチャートにして描き，もし各信号が同じ状態にあるときに，ある信号があるときは立ち上がり，あるときは立ち下がるような場合があると，その二つを区別するために別のもう一つの信号を加えるということを行っている．この過程は前章までに述べた方法により次のように支援できる．

（1） 正しいと思われる仕様を記述する．

（2） (1)の仕様を合成アルゴリズムによって展開する．

（3） もし展開途中で矛盾の状態（ある信号が同時に active でかつ inactive でなければならない状態）に陥ったら，その展開に対応する入力信号に対しては矛盾を起こしていることになるので，仕様を修正し，(2)へもどる．もしすべての展開がうまくいったら終了する．

このような通常タイムチャートをもとに人が行っていた仕様の修正を時相論理で仕様を記述することにより，計算機で円滑に支援することができる．

6.2 合成時間，および，合成結果について

5 章で示した合成例では，処理時間が短く，合成結果も人手によるものに対し遜色なく，十分実用的であるといえる．

合成に要する時間は，与えられた時相論理の仕様全体をそのまま表 1 にしたがって展開するようにすると，時相演算子の数に対し，最悪の場合指数的に増大する[7),8)]．しかし，一般に仕様は 4 章の⑩の形で記述されているため，4 章で示したアルゴリズムにしたがって展開するとすれば，以下に示すように多項式オーダの処理時間に抑えられる．まず，N_i, N_o をそれぞれ入力および出力信号の数とすると，⑩における n は 2 章で示したように仕様は各信号の立上り，立下り条件を記述するため，

$$n \propto N_o$$

となる．全体に対する状態遷移表現は，各条件式 T_i を一つずつ加えながら行うものとすると，全体を合成するために必要な繰り返し数 N_r（ステップ 2，4 を繰り返す数）は，

$$N_r \propto n \propto N_o$$

となる．また，N_s を合成した結果最後に得られる状態数とすると，合成途中でも状態数は N_s の数倍を越えないと考えられるので，1 回のステップ 2，4 の実行に必要な時間 T_c は，

$$T_c \propto N_s^2 \cdot (N_i + N)$$

となる．ゆえに全体に対する合成時間 T_s は，

$$T_s \propto N_r \cdot T_c \propto N_o \cdot N_s^2 \cdot (N_i + N_o)$$

となり，多項式時間で処理できる．

7. むすび

時相論理を用いたハードウェア同期部の仕様記述法，および，処理系に Prolog を用いた効率的な状態遷移表現への展開法について述べた．内部状態を表すような変数を仕様記述に用いることにより，仕様は簡単な時相論理の条件式の積で表現することができ仕様が記述しやすくなり，また理解しやすくなるとともに，合成に要する時間を低く抑えることができる．また，Prolog のもつ強力なパターン照合機構と自動バックトラック機構により，合成アルゴリズムを容易に実装することができる．合成時間についても階層・構造化設計と組み合わせ，合成するモジュールの大きさを小さくしておくことにより，十分実用的な時間にすることができる．

また，矛盾を含む仕様に対し，具体的に矛盾が起こる各信号値のシーケンスを示し，設計者に対し対話的に仕様の修正を支援することができる．

時相論理で記述されたモジュールが別の時相論理の仕様を満たすか否かの検証を行うこともでき，ゲート回路，状態遷移図に対する検証手法[4)-6)]と組み合わせ

ることにより，仕様記述からゲート回路まで一貫して支援するシステムを Prolog を用いて構築することができる．

参考文献

1) Manna, Z.: Verification of Sequential Programs: Temporal Axiomatization, Dept. of Computer Science, Stanford University, Report No. STAN-CS-81-877 (1981).
2) Clocksin, W.F. and Mellish, C.S.: *Programming in Prolog*, Springer-Verlag, New York (1981).
3) 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).
4) 藤田，田中，元岡：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，情報処理学会論文誌，Vol. 25, No. 2, pp. 173-179 (1984).
5) 藤田，田中，元岡：ハードウェア状態遷移表現の Prolog による検証，情報処理学会論文誌，Vol. 25, No. 4, pp. 647-654 (1984).
6) Fujita, M., Nishiyama, S., Tanaka, H. and Moto-oka, T.: Efficient Verification Methods for Hardware Logic Design and Their Implementation with Prolog, Proc. of Logic Programming Conference '84, Tokyo (1984).
7) Wolper, P.: Synthesis of Communicating Processes from Temporal Logic Specifications, Dept. of Computer Science, Stanford University, Report No. STAN-CS-82-925 (1982).
8) Clarke, E.M. and Emerson, E.A.: Design and Synthesis of Synchronization Skeltons Using Branching Time Temporal Logic, Harvard University, Report No. TR-12-81 (1981).
9) Pereira, F.: *C-Prolog User's Manual*, Version 1.2a, Ed CAAD, Edinburgh (1983).
10) Moszkowski, B.: Resoning about Digital Circuit, Dept. of Computer Science, Stanford University, Report No. STAN-CS-83-970 (1983).
11) 後藤：Highly Parallel Inference Engine Based on Goal Rewriting Model: PIE，東京大学大学院工学系研究科情報工学専門課程学位請求論文 (1983).
12) 湯原：高並列推論エンジン PIE の単一化プロセッサ，東京大学大学院工学系研究科情報工学専門課程修士論文 (1983).
13) 藤田，田中，元岡：時相論理を用いたハードウェア同期部の記述と状態遷移表の自動合成，電子通信学会電子計算機研究会資料 EC 83-59 (1984).

(昭和59年4月23日受付)
(昭和59年6月19日採録)