EC83－59

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

Specifying Synchronization Parts of Hardware with Temporal Logic and Automatic Synthesis of State Diagrams

藤田 昌宏　田中 英彦　元岡 達

Masahiro Fujita, Hidehiko Tanaka, Tohru Moto-oka

（東京大学　工学部）

(Faculty of Engineering, University of Tokyo)

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益々大規模・複雑なものとなってきており、また、計算機が社会に占める比重が大きくなるにつれ、その信頼性が特に重要な問題となっている。

このため、大規模・複雑なシステムの設計を短期間に誤りなく行えるような手法・手段が必要不可欠となってきた。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するツールが強く望まれている。

そこで我々は、時相論理（Temporal Logic）［1、2］を用いて仕様を記述し、処理系にProlog ［3］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［4、5］、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［6］の記述に対する具体的手法を示した［7～10］。検証対象をシステムの状態を制御する同期部に絞り、各種の手法を用いて効率化［9、10］を行ないながら、必要な全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、誤設計をおこしやすいところであり、検証を特に必要としている。

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の自動合成について述べる。状態遷移表は、時相論理の決定手続き［2］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で合成することできる。ここでは、Prolog を用いてこれを効率的に実装する手法を示すと共に、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すか否かの検証を行なうこともできる。したがって、文献［6、7］と組合せることにより、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3章のアルゴリズムをProlog で効率的に実装するための方法について述べ、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なった例を示す。6章では時相論理の記述の検証に応用した例を示す。そして、7章で処理能力や人手の設計との比較等の考察を行ない、実用性について検討し、最後に8章で結論を述べる。

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 ［3］であるC－Prolog ［11］を用いる。

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・算術演算を行なう演算部と、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［8］。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、設計支援が特に望まれる。

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、2］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4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： 現在から将来の少くとも1時刻でPが成り立つ

P U Q： 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[2]を用いる。)

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、

□(P→○Q) …①

と表現できる。また、具体的に時刻は言えないが、将来少くともQがactiveになる場合は○を▽にかえて次のように表現できる。

□(P→▽Q) …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にへ変化するのは、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。

□(~Q→((○~Q) U P)) …③

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

条件②について同じことを言うには、次を付け加えればよい。

□(~Q→(~Q U P)) …④

信号の因果関係の表現は、①と③、又は、②と④を合せたものが基本となる。また、④を少し変形して、『最初にPがactiveになるまでは、一旦Qがinactiveになるとinactiveのままである』は、次のようにすればよい。

(~Q→(~Q U P)) U P …⑤

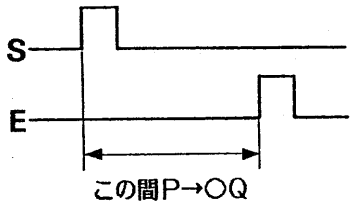


図1 タイムチャートの例

⑤を拡張すると、例えば図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。

□(S→((P→○Q) U E)) …⑥

[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より先にくるとする。このことは、次のように時相論理で表現できる。

□(S→((○~S) U E)),
(○~E) U S,
□(E→((○~E) U S))]

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)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。

□(~I→((~I) U S))),
□(S→I),
□(I→(I U E)), …⑦
□(E→(~I)),
□((I∧P)→○Q)

これは、最初4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、最後の条件でその時に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⑦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わすものであるが、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、⑦から分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。

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。我々の研究室では高並列推論マシンPIE[12]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(以下UPと略す)[13]であり、現在試作が進められている。試作UP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4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1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く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なう部分を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ある）。

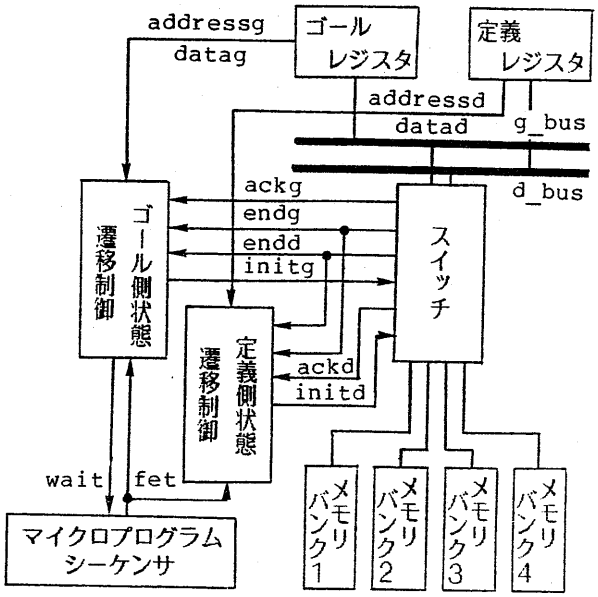


図2　UPの構成

まず、仕様記述に用いる各信号の意味を説明する。fet 、wait、initg 、endg、ackg、initd 、endd、ackdは全て1ビットの信号であり、これら以外は複数のビット幅をもつ信号である。

<共通>

fet ：　メモリに対し連続して参照を行なうマイクロ命令（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

<ゴール側>

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

addressg：　ゴールレジスタからのメモリ参照アドレス

datag ：　ゴールレジスタの内容

<定義側>

initd ：　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

endd：　定義側のメモリ連続参照が終了したか否かを示す信号

ackd：　定義側が実際にアクセス権を得たか否かを示す信号

addressd：　定義レジスタからのメモリ参照アドレス

datad ：　定義レジスタの内容

また、if-then-elseを次のように定義する。

if P then Q else R ≡
　□((P→Q)∧(～P→R))

まずスイッチの部分の仕様を考える。アクセスが競合した場合にはゴールレジスタを優先するのでゴール側の仕様は簡単になる。

（S－1）ゴールレジスタの内容がメモリへのアクセスを要求しているか、または最初のアクセスであればackgをactiveに、さもなければinactiveにする。

if (datag requests access-to-memory) ∨initg
　then ackg else ～ackg ;

ただし（datag requests access-to-memory ）は、datag の値がメモリバンクへの参照を要求しているか否かを示すプレディケイトである。実際のハードウェアでは適当な組合せ回路で実現されているが、ここでは簡単のためこのように記述することにする。

（S－2）定義レジスタに関しては、ゴールレジスタと競合しない限りアクセスが許される。すなわち、ゴール側が参照していないか、または、両者の参照するメモリバンクが異なる時に許される。

if ((datad requests access-to-memory) ∨initd) ∧
(～ackg∨ ((bank-of addressg) ≠ (bank-of addressd))
　then ackd else ～ackd ;

（S－3）ゴールレジスタの内容がメモリ参照の終了を示したならば、endgをactiveに、さもなければinactiveにする。さらに、一旦endgになると、次のメモリ連続参照命令の最初のアクセスになるまでendgのままである。

□(～(datag requests access-to-memory) →endg),
□(～endg→
(～endg U
　(～(datag requests access-to-memory))))
□(initg →～endg),
□(endg→(endg U (initg ∨～fet)))

（S－4）定義レジスタについても同様。

□(～(datad requests access-to-memory) →endd),
□(～endd→
(～endd U
　(～(datad requests access-to-memory))))
□(initd →～endd),
□(endd→(endd U (initd ∨～fet)))

この他スイッチ部は、addressgやaddressdの値によりレジスタ・メモリバンク間のスイッチ切り換えを行なうが、これは演算部に相当するのでここでは省略する。

次にゴール側のアクセスに関する状態遷移を制御する部分（ゴール側状態遷移制御）の仕様を考える。入力信号としてackg、endg、endd、fet を受け取り、信号initg 、waitを制御する。

（G－1）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。

□（（initg ∧fet ∧ackg）→（○～initg ））,
□（initg →（（○initg ） U （fet ∧ackg））

（G－2）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 ）の時に限る。

□（（initg ∧fet ）→wait）,
□（～wait→（（～wait） U （initg ∧fet ））

（G－3）シーケンサに対しwaitを終了するのは、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 ）時に限る。

□（（～fet ∨（fet ∧endg∧endd∧～initg ））
→～wait）,
□（wait→
（wait U （～fet
∨（fet ∧endg∧endd∧～initg ））））

（G－4）最初のアクセス状態へもどるのも③と同じ条件である。

□（（～fet ∨（fet ∧endg∧endd∧～initg ））
→（○initg ））,
□（～initg →
（（○～initg ） U （～fet
∨（fet ∧endg∧endd∧～initg ））））

定義側状態遷移制御部は入力信号として、ackd、endg、endd、fet を受け取り、initd を制御する。ゴール側と同じように記述でき、次のようになる。

（D－1）

□（（initd ∧fet ∧ackd）→（○～initd ））,
□（initd →（（○initd ） U （fet ∧ackd））

（D－4）

□（（～fet ∨（fet ∧endg∧endd∧～initd ））
→（○initd ））,
□（～initd →
（（○～initd ） U （～fet
∨（fet ∧endg∧endd∧～initd ））））

シーケンサに対し、waitを要求する条件もゴール側と同じであり、次のようになる。しかし、wait信号の制御はゴール側に任せる（G－2、G－3）ことにし、ゴール側定義側合せた全体として上の以下の仕様を満していることを後で検証する。

（D－2）

□（（initd ∧fet ）→wait）,
□（～wait→（～wait U （initd ∧fet ））

（D－3）

□（（～fet ∨（fet ∧endg∧endd∧～initd ））
→～wait）,
□（wait→（wait U （～fet
∨（fet ∧endg∧endd∧～initd ））））

以上のようにしてハードウェア同期部を記述することができる。前述のように、2．1の①と③や②と④の組が仕様記述の基本となっている。

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［2］を参照されたい。文献では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）に適用する。

合成は次のようにして行なえる。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。つぎに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。このような展開を既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全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は時相論理の公理から得られるもので例えば、＜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。

展開過程に現れる全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実際にeventuality が満されているか否か調べ、もし満されない状態遷移があればこれを削除する。

<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})

表1 時相演算子の展開規則 ただし、P、P1、P2は任意の時相論理の式

例として、□(P→▽Q)を展開してみる。まず表1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}))

したがって次の時刻に対する条件は、現在に対する条件に対して次のように決まる。

□(P→▽Q)の展開

現在の条件	次以降の条件	
~P∨Q	□(P→▽Q)	…⑧
P∧~Q	▽Q∧□(P→▽Q) {Q}	…⑨

⑧は最初の条件と同じである。次に⑨を展開する。

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

▽Q∧□(P→▽Q) {Q}の展開

現在の条件	次以降の条件	
Q	□(P→▽Q)	…⑧
~Q	▽Q∧□(P→▽Q) {Q}	…⑨

これらはいずれも今までに現れてきた条件と同じである。以上から図3のような状態遷移図が得られる。図3の状態2で*の遷移を取り続けると {Q} のeventuality が満されない。そこで自動合成では遷移④を削除し、図4のような状態遷移図を生成するようにする。

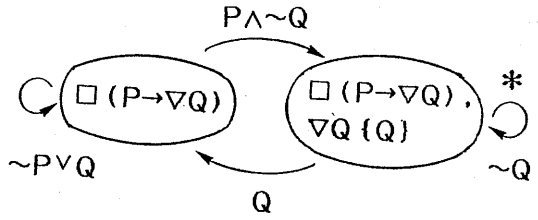


図3 決定手続きによる□(P→▽Q)の展開

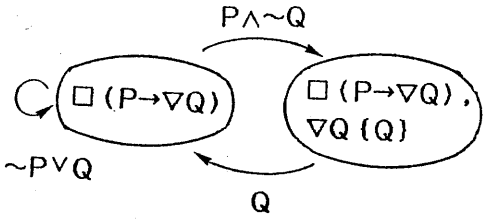


図4 図3からeventiality を満さない遷移の削除

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効率よくProlog で実装する手法について説明する。なお処理系にはEdinburgh大学で開発されたC−Prolog を用いる。

<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]
<7> p=[val , p] ただし、p は任意の命題変数

表2 時相論理の式のリストによる表現

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 のうち、表1の<1>、<2>に相当する部分は図5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2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り、次の形をしている。

P＝［［変数1，値1］，［変数2，値2］，…，
［変数n，値n］］

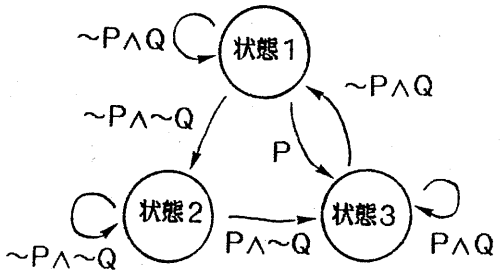
最初、値i は変数になっており、展開していく過程でmember（［変数i，1］，P）やmember（［変数i，0］，P）により1か0に設定される。このdevelop に対し強制的にバックトラックをかけることにより、次の時刻以降に対する条件を全て求めることができる。したがって3章に示したアルゴリズムにしたがって処理すれば状態遷移表現を得ることができる。このようにProlog のもつ強力なパターン照合機構や自動バックトラック機構により、極めて簡単に処理プログラムを作成することができる。

```
                        現在と次のeventuality
                                    現在の各変数の
      現在の条件   次以降の条件          値のリスト
develop([alw,F],[[alw,F]|Fnn],E,En,P):-
        if_exist_delete(F,E,El),   Eの中にFがあればそれ
        develop(F,Fn,El,En,P),     を削除してE1とする
        simplify(Fn,Fnn).  ←
develop([eve,F],Fn,E,En,P):-          式の簡単化
        if_exist_delete(F,E,El),
        develop(F,Fn,El,En,P).
develop([eve,F],[[eve,F]|Fnn],E,[Ff|Enl],P):-
        simplify(F,Ff),
        if_exist_delete([not,F],E,El),
        develop([not,F],Fn,El,Enl,P),
        simplify(Fn,Fnn).
```

図5　表1のProlog による記述：develop

またeventuality については、ループになった状態遷移列に対して、状態を順に調べていき各eventuality が満されているか否かを調べる。このため、その状態遷移表現に現れる全てのeventuality の数と同じ長さのリストを用意し、このリストの各要素を各々eventuality に対応させ、もしその状態でそのeventuality があれば1、なければ0とする。このようにしておけば、『ループとなった状態遷移列の全ての状態のeventuality のリストにおいて、もしある要素がどのリストにおいても1である場合にはそのeventuality が満されない』ことになり、各ループとなった状態遷移列に対し、eventuality が満されているか否かを容易に調べることができる。

例えば、2．1の①∧③を展開すると図6のようになる。



注：この場合にはeventuality はない

図6　□（P→○Q）∧□（~Q→（○~Q）　U　P）の展開

以上のような方法により、任意の時相論理による仕様を状態遷移表現に展開することができる。しかし、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。例えば、2．2の（G－1）や（G－4）はいずれも2．1の①や③の形をしている。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合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、2章でみてきたように時相論理の式T1 、T2 、…、Tn に対し、

T＝T1 ∧T2 ∧…∧Tn　…（＊）

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、T1 、…、Tn の各々の状態遷移表現をまとめ、可能な全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分かる。

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なうことができる。

＜合成アルゴリズム＞

（ステップ1）　仕様は（＊）の形で記述されているものとする。まず、各Ti は図5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Ti に対する状態遷移表現から図7にしたがってTに対する状態遷移表現を作る。

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

（ステップ4）　同じ処理をしている状態を1つにまとめる。

（ステップ5）　各遷移条件の論理式を簡単化する。

ステップ4及びステップ5は、合成された状態遷移表現の冗長性を除き簡単化するためのものである。

```
procedure synthesis ;
var L: set of state-set of all Ti ;
   s,s': set of state ;
 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7　個々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プロシージャ

実際の仕様記述では、まず初めにおおまかな仕様：

Told ＝T1 ∧…∧Tn

を考え、その後新たな条件（Tnew1∧…∧Tnewm）を付け加

えて最終的な仕様：

Tnew ＝Told ∧（Tnew1∧…∧Tnewm）

とすることも多い。この過程はアルゴリズム1を用いて次のように効率的に支援できる。

まず各T1 、…、Tn をステップ1に従い展開して状態遷移表現にし、ステップ2によりTold の状態遷移表現を作る。次に再びステップ1に従いTnew1、…、Tnewmを展開して状態遷移表現とし、Told の状態遷移表現とあわせてステップ2によりTnew をの状態遷移表現を作成する。

ステップ2で用いる状態遷移表現は、DDLの設計記述から得られるものでもよい。したがって、既にDDLで設計されたもの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ができる。

5. 合成例

本章では2．2で示したUPのゴール及び定義状態遷移制御回路を実際に合成した結果を示す。合成の手順は以下の通りである。

・ゴール側の合成

（1） 仕様は（G−1）∧（G−2）∧（G−3）∧（G−4）であり、（G−1）、（G−4）はともに

□（A→○B）∧□（～B→（（○～B） U A）

（ただし、A、Bには時相演算子はない） …＊1

の形をしており、また（G−2）、（G−3）は

□（A→B）∧□（～B→（（～B） U A）

（ただし、A、Bには時相演算子はない） …＊2

の形をしているので、合成ステップ1に従い、＊1、＊2を展開し、状態遷移表現に直す。

（2） 合成のステップ2を実行する。まず＊1で、A＝initg ∧fet ∧ackg、B＝～initg とすれば、条件式（G−1）となり、A＝～fet ∨（fet ∧endg∧endd∧～initg ）、B＝～initg とすれば条件式（G−4）となる。また＊2で、A＝initg ∧fet 、B＝～waitとすれば条件式（G−2）となり、A＝～fet ∨（fet ∧endg∧endd∧～initg ）とすれば条件式（G−3）となる。さらに、これだけでは仕様自体に矛盾が生じるため、環境に関する条件として外部信号endgに対し、（S−3）の中の

□（endg→（endg U （initg ∨～fet ））

を加える。そして、これら全体を状態遷移させゴール側に対する状態遷移表現を作成する。

（3） ステップ4、5による冗長性除去を行なう。

・定義側についても同じようにして合成することができる。また、ゴール定義両方合せた全体に対する状態遷移表現もステップ2、3、4、5によって行なえる。

実際に自動合成した結果を図8に、各合成ステップにおける処理時間を表3に示す。使用した計算機はVAX11／ 730である。合成結果はゴール側、定義側状態遷移制御は人手による設計と同じであった。また、ゴール・定義両方合せたものは、人手では複雑で1つの状態遷移表に直すことができず、回路設計はゴール・定義別々に行なっている。このため、状態を表現するフリップフロップはゴール・定義それぞれ2個ずつ必要で合計4個となった。一方自動合成では、全体に対する状態遷移表は状態数が8個のためフリップフロップは3個で済むことになる。

表3から分かるように、合成に要する時間も短く、この程度の規模の合成には十分実用的であると言える。

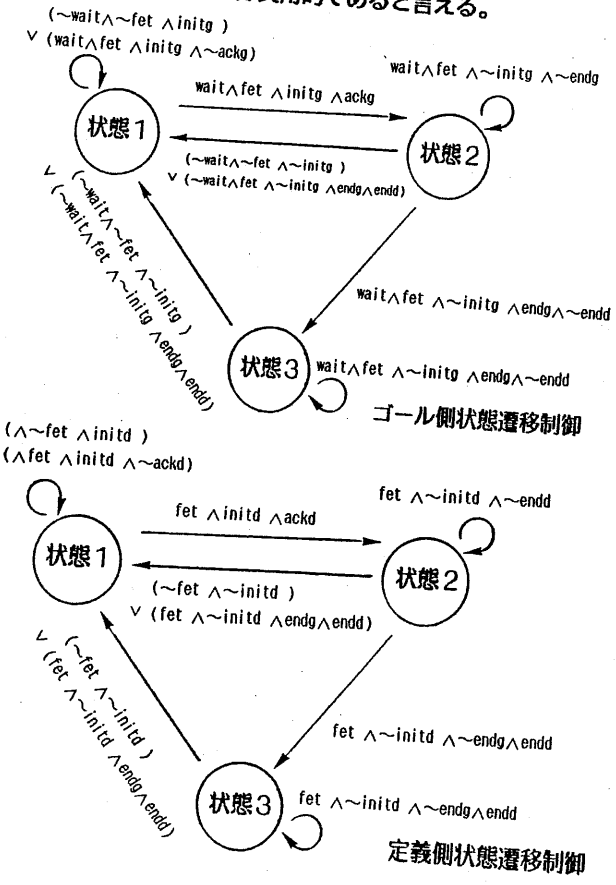


図8 合成結果

ステップ	ゴール側状態遷移制御の合成	定義側状態遷移制御の合成	2つをまとめた全体の合成
1	49．97	−	−
2、3	32．78	24．65	21．02
4、5	34．09	31．90	46．44

表3 図8の自動合成に要するCPU時間

VAX11／ 730・UNIX上のC−Prolog 、単位：秒

6. 検証例

本章では、時相論理で記述されたハードウェアに対する時相論理の仕様の検証について述べる。2．2．の（G－1～4）と（D－1）、（D－4）を合せたものが、（D－2）、（D－3）を満しているか否か、つまり、信号waitをゴール側のみで制御している設計となっているが、これで定義側のwaitに対する条件を満していることを検証することを考える。

検証は背理法、つまり、仕様の否定が満され得ないことを示すことで行なうが、次の2通りが考えられる。

（1） 5章で合成した状態遷移表現に対して、（D－2）、（D－3）を検証する。

（2） （G－1）∧（G－2）∧（G－3）∧（G－4）に仕様の否定を加えたものを展開し、どの状態遷移列も矛盾に達することを示すことで検証する。

（1）は、状態遷移図の検証であり、文献［8］と同じ手法でできる。しかし、一般には仕様は必ずしも状態遷移表現に展開されているわけではないので、ここでは（2）にしたがって検証することにした。

実際の検証では、（D－2）、（D－3）に現れる4つの式を順に調べていく。ゴール、定義側合せた仕様（G－1～4）、（D－1）、（D－4）に検証すべき仕様の否定を付け加えたもの全体を5章の合成の時と同じように展開していけばよい。検証の結果正しいことが分かり、要した時間は約50～100秒（VAX11／730・CPU時間）であった。

7. 考察・検討

7．1． 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、時相演算子が次々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．の⑦におけるIのように、それがactiveの時のみある関係が成立するような変数を内部状態を表わすものとして、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わ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わ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全て前もって展開してライブラリとして貯えておくことができる。このようにすれば、実際の合成は既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なうことができ、ライブラリが十分充実していれば、合成を円滑に行なうことができる。

7．2． 合成時間、及び、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と同じであり、十分実用的であると言える。

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、最悪指数的に増大する。

しかし、一般に仕様は4章の（＊）の形で記述されており、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、各Tiについては既に展開されたものを用いるため、各Tiの状態の数をNiとすれば合成処理時間Tsは、

$$Ts \propto \prod_{i=1}^{n} Ni$$

となる。

実際には各Ti ごとに展開されているわけではなく、いくつかまとめたものが既に展開されている場合もあり、Tsは使用する展開された状態遷移表現の数Ns（5章の例では、ゴール側状態遷移制御4、定義側状態遷移制御2）に対し、指数的に増大することになる。Nsは合成すべきモジュールが制御する信号の数Nmの数倍程度なので、うまく階層設計されており、Nmが小さければ十分実用的な時間で合成を行なうことができる。

一方、合成された状態遷移表現の簡単化（合成アルゴリズムのステップ4、5）は、もとの状態の数Noに対し、No程度の手間でできる。

以上から、うまく階層・構造化設計されたシステムに対しては、十分実用性を発揮することができる。

7．3． 検証について

6章で示したように、合成アルゴリズムを用いて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すか否かを検証することができる。したがって、状態遷移図、ゲート回路に対する検証手法と組合せて円滑に階層・構造化設計を支援することができる。

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、各信号間の立ち上り、立ち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時に、ある信号がある時は立ち上りある時は立ち下るような場合があると、その2つを区別するために別のもう1つの信号を加えるということを行なっている。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。もし全ての展開がうまくいったら終了する。

このように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。

8. 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、及び、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。内部状態を表わ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［14］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・構造化設計と組合せ、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、設計者に対し会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と組合せることにより、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。

9. 参考文献

［1］Z. Manna：“Verification of Sequential Programs： Temporal Axiomatization”, Dept. of Computer Science, Stanford University, Report No.STAN-CS-81-877, 1981

［2］P. Wolper：“Synthesis of Comminicating Processes from Temporal Logic Specifications”, Dept. of Computer Science, Stanford University, Report No.STAN-CS-82-925, 1982

［3］W. F. Clocksin, C. S. Mellish: “Programming in Prolog”, Springer-Verlag, 1981

［4］M. Fujita, H. Tanaka, T. Moto-oka：“Verification with Prolog and Temporal Logic”, Proc. of 6th IFIP CHDL, Pittsuburgh USA, 1983

［5］M. Fujita, H. Tanaka, T. Moto-oka：“Temporal Logic Based Hardware Description and its Verification with Prolog”, New Generation Computing, Vol. 1, No.2, pp.195-203, 1983

［6］J. R. Duley, D. L. Dietmeyer：“A Digital System Design Language（DDL）”, IEEE Trans. on Computer, Vol. C-17, pp.850-861, 1968

［7］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, 1984

［8］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、採録済み

［9］藤田、中田、田中、元岡：“時相論理を用いた論理設計検証システムの実際的仕様について”、電子通信学会電子計算機研究会資料、EC83-29、1983

［10］M. Fujita, S. Nishiyama, H. Tanaka, T. Moto-oka：“Efficient Verification Methods for Hardware Logic Design and their Implementation with Prolog”, Proc. of the Logic Programming Conference '84, Tokyo Japan, 1984

［11］F. Pereira：“C-Prolog User 's Manual Version1. 2a”

［12］後藤：“A Highly Prallel Inference Engine Based on Goal-Rewriting Model： PIE”、学位請求論文、東京大学大学院工学系研究科、情報工学専門課程、1983

［13］湯原：“高並列推論エンジンPIEの単一化プロセッサ”、修士論文、東京大学大学院工学系研究科、情報工学専門課程、1984

［14］B. Moszkowski：“Resoning about Digital Circuit”, Dept. of Computer Science, Stanford University, Report No.STAN-CS-83-970, 1983