時相論理に基づくハードウェア機能記述とそのインタプリタについて

Hardware Functional Description Based Upon Temporal Logic and its Interpreter

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）

時相論理とProlog を用いた、効率的なハードウェア論理設計の検証・合成システムを開発してきた[2]。そこでは、ハードウェアを同期部と演算部に分けて処理を行なっており、Interval と概念が処く効率化していた。ここではこれをふまえて、アルゴリズムの記述の段階から支援を行なうことができハードウェア機能記述言語を提案する。通常の時相論理にMoszkowski ら[17、18]が提案していval Temporal Logicの概念をもち込み、ハードウェア記述に必要な並列性・順序性共に容易に記述でを示す。本論文では主に、シミュレーション、設計検証、同期部の自動合成について考察する。

に

記述等の進歩により、ハードウェアシステムは複雑化している。論理設計段階の誤りは設計・きく影響を及ぼすため、実装段階のみではなく、計支援が特に重要になってきている。
・機能設計の支援はハードウェア記述言語を中われてきた[1]。支援ツールが比較的整ってェア記述言語としてISPS[2]やDDL、ISPSは主にCMUにおいて、シミュレースやゲート回路の自動合成等[4、5]の支援いる。一方、DDLも様々なところで、シミュ・結果の因果関係表へのトランスレータ、ゲー合成等[6～8]の支援が行なわれている。然言語等で記述された仕様から、これらのハー言語でいきなり記述するのはむずかしく、よりベルの言語が必要である。すなわち、仕様かSやDDL等の既存のハードウェア記述言語うな支援言語やツールが望まれる。特に設計のは、処理アルゴリズムの検証も必要であり、外アルゴリズム（内部仕様）も記述できるようなってきている。

仕様記述に時相論理（Temporal Logic）をにProlog を用いてハードウェア論理設計を効合成する手法を開発した[9～12]。そこでは、システムを同期部（コントロール部）と演算部（データパス）に分け、特に誤設計が起きやすいと考えられる同期部に的を絞っている。本論文では、これをふまえて、同期部、演算部まとめたシステム全体に対する仕様記述から論理設計にいたる円滑な支援について考える。

我々は、基本的に時相論理を用い、同期部の仕様記述は命題論理の範囲で、演算部の仕様記述は述語論理の範囲で行なうことで、効率よく検証できることを示している[13、14]。しかし、処理アルゴリズムの記述を行なうようなレベルでは、同期部、演算部を明確に分けて記述することは、簡単ではない。従って、両者合せた全体の記述から自動的に同期部に対する記述と演算部に対する記述を分けられるような言語が望まれる。また、従来のCADシステムとのつながりを考えて、仕様やアルゴリズムの記述を受け取り、最終的にDDLを出力するような言語・ツールを考えていく。

システムの処理アルゴリズムの記述においては、次の点が特に重要であると考える。

1）並列性の記述ができること。
2）ソフトウェアもハードウェアも共に記述可能であること。
3）外部仕様も記述可能であること。

現在このレベルの支援言語として考えられているのは、汎用の並行プロセス記述言語であり、具体的には、Ada、occam 、concurrent-prolog 等[15、16]である。このような汎用言語を用いる利点としては、デバッガ等の支援ツールが整っている点、またoccam ではtansputer によって高

速処理が可能である点［16］等である。逆に欠点としては、ハードウェアをどのように意識して記述するかは設計者（記述者）まかせである点、ハードウェアを考えた検証・合成は別に考えなければならない点、そして、従来のハードウェア記述言語とどのように結びつけていくかという点等である。

確かに支援ツールが整っていることは、大きな利点となるが、従来のハードウェア記述言語やCADシステムといかに円滑につなげていくかは重大な問題である。

一方、我々は時相論理による仕様記述を中心とした論理設計検証・合成システムを開発している。そこでは、Intervalという考え方が大きく効率や仕様記述のしやすさに貢献している［12］（2．参照）。時相論理は本質的に並列性の記述は容易であるが、逆に順序性（sequentiality）の記述は面倒なものとなる。しかし、Intervalという概念を用いると、容易に順序性も記述でき、アルゴリズムも円滑に記述できる。その上、時相論理の記述は、状態遷移表現に展開することができるため［11］、状態遷移に基づくハードウェア記述言語DDLと容易につなぐことができる。本論文では、このような考え方に立つ時相論理に基づくハードウェア機能記述言語を考え、シミュレーション、検証、合成についてその考え方を述べる。Moszkowskiらは、Intervalという考え方に基づく時相論理（Interval Temporal Logic、以下ITLと略す）［17、18］を提案し、幾つかのハードウェアが容易に記述できることを示している［17］。しかし、ITLには決定手続きが存在しないため［18］、計算機処理では不都合なことも多い。そこでここでは、通常の時相論理（Linear Time Temporal Logic、以下LTTLと略す）［19、20］を用いるが、ハードウェア機能記述を円滑に行なえるように、Intervalという概念を取り入れて記述する手法を示し、従来の我々の検証・合成手法とうまく結びつくことを示す。

まず2章で我々が既に示した時相論理（LTTL）を用いたハードウェア同期部の仕様記述［11、12］について述べる。3章では、Moszkowskiらが提案しているITLについて説明する。4章では、ITLの考え方を取り入れたLTTLによるハードウェア機能記述について説明し、検証・合成、シミュレーションが円滑に支援できることを示す。5章では、Prologによるシミュレータの実現方法について述べ、最後に6章で結論を述べる。

2．時相論理（LTTL）によるハードウェア仕様記述

2．1 時相論理

本節ではまず時相論理について簡単に述べ、次に検証や自動合成を効率よく行なえるようなハードウェア同期部の仕様記述法について説明する。なお、時相論理についての詳細は参考文献［19、20］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［20］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。）

信号の因果関係の表現は、①（又は②）と③を合せたものが基本となる。

各信号間の時間順序関係はU演算子を用いて表現できる。例えば、『Qよりも先にPがactiveになる』は、次のようになる。

～（～P　U　Q）　…④

また③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

2．2 タイムチャートの記述［11、12］

図1（a）に示すタイムチャートのように、『スタート信号S1とエンド信号E1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。

□（S1→（（P→○Q）　U　E1））　…⑥

（ただし、S1とE1は図1のようにパルス状に外部から与

えられ、かつS1 がE1 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。

□(S1 → ((○~S1) U E1)),
(○~E1) U S1 ,
□(E1 → ((○~E1) U S1)))

さらに (b) のように、スタート信号S1 とエンド信号E1 の間の時刻のさらにスタート信号S2 とエンド信号E2 の間の時刻においては、Pがactiveになると次の時刻Qがactiveになる』は次のようになる。

□(S1 → ((S2 → ((P→○Q) U E2))
U E1)) …⑦

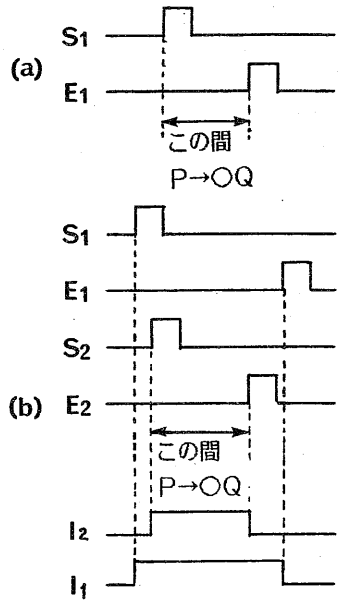

図1 タイムチャート

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S1 とエンド信号E1 の間の時刻 (interval) のみI1 という信号がactiveになり、I1 がactiveの時にスタート信号S2 とエンド信号E2 の間の時刻のみI2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。

□(~I1 → ((~I1) U S1)),
□(S1 → I1),
□(I1 → (I1 U E1)), …⑧
□(E1 → (~I1)),
□(~I2 → ((~I2) U (I1 ∧S2))),
□((I1 ∧S2) → I2),
□(I2 → (I2 U (I1 ∧E2))),
□((I1 ∧E2) → (~I2)),
□((I2 ∧P) →○Q)

これは最初の4つの条件で信号I1 がactiveになる時刻をS1 とE1 の間の時刻のみにし、次の4つの条件で信号I2 がactiveになる時刻をI1 がactiveな時刻内のさらにS2 とE2 の間の時刻のみにし、そして最後の条件でI2 がactiveな時に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⑧のように適当な時間の幅 (interval) (3. 参照) 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I1 やI2 は外部には関係ない内部状態を表わすものであるが、これらを導入することにより仕様が記述しやすくなる。また、⑧から分かるように同じ形の条件式が多く、各々は単純な式となるため、後で述べるように検証に要する時間を低く抑えることができる。

Interval の導入によって、順序性の記述も容易になる。このようなInterval を基礎とした時相論理も考えられており、次章で説明する。

3. Interval Temporal Logic [17、18]

Moszkowski らは、前章でできたInterval と呼ばれる時間の幅を基礎としたInterval Temporal Logicを考え、様々なハードウェア・ソフトウェアの記述例を示した [17、18]。本章では、ITLについて簡単に説明する。

ITLも通常の時相論理 (LTTL) と同じく離散時間 t0,t1,…,tn,…上で定義される。時刻tiにおける記述対象の状態をsiとすると、Interval は有限個の連続した状態 sk,sk , …,sl (k ≦l) として定義される。Interval の長さlen とは、

len ≡ l−k

として定義される [17]。(ここでは有限長のInterval のみ考える。)

ITLにおける変数は状態ごとではなく、Interval ごとにその真偽が決まる。ITLには、古典論理の演算子の他に、; (セミコロン) と○ (next) の2つの時相演算子がある。○はLTTLと同じ意味である。;演算子は、任意のInterval を前半と後半の2つのInterval に分ける。ただし、いつ前半と後半に分かれるかは問わない。

例えば、(P;Q) は、図2のように今考えているInterval の前半でPが成立し、後半でQが成立することを示している。

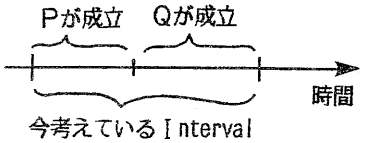

図2 Interval の分割

LTTLの▽演算子は、

$\nabla P \equiv (true ; P)$

と表現できる。同様に□や∪演算子も表現できる。

また、beg 及び、fin でそれぞれ今考えているＩnterval の最初と最後を示す。今、長さ0のＩnterval （状態1つだけ）としてempty

$empty \equiv \sim\bigcirc true$

を考えると、

$beg(P) \equiv (empty \wedge P) ; true$

$fin(P) \equiv true ; (empty \wedge P)$

と表現できる。

実際のソフトウェア・ハードウェアの記述においては、『各Ｉnterval の最初での値を用いて計算を行ない、そのＩnterval の最後での値とする』ということがよく現れる。これは、1階のＩＴＬで次のように表現できる。

$B \leftarrow A \equiv \forall c.\ (\ beg(A=c) \rightarrow fin(B=c)\ )$

B←AはそのＩnterval の最初でのBの値がそのＩnterval の最後でのAの値となることを示している。

例えば、『変数Xの値を2増す』は、

X←X＋2

と記述できる。これを実際にはハードウェアの都合等により＋1を2回行なって実現するのであれば、次のように記述できる。

（X←X＋1）；（X←X＋1）

これは、＋2を実行するのに、前半で＋1を実行し、後半でもう1回＋1を実行することを意味している。このように順序性も容易に記述できる。

文献［17］では、ハードウェアの様々な記述例、delay element からＡＬＵまでが示されている。また文献［18］では、quick sort等のソフトウェアの記述例が示されている。

このようにＩＴＬを用いて容易に様々なものを記述することができる。しかし、ＩＴＬは命題論理の範囲でもその充足可能性に対する決定手続きが存在しないことことが示されている［17］。文献［17］では決定手続きが存在するようなＩＴＬのサブセットが示され、それをＱＴＬ（ＬＴＴＬにおいて、各変数にquantificationを付けることを認めたもの）への変換法が示されている。

ＩＴＬは確かに記述能力は高いが、その非決定性ゆえ、計算機による自動処理は簡単ではない。

4．　時相論理によるハードウェア機能記述

4．1　ハードウェア機能記述言語

本章では、時相論理（ＬＴＴＬ）に基づく、Ｉnterval の概念を取り入れたハードウェア機能記述について述べる。

ハードウェア機能記述に必要なこととして、次のことがあげられる。

1）アルゴリズムを容易に記述できること、つまり、並列性、順序性（sequentiality ）を平易に表現できること。

2）数学的モデルがはっきりしていること。

3）階層設計や既存のＣＡＤとの相性がよいこと。

1）は、ハードウェアのような並列動作するものを記述する際には必要であり、2）は、計算機による検証・合成を行なうためには、是非必要である。また、3）は、実際上の運用を考えると必要となる。

階層設計では、抽象的なものから、具体的なものへ設計を詳細化していくが、具体化するものは、構造（structure ）と動作（behavior）の2種類に分けられる。構造の具体化とは、例えば、『上位レベルでは1本の線で表現されていたデータパスが、実は8ビット幅である』とか、『全加算器は、2つの半加算器と1つのＯＲゲートからできている』等である。また、動作の詳細化とは、例えば、『上位レベルでは、X←X＋2で表現されていたものが、実はX←X＋1を2回で実現する』等である。本論文では、主として、動作の詳細化について考える。

一方、ハードウェア機能記述に対し、計算機支援すべきものとしては、次のことが考えられる。

1）シミュレーション

2）合成（詳細化）

3）検証

以上のことを念頭において、以下時相論理に基づくハードウェア機能記述言語を導入する。

ＬＴＴＬの記述は決定手続きにより、状態遷移表現の自動合成が可能であり［20］、記述の仕方を工夫することにより、かなり効率化することができる［11］。また、各種工夫を行なうことにより、検証も効率よくできる。しかし、順序性の記述は容易ではない。一方、ＩＴＬを用いれば、ソフトウェアもハードウェアもかなり自由に表現できるが、決定手続きが存在しないため、その計算機支援は簡単ではない。しかし、Ｉnterval という考え方は、2．2でも示したようにＬＴＴＬの中に取り込むことができ、このようにすることで、並列性のみではなく順序性も容易にＬＴＴＬで記述できるようになる。そこでここでは、ＬＴＴＬを用いて、Ｉnterval を表わす変数に基づく記述を考えていく。

まず、Ｉnterval を表わす変数を2．2のＩ1 やＩ2 のように、□、▽、○、∪で記述された時相論理の式の有効範囲を示すものとして導入する。そのＩnterval の記述の範囲を｛と｝で囲み、その前にＩnterval 変数を記述する。例えば、Ｉnterval ：int でP→○Qが成立するのであれば、

int (n,m) ｛P→○Q｝　…⑨

とする。ここで、(n,m) は、そのＩnterval の長さを示すものであり、int は、 nクロック以上 mクロック以下の長さであることを示す。本論文では、Ｉnterval の長さは全て、1以上有限であるとする。従って、長さの記述を省略した時は、1以上有限であるとみなす。

```
<MODULE> unification_processor ;

<VAR>
 <EXTERNAL>
  <REGISTER>  g_addr, d_addr, return_code, run ;
  <STORAGE>   memory ;
 <INTERNAL>
  <REGISTER>  length, g_mem, d_mem, g_cell, d_cell, stack_depth ;
  <STORAGE>   stack_ga, stack_da, stack_la ;
  <INTERVAL>  init, loop_unif, fetch_unif1, fetch_unif2, fetch_unif0g,
              fetch_unif1g, fetch_unif2g, fetch_unif0d, fetch_unif1d,
              fetch_unif2d, idle ;

<SPEC>

 init{
       length ← memory(g_addr) - 1
      ∧ d_addr ← d_addr + 2
      ∧ g_addr ← g_addr + 2
      ∧ stack_depth ← 0
      ∧ run ← 1
      ∧ GOTO loop_unif }

 loop_unif{
      IF (beg(length) > 0) THEN GOTO fetch_unif1,
      ELSE IF (beg(stack_depth)=0) THEN (return_code ← 'SUCCESS'
                                        ∧ run ← 0
                                        ∧ GOTO idle)
      ELSE (length ← stack_ln(stack_depth-1)
         ∧ g_addr ← stack_ga(stack_depth-1)
         ∧ d_addr ← stack_da(stack_depth-1)
         ∧ stack_depth ← stack_depth-1
         ∧ GOTO fetch_unif1 )}

fetch_unif1{
      length ← length - 1
     ∧ g_addr ← g_addr + 1
     ∧ d_addr ← d_addr + 1
     ∧ (fetch_unif0g(1,_){
           g_cell ← memory(g_addr)
           ∧ g_mem ← g_addr
           ∧ IF(fin(g_cell.tag)='VAR'))
              THEN GOTO fetch_unif1g
              ELSE GOTO fetch_unif2g }
       fetch_unif1g{
           g_cell ← memory(g_cell)
           ∧ g_mem ← g_cell
           ∧ IF(fin(g_cell.tag)='VAR'))
              THEN GOTO fetch_unif1g })
    ∧ (fetch_unif0d{
           d_cell ← memory(d_addr)
           ∧ d_mem ← d_addr
           ∧ IF(fin(d_cell.tag)='VAR'))
              THEN GOTO fetch_unif1d
              ELSE GOTO fetch_unif2d }
       fetch_unif1d{
           d_cell ← memory(d_cell)
           ∧ d_mem ← d_cell
           ∧ IF(fin(d_cell.tag)='VAR'))
              THEN GOTO fetch_unif1d })}

 fetch_unif2{
      (IF (g_mem = d_mem) THEN GOTO loop_unif
      ELSE CASE (g_cell.tag , d_cell.tag) OF
       ('UNDEF','UNDEF'): (memory(beg(g_mem)) ← d_mem
                          ∧ GOTO loop_unif),
       ('UNDEF',~'UNDEF'): (memory(beg(g_mem)) ← d_cell
                          ∧ GOTO loop_unif),
       (~'UNDEF','UNDEF'): (memory(beg(d_mem)) ← g_cell
                          ∧ GOTO loop_unif),
       ('LIST','LIST'): (IF (beg(length)>0) THEN
                          (stack_depth ← stack_depth + 1
                          ∧ stack_ga(beg(stack_depth)) ← g_addr
                          ∧ stack_da(beg(stack_depth)) ← d_addr
                          ∧ stack_ln(beg(stack_depth)) ← length)
                         ∧ length ← 2
                         ∧ g_addr ← g_cell
                         ∧ d_addr ← d_cell
                         ∧ GOTO fetch_unif1 ),
       ('ATOM','ATOM'): IF ( beg(g_cell.tag) ≠ beg(d_cell.tag)
                          ∨ beg(g_cell.data) ≠ beg(d_call.data))
                         THEN (return_code ← 'FAIL'
                             ∧ run ← 0
                             ∧ GOTO fail )
                         ELSE GOTO loop_unif
       OTHERWISE: (return_code ← 'FAIL'
                 ∧ run ← 0
                 ∧ GOTO idle )}

 idle(1,1){
      IF (beg(run)=1)
         THEN GOTO init
         ELSE GOTO idle }
```

図4　UP[22]のユニフィケーション・アルゴリズムの記述

⑨は次のＬＴＴＬの式と同じであるとする。

□（int →（P→○Q））　…⑩

ＩＴＬと同様、任意のＩnterval は任意個のサブＩnterval に分けることができる。Ｉnterval ：int 内では２つの並列なＩnterval ：sub1とsub2があり、sub1ではP→○Qが成立し、sub2ではR→○Sが成立するとすると、次のように記述できる。

int {sub1{P→○Q}∧sub2{R→○S}}　…⑪

これは、図３の（a ）に対応する。また、sub1とsub2に順序性があり、前半がsub1で後半がsub2とすると、次のようになる。

int {sub {P→○Q} sub2{R→○S}}　…⑫

（⑪と比べて、∧がなくなっている）

これは、図３の（b ）に対応する。

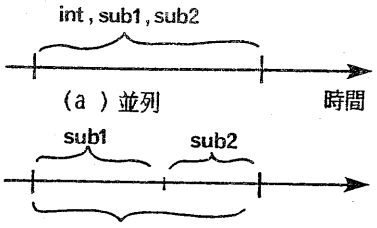

図３　並列性と順序性

ＩＴＬと同じように、各Ｉnterval には、その最初と最後を示す begと finが定義される。Ｉnterval をint とすると、int.beg 、int.fin と表現する。正しくＩnterval を表わすため、int 、int.beg 、int.fin は、次の条件を満すように制御される。

□（int.beg →　…⑬

（（int ∧○～int.beg ）　U　（int ∧○int.fin ））

また、Ｉnterval の長さが有限であることから、次が仮定される。

□（int.beg →▽int.fin ）　…⑭

また、陽にＩnterval の長さが示されている時、すなわち int（n,m ）{…} となっている時は、

□（int.beg → n▽m int.fin ）　…⑮

（ただし、n▽m P≡⓪P∨①P∨…∨⑪P

$$⓪P \equiv \overbrace{○\cdots○}^{n\ コ} P$$

とする。）

beg とfin を用いることにより、図３に示されている動作はＬＴＴＬの簡単な式のＡＮＤで表現できる。

例えば、（a ）は、次のようになる。

□（int.beg ⇔（sub1.beg∧sub2.beg））,

□（sub1.fin⇔（sub2.fin∧int.fin ））,　…⑯

□（sub2.fin⇔（sub1.fin∧int.fin ））

（ただし、A⇔B≡（A→B）∧（B→A））

また、（b ）は、次のようになる。

□（int.beg ⇔sub1.beg）,　…⑰

□（sub1.fin⇔sub2.beg）,

□（sub2.fin⇔int.fin ）

このように並列性、順序性共に簡単な式のＡＮＤで表現できる。

各Ｉnterval には、名前がついているので、ＧＯＴＯ文によるジャンプを認める。

int1{GOTO int2 }

は次を意味する。

□（int1.fin→int2.beg）

この他に、ＩＴＬの←はそのまま導入する。

また、ＩＴＬでは各変数は何も記述されていなければ、その値は全て不定となる。これでは実際にハードウェアの記述を行なうには不便なので、変数をＤＤＬと同じようにregisterとterminalに分け、register宣言されたものについては、何も記述されていなければ値が変化しないものとして扱うようにする。

以上が現在考えているハードウェア機能記述言語の概要である。各Ｉnterval の長さを全て１に固定するとＤＤＬと同じになる。つまり、本言語はＤＤＬにおいて、各状態にいる時間を不定として拡張したものとも考えられる。

高並列推論エンジンＰＩＥ［２１］のユニフィケーションを実行するＵＰ（Unify　Proceesor）のユニフィケーション・アルゴリズム［２２］を記述した例を図４に示す。図において、特にことわりがない限り、←の左辺はそのＩnterval の最後での値を指し、右辺は最初での値を指すものとする。定義・ゴール共にmemoryの中に入っており、ユニフィケーション結果もmemoryに入る。

４．２　設計支援

支援すべきこととして、次のことがあげられる。

１）シミュレーション

２）検証

３）合成

１）については、Prolog を用いたシミュレータについて次章で説明する。２）は下位レベルの記述が上位レベルの記述を満すことを確認することであり、３）は上位レベルの記述から下位レベルの記述を自動的に合成することである。検証においては、レベルの異なる記述は、共に時相論理で記述されている場合もあるし、また、片方は時相論理であり、もう一方はＤＤＬやゲート回路である場合もある。図３の例では、いずれの場合も検証すべきことは、（a ）なら⑯と

□（（sub1.beg∧P）→○Q）,

□（（sub2.beg∧R）→○S）　…⑱

であり、（b ）なら⑰と⑱となり、いずれの場合も簡単な時相論理の式のＡＮＤで表現できるため効率的に検証を行なうことができる［１２］。ただし、sub1.begやsub2.beg等の信号が下位レベルの記述（ＤＤＬ、ゲート）でどのように表現されているかを設計者が示す必要がある。

また合成については、時相論理のアルゴリズムの記述から直接ＤＤＬ等へ自動合成することが望ましいが、現状では最適なものを得るのは困難である。そこで、データパス（アーキテクチャ）を設計者が与え、その上で必要な制御信号（同期部）に対する状態遷移表を自動合成することを考える。上で述べたように時相論理の記述は簡単な式のＡＮＤで表現されるため、状態遷移表の自動合成は効率的に行なえる［１２］。こごては、単なる自動合成ではなく、データパスの情報をconstraints として受け取り、設計に対する知識を用いて、その上で実行できるように合成を行なう。

例えば、図４のＩnterval ：int において、図５のようなデータパスが与えられた場合には、次のようにＩnterval を分けて、＋２を＋１・２回で実行するように合成する。

```
int1 ( 1,1) {length←memory (g-addr)
            ∧d-addr←d-addr＋１
            ∧g-addr←g-addr＋１
            ∧stack-depth ←０
            ∧run ←１  }
int2 ( 1,1) {length←length－１        …⑲
            ∧d-addr←d-addr＋１
            ∧g-addr←g-addr＋１}
```

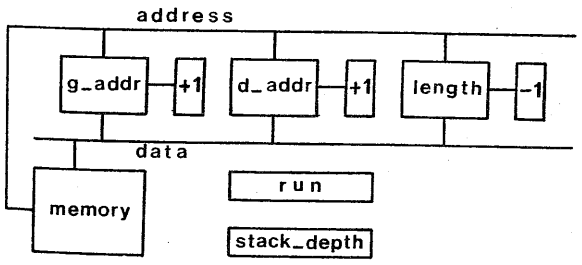

図５　設計者により与えられたデータパスの例

データパスを自動合成することも考えられるが、最適なものを合成するのはむずかしい。また、この辺は設計者の腕のみせどころでもあるため、設計者が自分で決定することとし、同期信号を自動的に合成するようにするのが、実際上実用的だと考える。将来的には、設計者の知識を計算機に蓄えて、データパスの自動合成へと発展することもできる。

以上がハードウェア機能設計に対する基本的な支援姿勢である。従来我々が示した検証・合成手法［９～１２］をうまく活用することができる。現在Ｅdinburgh大学で開発されたＣ－Prolog ［２３］を用いて、処理システムを開発中である。

５．　Prolog を用いたシミュレーション

４．で示した時相論理に基づくハードウェア機能記述は、以下に示すようにProlog の記述に変換して、シミュレーションを行なうことができる。

変換は各Ｉnterval ごとに行なう。４．で示したように、ハードウェア機能記述は時相論理の簡単な式のＡＮＤで表現できるため、文献［１１］の手法を用いて効率的に状態遷移表現に展開することができ、文献［１０］に示すようにProlog の記述に変換することができる。このようにして得られるProlog の記述から制御信号を作り、各Ｉnterval 間の実行の制御を行なう。

各Ｉnterval の変換は、←、beg 、fin 演算子を用いたものと、そのＩnterval 内での□、▽、Ｕ、○演算子を用いたものの２つに分けられる。後者は文献［１１］の手法に従って行なう。また前者は、そのＩnterval におけるbeg とfin の関係としてProlog の記述に変換する。ＩＦやＣＡＳＥ文は、条件分岐の数に従ってProlog のclauseに分け、制御信号によって制御する。また、処理系にProlog を用いているので、記号シミュレーションも比較的容易に実行することができる。

以上のような方法でProlog の記述に変換することができる。図４のユニフィケーションのアルゴリズムに対し、簡単なappendを実行するのに、ＶＡＸ11／ 730上のＣ－Prolog で約１０秒程度かかる。

６．　おわりに

時相論理（ＬＴＴＬ）に基づくハードウェア機能記述とその支援について述べた。

Ｉnterval という概念を導入することにより、アルゴリズム等も容易に記述することができる。また、機能記述に関する制御は簡単な時相論理の式で表現でき、検証・合成等を効率よく実行することができる。

現在、時相論理に基づくハードウェア機能記述言語を設定し、そのProlog を用いたシミュレータについて設計を終え、幾つかの例についての動作を確認している。今後、シミュレーション、検証、合成をシステムとして実装していく予定である。

参考文献

[1] 樹下(編):論理装置のCAD、情報処理学会叢書5、オーム社、東京(1981).

[2] M. R. Barbakki : Insttuction Set Processor Specification (ISPS) : The Notation and its Applications, IEEE Trans. Computers, Vol. C-30, No.1, Jan. (1981).

[3] J. R. Duley, D. L. Dietmeyer : A Digital System Design Language (DDL), IEEE Trans Computers, Vol. C-17, No.9, Sept. (1968).

[4] T. J. Kowalski, D. E. Thomas : The VLSI Design Automation Assitant : Prototype System, 20th DAC, June (1983).

[5] C. Y. Hitchcock, D. E. Thomas : A Method of Automatic Data Path Synthesis, 20th DAC, une (1983).

[6] N. Kawato, T. Saito, F. Maruyama, T. Uehara : Design and Verification of Large-Scale Computers by using DDL, 16th DAC, June (1979).

[7] J. R. Duley, D. L. Dietmeyer : Translation of a DDL Digital Sstem Specification to Boolean Equation, IEEE Trans. Computer, Vol. C-18, pp.305-313 (1969).

[8] T. Uehara, F. Maruyama, T. Saito, N. Kawato : n Interactive Logic Synthesis System Based Upon AI Technique, 19th DAC, June (1982).

[9] 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, March (1984).

[10] 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、情報処理学会論文雑誌、Vol.24, No.4, July (1984).

[11] 藤田、田中、元岡:時相論理を用いたハードウェア同期部の仕様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、Ec83-59、March (1984).

[12] 藤田、田中、元岡:時相論理とProlog を用いたゲート回路の効率的検証、情報処理学会、設計自動化研究会資料、DA21-4, May (1984).

[13] 西山:Prolog を用いたハードウェア同期部検証の効率化、東京大学工学部電気工学科卒業論文、March (1984)

[14] 藤田、田中、元岡:定理証明法によるハードウェア演算部の検証、情報処理学会第28回全国大会、5P-1、March (1984).

[15] E. I. Organick, et.al : Transforming an Ada Program Unit to Silicon and Verifying Its Behavior in an Ada Environment : A First Experiment, IEEE Software Magazine pp.31-49, Jan. (1984).

[16] occam Programming Manual, INMOS ltd. England (1983).

[17] B. Moszkowski : Reasoning about Digital Circuit, Dept. of Computer Science, Stanford University, Report No.STAN-CS-83-970 (198

[18] B. Moszkowski, Z. Manna : Reasoning in Interval Temporal Logic, Dept. of.Computer Science, Stanford University, Report No.STAN-CS-83-969 (1983).

[19] Z. Manna, A. Pnueli : Verification of Concurrent Programs Part1 : The Temporal Framework, Dept. of Computer Science, Stanford University, Report No.STAN-CS-81-836 (1981

[20] P. Wolper : Temporal Logic Can Be More Expressive, 22nd Annual Synposium on Foundation of Computer Science, Oct. (1981).

[21] A. Goto, H. Tanaka, T. Moto-oka : Highly Parallel Inference Engine PIE -Goal-Rewriting Model and Machine Archtecture-, New Genaration Computing, Vol.2, No.1, Feb. (1984)

[22] M. Yuhara, H. Koike, H. Tanaka, T. Moto-oka : A Unify Processor Pilot Machine for PIE, The Logic Programming Conference '84 Tokyo, Mrch (1984).

[23] F. Pereira : C-Prolog Users Manual Version1.5, Ed CAD, University of Edinburgh (1984).