時相論理を動作記述に用いたデータパス検証システム†

中村 宏††* 久木元 裕治††† 田中 英彦††††

本論文では，機能設計支援を目的としたデータパス検証システムを提案する．本システムはレジスタトランスファレベルの動作記述と構造記述の両方を入力とし，a）動作記述と構造記述の間の対応情報の抽出，b）動作可能性の検証，c）制御論理の抽出を行う．本システムでは，動作記述を時相論理に基づく言語で与える．時相論理を用いることにより，時間に関する記述を容易かつ明瞭に行える．本論文では，パイプライン動作等の，並列性を内在する動作記述例を用いることによって，時間に関するその記述能力を示すとともに，それらの動作記述に対して検証システムを適用した結果を報告する．本システムは実用的な処理能力を有し，これを用いることにより効果的な機能設計支援を実現できると考えられる．

1. はじめに

近年の半導体技術の進歩により，VLSI の規模は増大し，また ASIC の普及に見られるようにその応用分野も急激に広がりつつある．これに伴い，CAD システムの必要性が以前にも増して高まっている．

VLSI 設計では複数の設計段階を設定し，抽象的な仕様から出発して段階的に詳細化を行う．まず，システムレベル設計として VLSI で実現すべき問題を想定し，その問題を所定のコスト性能で実現するための方式，アルゴリズムを決定する．次の機能設計では，システムレベル設計で得られた動作記述を，レジスタ間転送を基本とした記述に詳細化し，演算器，レジスタ等の機能ブロックを構成要素としたデータパスの構造を決定する．詳細化した動作記述がデータパス構造上で動作するための制御論理の設計も，この機能設計段階で行う．次のゲートレベル設計では，データパス系，制御系の各機能ブロックの詳細論理を定め，基本素子の接続構造に変換する．その後，実際のチップ上へのマッピングをする実装設計へと進む．

現在のところ，各設計段階ともその支援は未だ十分ではないが，実装設計の支援はある程度自動化が進み，その上位レベルにあたるゲートレベル設計を支援する合成ツールも商品化されつつある．しかし，それより上位レベルの設計支援はまだ研究の段階である．

本論文で提案するデータパス検証システムは機能設計の支援を目指すものである．われわれは，機能設計を以下のような流れで行うことを考えている．

まず，設計者は設計したいハードウェアの動作アルゴリズムを記述する．次にシミュレーション等で動作確認を行いながら記述を詳細化し，レジスタトランスファレベルの動作記述とする．ここで用いる動作記述言語は以下の点を満足する必要があると考えられる．

- 順序性・並列性といった時間に関する記述が容易かつ明瞭にできる．
- アルゴリズムレベルからレジスタトランスファレベルまでの様々なレベルの記述が同一の言語で行える．
- 実行可能な（シミュレート可能な）言語である．

われわれは，この三点を満たす言語として，時相論理に基づく言語 Tokio を用いることとした．

さらに，設計者は同時に，データパスの構造記述を与える．動作記述からデータパスを自動生成する研究も盛んに行われてはいる[1]が，現在のところ結果の品質は必ずしも十分ではない．さらに，下位レベルからのフィードバックによる修正，あるいは，以前に設計したものの再利用が難しいという問題点もある．これは，動作記述と自動合成された構造記述の対応が明確でないことによる．

実際の人間による機能設計を考えた場合，通常，設計者は過去の経験などにより設計の初期の段階からデータパスをある程度意識し，また動作記述と構造記述との対応情報を頭の中で作りながら設計を進めていく．この対応情報が，高品質な VLSI 設計への鍵である．そこでわれわれは，設計者が動作記述と構造記述の両方を与えることを仮定し，以下の支援を実現するのが現実的かつ効果的な機能設計支援であると考える．

† Data Path Verification System Using Temporal Logic for Behavioral Description by HIROSHI NAKAMURA (Electrical Engineering Course, Graduate School of Engineering, University of Tokyo), YUJI KUKIMOTO (Information Engineering Course, Graduate School of Engineering, University of Tokyo) and HIDEHIKO TANAKA (Department of Electrical Engineering, Faculty of Engineering, University of Tokyo).
†† 東京大学大学院工学系研究科電気工学専攻
††† 東京大学大学院工学系研究科情報工学専攻
†††† 東京大学工学部電気工学科
* 現在 筑波大学電子・情報工学系

1. 動作記述と構造記述との間の対応情報を自動的に抽出する.
2. 与えられた構造記述上で，動作記述が動作可能であることを自動的に検証する.
3. 与えられた構造記述上で，動作記述を実現するために必要な制御情報（制御部の論理）を自動的に抽出する.

将来，品質の良いデータパスが自動合成されるとしても，やはり修正の部分には人手が介入するので，ここで述べた三つの支援の重要性は変わらない.

われわれは，以上の三点を実現するデータパス検証システムを作成したので，以下報告する．以前に報告した検証システム[2]は，プロセッサのような例には適用できず，また第三の点である制御論理の抽出を行えなかったが，本論文のシステムはそれらの問題を解決したものである.

データパス検証に関しては，文献 3）による報告があるが，そこでは，パイプライン動作といった並列性を持つ例はとり上げられていない．本データパス検証システムは，動作記述言語 Tokio の持つ，時間に関する豊富な記述能力に十分対応できるものになっている.

本論文の構成は以下のようである．２章でまず，動作記述言語として採用した Tokio について説明する．３章で検証システムの構成を示し，４章で検証手法を示す．５章では，本システムを例題に適用させた結果について述べる.

2. 動作記述言語 Tokio

2.1 Tokio

Tokio[4)~6)] は時区間時相論理[7)] に基づく論理型言語であり，直観的には Prolog に時間の概念を入れたものである．したがって，Tokio は Prolog 同様にアルゴリズムを自由に記述することができる．また，Tokio は論理に基づくため数学的基礎がはっきりしておりセマンティクスが明瞭である．このためハードウェアに内在する並列性，順序性といった時間関係を容易かつ明瞭に記述できる．Tokio と Prolog の違いは，時相演算子と変数に代表されるので，ここではその二点についてのみ述べる.

（1） 時相演算子

```
head :- p, q.
```

この記述は，head が定義されている時区間（インターバル）内で，ゴールと p と q が並列に実行されることを表す.

一方，順序性は chop 演算子（&&）で表される.

```
head :- p && q.
```

この記述は，head の定義されている時区間を前半と後半の二つにわけ，前半で p を実行し，後半で q を実行することを示す．時区間を前半と後半に分割する時刻は非決定的に選択される.

（2） Tokio 変数

Prolog の変数が一度値が決まるとそれを保持し続けるのに対し，Tokio の変数は，各時刻内では Prolog の変数と同様に扱われるが，時刻が変わると値を変える．Tokio の変数には二種類ある.

（a） local 変数：これは時刻と共に値を変える変数である．変数名は大文字で始まる.

（b） global 変数：一度値が決まると次の代入までその値を保持する．変数名のはじめに ‘*’ がつく.

変数への値の代入には，Temporal Assignment と即時代入がある.

（a） Temporal Assignment

これは，インターバル内で定義され，local 変数に対しては，‘P <- (value)’，global 変数に対しては，‘*p <= (value)’ と記述される．いずれも，Temporal Assignment が定義されているインターバルの最初の時刻の (value) を，インターバルの最後の時刻で P(*p) に代入する．これは，レジスタ転送の一般形と考えられる.

（b） 即時代入

これは，時刻に対して定義され，local 変数に対しては，‘P = (value)’，global 変数に対しては，‘*p := (value)’ と記述される．いずれも，その時刻における (value) をその時刻に P(*p) に代入する.

2.2 RTL-Tokio

Tokio の時間は離散時間であり，最小時間単位が存在する．この最小時間をハードウェアの何に対応させるかを変えることにより，Tokio は様々なレベルの動作記述を行うことができる．例えば，最小時間単位をクロックと考えれば同期回路に対応し，絶対時間と結び付ければ１クロック内の非同期動作といったハードウェアの細かい動作の記述が行える.

RTL-Tokio (Register Transfer Level Tokio) はレジスタトランスファレベル動作記述言語であり，最小時間を１マシンサイクルに対応させる．RTL-Tokio は，以下のような制限を加えた Tokio のサブセットである.

（1） 変 数

時間の経過によらず値を保持するglobal変数は，レジスタやメモリを表す．レジスタトランスファレベルにおいてレジスタあるいはメモリに対する値の転送は必ず時間の経過と共に行われるので，global変数に対する代入はTemporal Assignmentに限り，即時代入は，許さない．また，local変数も間接的にregister, memoryなどのデータを表すため，global/local変数共にリスト構造を持つことができない．

（2） インターバルの長さ

Tokioでは，chop演算子によって分割されたインターバルの長さが非決定的に定まる．レジスタトランスファレベルの記述としては，レジスタ間転送が1マシンサイクルで実行されるように，インターバルの長さを決めるべきである．したがって，RTL-TokioではTemporal Assignmentがすべて長さ1で実行されるように，インターバルの長さを決める．

（3） 再帰呼び出し

Tokioでは，Prolog同様再帰呼び出しが行える．しかし，一般の再帰構造をハードウェアで実現することを考えると，再帰構造を制御するスタックが必要になる．レジスタトランスファレベルの動作記述としては，再帰構造を制御する暗黙のスタックを陽に記述するべきである．したがって，ループ構造と等価なtail recursionのみ許すことにする．RTL-Tokioの構文は以下のように制限される．

head(arg) :- localConds, !, recursiveCall.

head(arg) :- localConds, !, actions && actions && ……&& recursiveCall.

ここで，localCondsは現在時刻で判定可能なものであり，複数あっても構わないし，なくても構わない．recursiveCallでは再帰呼び出しを許すのに対し，actionsでは再帰呼び出しを許さない．

2.3 RTL-Tokioによる記述例

平方根の近似値を求めるニュートン法を例に，TokioおよびRTL-Tokioによる記述を示す．図1(a)がそのアルゴリズムである．(b)に示すのは，(a)中の演算'Y := (Y+X/Y)/2'を'Tmp <- X/Y+Y, X <- X'と'Y <- Tmp/2, X <- X'としてスケジューリングを行うが，'C <- C+1'についてはスケジューリングを行っていないTokioの記述である．Tokioでは，(b)のような部分的にスケジューリングの行われた記述も行え，シミュレータで実行可能である．ここで，'X <- X'の記述は，インターバルの最初の時刻Xの値をインターバルの最後の時刻（次のインターバルの最初の時刻と同じ）のXに代入することを表す．時刻と共に値を変えるlocal変数Xの値を次のインターバルで使う場合は，この記述が必要になる．

```
Y := 0.222222 + 0.888889 * X ;
I := 0 ;
DO UNTIL I = 2 LOOP
 Y := 0.5 * ( Y + X/Y );
 I := I + 1 ;
ENDDO;
```

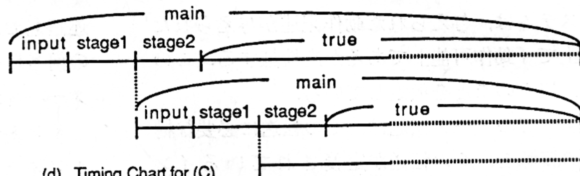
(a) Algorithm of Newton's Method

```
main(X) :-
  Y <- 0.222222 + 0.888889 * X,
  X <- X  && sub(X,Y,0).
sub(X,Y,C) :- C = 2, !.
sub(X,Y,C) :- !,
   (Tmp <- X / Y + Y , X <- X
    && Y <- Tmp / 2, X <- X),
    C <- C + 1 && sub(X,Y,C).
```

(b) Newton's Method in Tokio

```
main :-  *adr = 8, !, true.
main :- !, input && stage1 && main, (stage2 && true).
input :- !, *input1  <- *memory(*adr), *adr <- *adr + 1
 && *reg1 <- 0.22222 + 0.888889 * *input1.
stage1 :- !, *reg2 <- *input1 / *reg1 && *reg2 <- *reg2 + *reg1
 && *reg3 <- *reg2 / 2, *input2 <- *input1.
stage2 :- !, *reg4 <- *input2 / *reg3 && *reg4 <- *reg4 + *reg3
 && *output <- *reg4 / 2.
```

(c) Pipelined Newton's Method in RTL-Tokio



(d) Timing Chart for (C)

図1 Tokioによる動作記述例
Fig. 1 Behavioral description in Tokio.

(c)は，(b)の記述から導出された，RTL-Tokioによるパイプライン化された動作記述である．ここでは，演算のスケジューリングが行われ，ループの数(2回)を覚えていたカウンタを除く代わりにループをstage 1とstage 2に展開している．これは，機能設計段階における，データパスと制御部の切り分けに関係する問題である．パイプライン起動は'main, (stage 2 && true)'の部分で宣言されている．この記述は，stage 2と次のパイプラインのmainが同時に起動されることを表す．'true'は，任意の時間長で成功する組み込み述語であり，ここでは異なる時間長を持つmainとstage 2の時間長を合わせるために使われている．(c)に示す動作記述のタイミングチャートを(d)に示す．

3. データパス検証システムの構成

本論文で提案するデータパス検証システムの構成は，図2に示すとおりである．

入力はレジスタトランスファレベルの動作記述と構造記述，および operation rule である．動作記述は RTL-Tokio で与え，構造記述は Prolog で与える．ここでの構造記述は結線情報を表しており，他の構造記述言語からここで用いる Prolog 形式に容易に変換できる．変換するツールは今後作成する．また operation rule は，動作記述と構造記述との対応をとるための rule であり，これも Prolog で与えられる．

このシステムは，与えられた動作記述が与えられた構造記述上で「動作可能」であるかを検証する．この検証では，以下のことを仮定している．

（1） すべてのデータ転送は，その転送に必要な構造記述中の資源を，その転送が定義されているインターバルの時区間の間，占有し続ける．

（2） 動作記述中の演算子と構造記述中の演算器の種類の対応は operation rule で与える（4.2 節で述べる）．

（3） レジスタおよびメモリに関しては，動作記述と構造記述とで，同じものを指すのに同じ名前を用いる．

4. データパス検証手法

このシステムでは，まず Translator において入力動作記述を中間形式に変換する．その後，Facility Checker において，動作記述と構造記述との間の対応情報をファシリティ使用表 (Facility Usage Table) として導出する．最後に，動作記述中の並列性を Time Tracer が調べ，データパスのコンフリクト（二重使用）を検証する．Time Tracer においては状態遷移図 (State Transition Table) が抽出され，Facility Usage Table に導出されている対応情報と合わせて，制御部の論理を抽出したことになる．

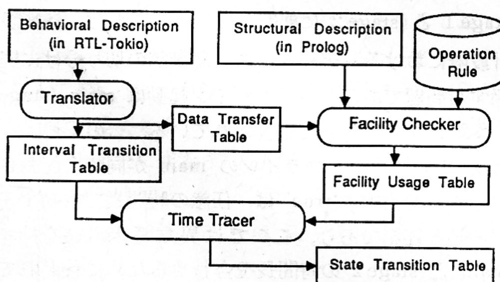


図 2 データパス検証システムの構成
Fig. 2 Structure of data path verifier.

4.1 Translator

Translator では，動作記述のインターバル遷移表・データ転送表への変換が行われる．

まず，RTL-Tokio による動作記述中の各インターバルにそれぞれ固有の名前をつけ，各インターバルの長さを決定する．そして，述語呼び出しと chop 演算子によるインターバル間の遷移関係を遷移条件と共に取り出し，インターバル遷移表を作る．インターバル固有の名前は intId = ‘(predName/arity, clauseNum, intNum)’ で与えられる．ここで，predName/arity：述語名およびその引数の数，clauseNum：同一述語中の何番目の定義か，intNum：述語中の何番目のインターバルかを表す．

インターバル遷移表を作成した後，動作記述中の各インターバルにおけるデータ転送を，インターバル固有の名前と共に取り出し，データ転送表を作成する．

4.2 Facility Checker

Facility Checker では，データ転送表中の各データ転送と構造記述とを照合し，そのデータ転送の実現に必要な構造記述の資源を探し，ファシリティ使用表を作る．資源の候補が複数ある時は，すべての候補をファシリティ使用表に記憶する．すなわち，以下の処理を行う．

（1） 各データ転送中の演算を実現する演算器（の組合せ）を構造記述中から探す．

（2） データ転送の転送元から（1）で探した演算器（の組合せ）へのパス，および演算器から転送先へのパスを探す．

（1）において，一つの演算は複数の方法で実現されることを考慮する必要がある．例えば，‘掛ける 2’ は，掛算器，1ビットシフタ，加算器等で，実現可能である．すべての候補を見つけるためには，各演算器の機能の宣言と，等価演算の規則とを与えなければならない．各演算器の機能の宣言は，構造記述として以下の形で与えられる．

```
func([function, input-port, output-port],
                                        signal-line).
```

例．
```
func([add, [[adder, in 1], [adder, in 2]],
[[adder, out]]], [adder, cnt]).
```

また，等価演算の規則は operation rule として与えられ，以下のように記述されている．

```
opSame([[multi, X, 2], [shift_left, X, 1],
```

[add, X, X]]).

operation rule は設計者が自由に変更できる．例えば operation rule を記述しないことは，動作記述中の‘*2’, ‘≫1’, ‘+’ はそれぞれ掛算器，1ビットシフタ，加算器でのみ実現されるという仮定を置くことに相当する．

(1)(2)が終了すると，次の(3)に進む．

(3) 一つのインターバル内で使用される構造記述中の資源の二重使用をチェックする．二重使用があれば誤設計である．そうでなければ，そのインターバルが使う資源の組合せをファシリティ使用表とし，次の Time Trace へ行く．ここで，あるデータ転送を実現する資源の組合せ方が複数ある場合は，二重使用を避けるような選択ができるか否かを調べることになる．実際のハードウェアを検証する場合，冗長なデータ転送路は多く存在しないので，資源の組合せの数はさほど大きくはならない．

4.3 Time Tracer

ここでは，同時に生起するインターバルの組を見つけ，そのインターバル同士が同一のデータパス上の資源を使用しているか否かを調べる．

インターバル間遷移を時間に対して順方向に探索する手法 (Forward Trace) と逆方向に探索する手法 (Backward Trace) の両方を実装した．

[順方向探索 (Forward Trace)]

この探索では，まず並列に実行されるインターバルを見つけ (第一段階)，その後，それらが同一資源を使用しているか否かを調べる (第二段階)．

(第一段階) 順方向探索では，時刻を一ずつ進ませながらインターバル間の遷移を取り出す．この探索は深さ優先で行われる．

RTL-Tokio の記述における状態を，Translator で決定された各 intId とその中の何クロック目かを表す clock で，state= {intId, clock} として定義する．するとこの探索の過程で，制御系の状態遷移図が抽出される．得られる状態遷移図は Moore 型のものである．処理の詳細は以下のとおりである．

(step 1) 最初のインターバル I_{init} とクロック clock (通常 0) を設定する．$S_0 = \{I_{init}, \text{clock}\}$.

(step 2) (並列関係の処理) もし S_i 中に述語呼び出しがあれば，呼ばれたインターバルを S_i に加え，S_i' とする．クローズ内の最後のインターバル以外での呼び出しの場合は，呼び出しの親側も記録する．遷移条件も記録する．

(step 3) (順序関係の処理) 一時刻進ませ，S_i' の次の状態 S_{i+1} を得る．以下の二つの処理のいずれかを行う．

- インターバルの長さを越えないならばクロック clock を一つ増やす．
- インターバルの長さを越える場合は chop 演算子による遷移を経て次のインターバルに進む．

(step 3) が終了すると (step 2) へ戻る．

〈停止条件〉

(step 2) でいま，S_n' が得られたとする．以下の二つのいずれかの条件が成立すれば探索は終了する．

- S_i' が S_n' を含むような $i(0 \leq i < n)$ が存在する．
- S_n' の一時刻後の S_{n+1} が存在しない．

実行例：**図 3** (a)に示す RTL-Tokio の記述を例として示す．(b)が最終的に得られる状態遷移図である．

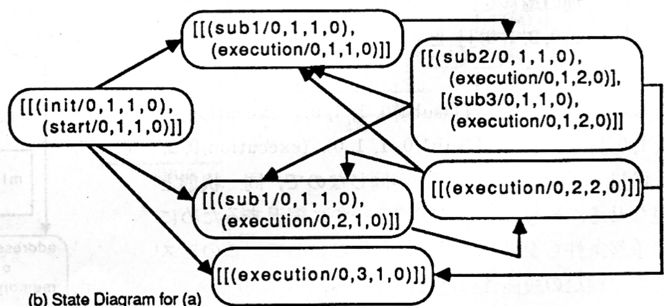
(step 1) S_0=[[(start/0, 1, 1, 0)]]

(最後の 0 は clock を表し，それ以外は intId を表す)

(step 2) S_0'=[[(init/0, 1, 1, 0), (start/0, 1, 1, 0)]]

```
start :- init && execution.
init :- *a <= 1, *b <= 2, *c <= 4, *d <= 0.
execution :- *a =< 10, !, sub1 && sub2, sub3 && execution.
execution :- *a =< 40, !, sub1 && *a <= *c + *d && execution.
execution :- !, empty.
sub1 :- !, *d <= *a.
sub2 :- !, *a <= *b  *  *d.
sub3 :- !, *c <= *c + 1.
```

(a) Traced Example in RTL-Tokio



(b) State Diagram for (a)

図 3 Time Trace の例題
Fig. 3 Example for Time Trace.

(step 3) まず S_1=[[(execution/0, 1, 1, 0)]] が遷移条件 [=<, *a, 10] と共に得られる．探索は深さ優先で行われるので他の候補である [[(execution/0, 2, 1, 0)]] は後で探索される．

(step 2) S_1'=[[(sub1/0, 1, 1, 0), (execution/0, 1, 1, 0)]] が得られる．この S_1' は，[[(sub1/0, 1, 1, 0), (execution/0, 2, 1, 0)]] とは区別される必要がある．なぜならば，(execution/0, 1, 1, 0) から呼ばれた場合と (execution/0, 2, 1, 0) から呼ばれた場合では，その後の状態はそれぞれ (execution/0, 1, 2, 0) と (execution/0, 2, 2, 0) であり，異なるからである．(step 2) において呼び出しの親側の名前も記憶する理由はこれである．

(step 3) S_2=[[(execution/0, 1, 2, 0)]]

(step 2) S_2'=[[(sub2/0, 1, 1, 0), (execution/0, 1, 2, 0)], [(sub 3/0, 1, 1, 0), (execution/0, 1, 2, 0)]] が得られる．このように S_i' の中が複数である場合は，並列動作が存在する時である．

(step 3) S_3=[[(execution/0, 1, 3, 0)]]

(step 2) S_3'=[[(execution/0, 1, 1, 0)]] がまず得られる．この場合のように最後のインターバルから呼ばれている場合は，述語呼び出しの親のいかんにかかわらず次の状態が一意に決まる．具体的に述べれば，この例では (execution/0, 1, 3, 0) から呼ばれようと (start/0, 1, 2, 0) から呼ばれようと，(execution/0, 1, 1, 0) の次の状態は (execution/0, 1, 2, 0) で同じである．したがって，(execution/0, 1, 3, 0) から呼ばれたことを記憶する必要がない．これが，(step 2) において最後のインターバルにある述語呼び出しの親を記憶しない理由である．

さらに述語呼び出しを考慮すると，S_3'=[[(sub1/0, 1, 1, 0), (execution/0, 1, 1, 0)]] が得られる．

ここで，$S_3'=S_1'$ が成立し，停止条件を満足する．そこで，別の遷移をたどり，S_3'=[[(sub1/0, 1, 1, 0), (execution/0, 2, 1, 0)]] とする． (以下省略)

〈状態の縮退〉

この例では，S_1'=[[(sub1/0, 1, 1, 0), (execution/0, 1, 1, 0)]] と S_3'=[[(sub1/0, 1, 1, 0), (execution/0, 2, 1, 0)]] は，行うデータ転送が同じなので，同一状態と見なせる．一般に二つの状態を一つに縮退するためには遷移条件も変更しなければならないので，このシステムは縮退の可能性のみを出力する．

(第二段階)

ここでは，第一段階で得られる同時実行インターバルが，構造記述中の同一資源を二重使用するかどうかを，ファシリティ使用表から調べる．

[逆方向探索]

順方向探索とは逆に，まず構造記述中の同一資源を使用する二つのインターバルの組合せを見つけ（第一段階），その二つのインターバルが同時に起こるか否かをインターバル間の遷移関係を時間に対して逆方向にたどりながら調べる（第二段階）．第一段階で見つけたすべての組合せに対して第二段階を行う．探索アルゴリズムは方向が逆な点以外は同じなので，詳細は省略する．

5. 実験結果および考察

本章では，二つの例題に検証システムを適用させた結果について述べる．本システムは SICStus-Prolog 上に実装されており，使用した計算機は SUN 4/260 である．

5.1 実験結果

（例題 1） ニュートン法による平方根の計算

2 章で取り上げたニュートン法を例に検証した．入力した動作記述は図 1 (c) で，構造記述は**図 4** である．

この例では，並列に動作する二つのインターバル (input/0, 1, 2) と (stage 2/0, 1, 2) において，構造記述中の addA が二重使用されていることが検出される．設計誤りがある場合のシステムの出力は，二重使用されるデータパス上の資源，その資源を用いるインターバルの名前，およびそのインターバルの並列実行を引き起こす状態遷移の履歴である．また，一つの誤りを

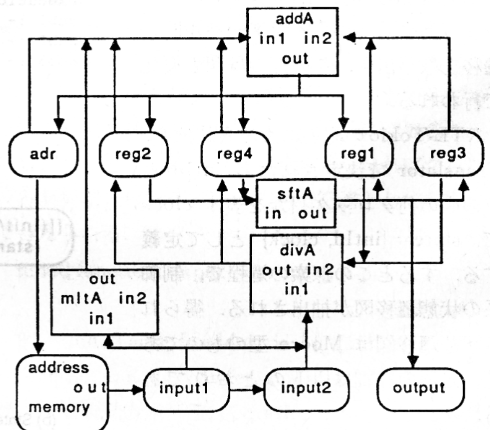


図 4 平方根計算用データパス
Fig. 4 Data path for computing square root

検出しても処理を中断せず，すべての誤りを検出する．

図1(c)ではループの数が2回であったが，4回，8回にした動作記述に対しても検証を行った．パイプラインの段数はいずれも二段であり，並列度は同じである．入力した構造記述は，いずれの動作記述に対しても図4を用いた．検出される誤りの数は，いずれの場合も一つだけである．

検証結果を**表1**に示す．状態の縮退は，いずれの動作記述に対しても行われない．

（例題2）　プロセッサ

文献8)にあるプロセッサを例に取り，データパス検証システムを適用させた結果について述べる．

（1）　RTL-Tokio による記述

RTL-Tokio による動作記述の一部を**図5**に示す．簡単のため，この記述ではデコーダは省略し，命令の長さ，種類を陽に記述している．このプロセッサには 10 個の1バイト命令，6個の2バイト命令がある．実行は ifetch 部と execution 部からなり，ifetch が終了すると execution が行われる．命令 add および adc 以外の場合は，execution が終了してから ifetch が行われるが，この二つの命令に対しては二サイクルからなる execution 部の後半と次命令の ifetch が並列実行される．図5の下線部にそれが示されている．

（2）　データパス検証結果

入力は図5の動作記述と**図6**に示すデータパスである．データパスのビット幅は，明記してない部分はすべて8ビットである．この例題に関しては，各パスはビット展開しないで一つの資源として扱えるのでビット幅は検証に要する CPU 時間に影響しない．

ALU 等の複数の機能を実現する演算器に関しては，文献 2) のシステムでは扱えなかったが，本システムでは，異なる信号線入力を持つ機能は同時に実行できないとして検証を行える．この例では，ALU の四つの機能 alu 0, alu 1, alu 2 a, alu 2 c のうち，alu 2 a と alu 2 c の入力信号線名は同じであり，同時に実現できるとしている．

検証結果を**表2**に示す．この例では，設計誤りは検出されない．また，状態の縮退は行われない．

表 1　平方根計算回路に対する実行結果
Table 1　Verification result of computing square root.

Number of Repetitions	CPU-Time (sec)				Number of Derived States
	Translator	Facility Checker	Time Trace		
			forward	backward	
2	0.40	3.19	0.77	3.04	12
4	0.57	4.75	1.45	13.91	21
8	0.88	8.20	3.28	122.2	39

```
ifetch :- !, *op1 <= *mem(*pc) , *pc <= *pc+1 && ifetch_branch.
    ifetch_branch :- *op1 = stop, !, empty.
    ifetch_branch :- (_,2,_) = *op1,!,ifetch2.
    ifetch_branch :- !, execution.
ifetch2 :- !, *op2 <= *mem(*pc) , *pc <= *pc+1 && execution.
execution:-  !, exec_first   && execution_latter.
    exec_first :- (add,_,_) = *op1, !,   *memd <= *mem(*op2).
    exec_first :- (adc,_,_) = *op1, !,  *memd <= *mem(*op2).
    exec_first :- (clr,_,_) = *op1, !,  *a <= 0.
    exec_first :- (com,_,_) = *op1, !,  *a <= alu0(*a).
    exec_first :- (push,_,_) = *op1, !, *mem(*sp) <= *a.
    .....(omitted)........
execution_latter :- (Op,By,short) = *op1,!,ifetch.
execution_latter :- (Op,By,long) = *op1,!,
                    (exec_second && true),ifetch.
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^<===== concurrency
execution_latter :- (Op,By,sec2) = *op1,!,exec_sec2_begin.
exec_second :- (add,_,_) = *op1, !, *a <= alu1(*a,*memd).
exec_second :- (adc,_,_) = *op1, !, *a <= alu2a(*a,*memd,*c),
                                    *c <= alu2c(*a,*memd,*c).
exec_sec2_begin :-  !, exec_sec2 && ifetch.
    exec_sec2 :- (pop,_,_) = *op1, !, *a <= *mem(*sp).
    exec_sec2 :- (jsr,_,_)  = *op1, !, *pc <= *op2, *sp <= *sp-1.
    exec_sec2 :- (rts,_,_)  = *op1, !, *pc <= *mem(*sp).
    exec_sec2 :- (push,_,_) = *op1, !, *sp <= *sp-1.
    '$function' alu0(A)   = Out :- alu0(A,Out).
            alu0(X,Out)   :- !, Out = -X.
    '$function' alu1(A,B) = Out :- alu1(A,B,Out).
            alu1(X,Y,Out) :- !, Out = (X + Y) mod 256.
    ( Definitions for alu2a and  alu2c are omitted.)
```

図 5　RTL-Tokio によるプロセッサ動作記述
Fig. 5　Behavioral description of processor in RTL-Tokio.

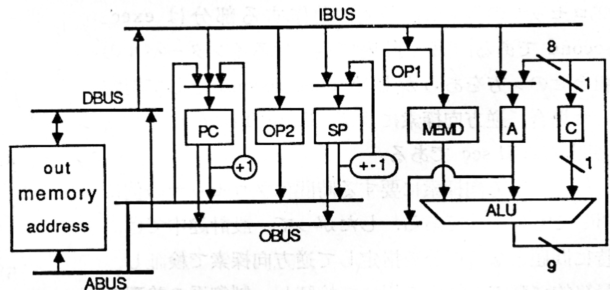


図 6　プロセッサのデータパス
Fig. 6　Data path of processor.

5.2 考　　察

図1および図5に示すとおり，Tokio は，パイプラ

表 2 プロセッサに対する実行結果
Table 2 Verification result of processor.

CPU-Time (sec)				Number of Derived States
Trans-lator	Facility Checker	Time Trace		
		forward	backward	
3.39	10.0	34.1	1254	26

イン動作等，並列動作の内在する動作記述を容易かつ明瞭に行える．したがって，Tokio は時間に関する十分な記述能力を持つと言える．

また，本論文で示したシステムと，文献 2) のシステムとの違いは主に，プロセッサの例で示したように複数の機能を実現する演算器を含むデータパスが検証できる点，および RTL-Tokio の構文として tail recursion 以外の再帰構造を認めないことにより，順方向探索の停止性を保証し制御論理の抽出を実現した点である．検証速度はさほど向上していない．

データパス検証速度に関しては以下のことが考察できる．表 1 より，Translator および Facility Checker に要する時間は，動作記述の大きさにほぼ比例する．検証に要する時間を支配するのは Time Tracer の部分である．逆方向探索の場合，第一段階で見つけられるインターバルの組合せの数は，インターバルの数を N とすると最悪 ${}_NC_2$ になる．ある資源が動作記述中のすべてのインターバルで使用される場合がその最悪の場合である．したがって，プロセッサのようにほとんどのデータ転送で用いられるバス構造を持つ例に対しては，表 2 からわかるように逆方向探索は不利である．しかし，第一段階で見つけるインターバルをあらかじめ限定することで高速化を図ることができる．プロセッサの例では，並列に動作する部分は exec_second である．第一段階で見つけるインターバルの組合せの一方をあらかじめ exec_second として指定した場合，逆方向探索による Time Trace に要する時間は 2.51 sec である．

一方，順方向探索に要する時間はプロセッサの例に対しても数十秒である．したがって，設計途中では，特に問題となる部分を指定して逆方向探索で検証し，最終的に順方向探索を用いて検証し，制御系の論理を抽出すれば，効率の良い設計支援を実現できると考える．

6. おわりに

本論文では，レジスタトランスファレベルを対象としたデータパス検証システムについて報告した．

本システムでは，動作記述を時相論理に基づく言語で与える．時相論理を用いることにより，時間に関する記述を容易かつ明瞭に行える．パイプライン動作等，並列動作の内在する動作記述を実際に行うことでこの点を示した．

また，本システムが，動作記述と構造記述を入力とし，a）動作記述と構造記述の間の対応情報の抽出，b）動作可能性の検証，c）制御論理の抽出を行うことができることを，並列動作を含む例に適用させることにより示した．プロセッサに対する処理時間は数十秒程であり，十分実用的な処理能力を持つシステムであると言える．

残された課題としては，機能設計段階における動作記述の詳細化を，Tokio を用いて支援することが挙げられる．

謝辞 日頃御討論頂く富士通研究所の藤田昌宏博士，ソニー CSL 研究所の河野真治博士，システムの実装を手伝って頂いた第一勧業銀行の中井正弥君に感謝いたします．

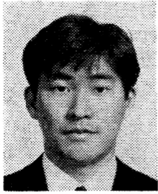
なお，本研究は一部文部省科学研究費補助金（課題番号 01790381）による．

参考文献

1) McFarland, M. C., Parker, A. C. and Camposano, R.: Tutorial on High-Level Synthesis, *Proc. 25th Design Automation Conference*, pp. 330-336 (1988).
2) Nakamura, H., Fujita, M., Kono, S., Nakai, M. and Tanaka, H.: A Data Path Verification System using Temporal Logic Based Language Tokio, *Proc. IFIP WG 10.2 Working Conference on the CAD Systems Using AI Techniques*, pp. 127-134 (1989).
3) 高木：データパス検証補正へのアプローチ，情報処理学会論文誌，Vol. 25, No. 1, pp. 9-18 (1985).
4) Aoyagi, T., Fujita, M. and Tanaka, H.: Temporal Logic Programming Language Tokio, *Proc. Logic Programming Conference '85*, LNCS-221, Springer-Verlag (1985).
5) Kono, S., Aoyagi, T., Fujita, M. and Tanaka, H.: Implementation of Temporal Logic Programming Language Tokio, *Proc. Logic Programming Conference '85*, LNCS-221, Springer-Verlag (1985).
6) Fujita, M., Kono, S., Tanaka, H. and Motooka, T.: Aid to Hierarchical and Structured Logic Design Using Temporal Logic and Prolog,

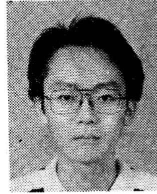
Proc. Pt. E, pp. 283-294, IEE (1986).
7) Moszkowski, B. : A Temporal Logic for Multi-Level Reasoning about Hardware, *Proc. CHDL '83*, IFIP (1983).
8) Karatsu, O. : VLSI Design Language Standardization Effort in Japan, *Proc. 26th Design Automation Conference*, ACM/IEEE (1989).

(平成元年12月 1 日受付)
(平成2年 4 月17日採録)



中村 宏（正会員）

昭和38年生．昭和60年東京大学工学部電子工学科卒業．平成2年同大学大学院工学系研究科電気工学専攻博士課程修了．工学博士．同年筑波大学電子情報工学系助手．論理設計支援に関する研究に従事．設計自動化，計算機アーキテクチャなどに興味を持つ．IEEE 会員．



久木元裕治（正会員）

昭和42年生．平成元年東京大学工学部電子工学科卒業．現在同大学院工学系研究科情報工学専攻修士課程在学中．論理設計支援に関する研究に従事．

田中 英彦（正会員）

昭和18年生．昭和40年東京大学工学部電子工学科卒業，昭和45年同大学院博士課程修了．工学博士．同年東京大学工学部講師，昭和46年助教授，昭和62年教授．昭和53年～54年ニューヨーク市立大学客員教授，現在に至る．計算機アーキテクチャ，並列推論マシン，知識ベース，オブジェクト指向プログラミング，分散処理，CAD，自然言語処理，等の研究を行っている．‘計算機アーキテクチャ’，‘VLSI コンピュータⅠ，Ⅱ’，‘ソフトウエア指向アーキテクチャ’（いずれも共著），‘情報通信システム’著．電子情報通信学会，人工知能学会，日本ソフトウエア科学会，IEEE，ACM 各会員．