Tokioかける言語
－Prologの自然な拡張－

青柳 龍也　藤田 昌宏*　元岡 達
（東京大学工学部，* 現在 富士通研）

Abstract

Tokio is a programming language based on temporal logic, and is considered as a natural extension of Prolog for parallel programming.

A set of Horn clauses including temporal operators is a Tokio program. If there is no temporal operator in the clauses, Tokio works like Prolog, and they are executed at Tokio clock moment.

Tokio has many useful Temporal operators. One defines declarative processes, and another one defines procedural processes. The first kind of Temporal operators (next,always etc.) comes from Linear Time Temporal Logic, and is evaluated at each Tokio clock moment. The second one (chop) comes from Interval Temporal Logic, and defines the ordering of several process executions.

Tokio variables may have different values in the Tokio clock's future. Various values can be assigned many times to the same Tokio variable at different times. Their values are linked by 'next'. This allows us to describe "increment of variable value", "exchange between two variables" etc., which can't be written in Prolog. Furthermore, in Tokio we can write elegant loop structures, because Tokio does not need any new variable at recursive calls. Tokio does not need any ugly repeat-fail loop.

Tokio has very useful functions for looping and status transition. And like other logic programming languages, its mathematical base is also sound. This is very useful for many applications, such as hardware description, formal verification of program, etc.

1. Why Tokio?

論理型プログラミング言語Prologは、一応一階述語論理に基づいているため、Pascal等のプログラミング言語にはない次のような特長を備えている。

1）プログラムの意味が明快で、正当性の証明がしやすい。

2）自然に並列性を含んだ記述になる（並列に実行しても意味が変わらない）。

3）不定部分を含む構造データが論理変数によって扱える。

一方、従来のプログラミング言語では容易に記述できるが、Prologでは記述しにくい例も多く存在する。その代表的なものは状態の変化を記述するもので、例えば、ある変数の値を1つふやす

X：＝X＋1

をPrologで記述しようとすると、値を返すための引数を1つふやして、

```
add1(X, Y) :－Y is X+1.
```

とするか、変数名とその値との対応をファクトとして記憶しなければならない。

```
add1(X) :－
  (retract (value (X, V)) ;true), !,
   V1 is V+1,
   assert(value (X, V1)).
```

これは、Prologの論理変数が一回の束縛しか許さないことから生じる。

ところで、従来のプログラミング言語の式をもう一度みてみると、式の右辺と左辺のXは、同じ変数でありながら違う値を持っている。同じ変数が違う値を持つということは、変数名というものが存在するために可能になっているとも考えられるが、式の右辺と左辺でXを評価する時刻が違うために可能になっているとみることもできる。すなわち、従来のプログラミング言語には、暗に時間の概念が含まれている。

そこで、Prologに時間の概念を導入することができれば、先にあげたPrologの特長を保持しつつ、Prologの記述しにくい部分を大幅に減少させることが可能になる。Tokioはそのような言語として設計された。

以下、次章でTokioのプログラミングに必要となる考え方について述べ、3章でプログラム例を与える。4章でTokioの形式的定義を、5章でTokioの位置づけを述べる。最後に、6章でTokioの未来を語る。

2. What's Tokio

2.1 Tokio時間

Tokioにおける時間は離散時間で、時刻は整数によって表現するものとする［図 2.1］。

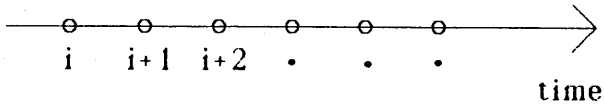


Fig 2.1 Tokio time

また、ある2つの時刻の間をインターバルといい、インターバルは2つの整数の対＜I.beg，I.fin＞で決まり、その対またはインターバルをIで表わすことにする。このとき、

I.beg ≦ I.fin

は常に成立していなければならない［図 2.2］。

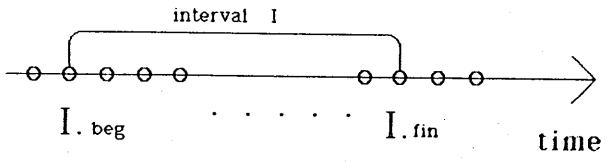


Fig 2.2 Interval

2.2 時間に関する実行制御

Prolog に時間を導入するためには、Prolog のゴールがTokio時間のどこで実行されるかを制御する時相演算子が必要である。Tokio時間のどこでというときにはインターバルを使って指定する。すなわち、あるインターバルIで実行されるというように表現する。実際には、そのインターバルの最初の時刻I.beg で実行されるものとする。

まず、時相演算子をまったく含まないTokioプログラムは、すべて同一インターバル（の最初の時刻）で実行されると考えてよい。この場合の動作はProlog とまったく同じになる。例えば、

P：－Q，R.

という節のQ，Rというゴールは同一インターバルI（の最初の時刻I.beg ）で実行される［図 2.3］。

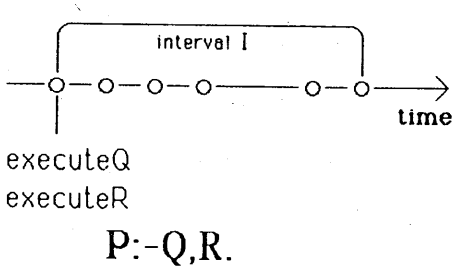


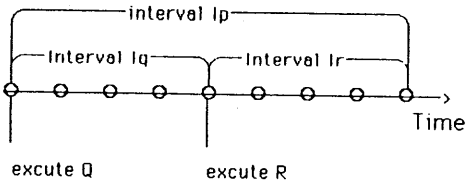
Fig 2.3 No temporal operator

&&

次に、2つのゴールの実行順序を記述するchop演算子を導入する。chop演算子は&&で表わす。

P：－Q &&& R.

chop演算子は、Pの実行されるインターバルIp を2つにわけ、前半のインターバルIq でQが、後半のインターバルIr でRが実行されることを表わす［図 2.4］。



P :- Q && R

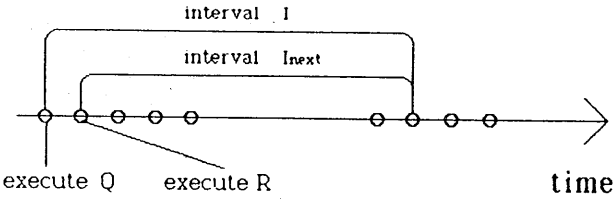
Fig 2.4 Chop operator

@

次の時刻での実行はnext演算子@によって表現する。次の時刻というのは、正確には、現在のインターバル（I.beg，I.fin ）に対して、（I.beg ＋1，I.fin ）というインターバルの先頭の時刻という意味である。

P：－@Q.

Pが実行される時刻の次の時刻でQが実行される［図 2.5］。



P:- Q,@R.

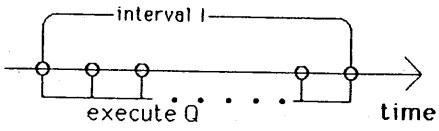
Fig 2.5 Next operator

#

あるインターバル中のすべての時刻であるゴールを実行するのは、always演算子#で表わす。

P：－#Q.

Pが実行されるインターバル中のすべての時刻でQが実行される［図 2.6］。



P:- #Q

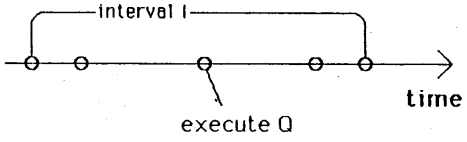
Fig 2.6 Always operator

<>

あるインターバル中のどこかの時刻であるゴールが実行されることは、sometime演算子<>で記述する。

P：－<>Q.

Pが実行されるインターバル中のどこかの時刻でQが実行される［図 2.7］。



P:-◇Q

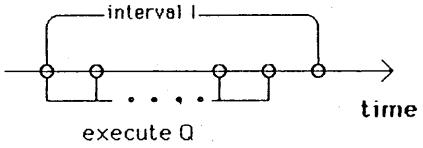
Fig 2.7 Sometime operator

keep

あるインターバルの最後の時刻以外であるゴールを実行することは、keep演算子で表わす。

P：－keep（Q）.

Pが実行されるインターバルＩp の最後の時刻Ｉp．fin 以外の時刻でQが実行される［図 2.8］。



P:- keep(Q)

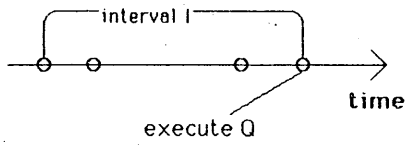
Fig 2.8 Keep operator

fin

あるインターバルの最後の時刻であるゴールを実行するには、fin 演算子を使う。

P：－fin （Q）.

Pが実行されるインターバルＩp の最後の時刻Ｉp．fin でQが実行される［図 2.9］。



P:- fin(Q)

Fig 2.9 Fin operator

2．3　Tokio変数

Tokioの変数は、Prolog の論理変数をすべての時刻に一対一対応するように並べたものである。すなわち、Tokio変数は、時刻ごとに別な値をもち、そのそれぞれがProlog の論理変数であるような論理変数の並びである［図 2.10 ］。

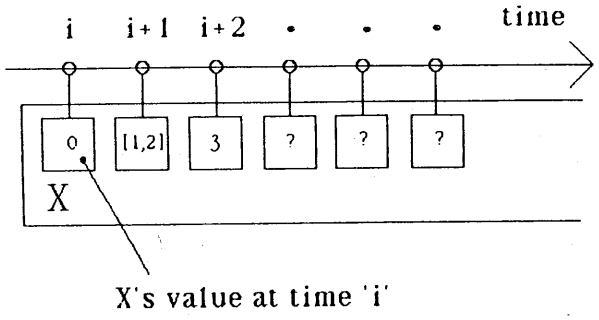


Fig 2.10 Tokio variable

2．4　変数の評価

通常Tokio変数は、すべての時刻の値を全部もっている。ゴールの実行に際しては、すべての時刻にわたってユニフィケーションが行なわれる。

一方、変数のある時刻についての値のみを対象とする述語も必要となる。

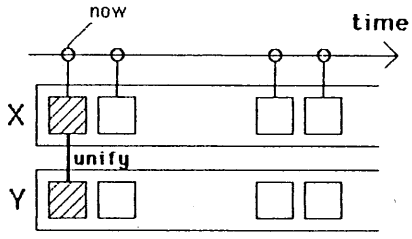
変数のある時刻での値を取り出すことを、変数をその時刻において評価するといい、そのような述語を評価型述語という。

＝

まず、＝は、現在の値と現在の値をユニファイする。

X＝Y.

すべての時刻にわたって値をもつTokio変数のX，Yから現在の値を取出して、ユニファイする［図 2.11 ］。



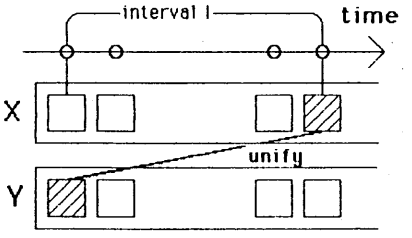
X = Y

Fig 2.11 Unify at now '='

<－

次に<－は、現在（のインターバルＩの最初の時刻Ｉ．beg ）の値を、インターバルの最後（Ｉ．fin ）の値にユニファイする［図 2.12 ］。

X<－Y.



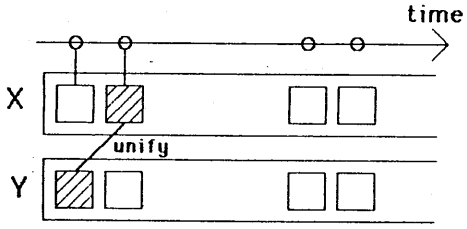
X ← Y

Fig 2.12 Temporal assignment '<-'

@

さらにnext演算子@を変数につけると、次の時刻からの論理変数の並びになる。すなわち、Xを現在から先のすべての時刻の値を持ったTokio変数とすると、@XはXの値のうち次の時刻から先のすべての値を持ったTokio変数である。この@と＝を組合せると、現在と次の時刻の値とのユニフィケーションが可能になる［図 2.13 ］。

@X＝Y



@X = Y

Fig 2.13 Unification between now and next

2．5　変数の型

Tokioの変数は、本質的には論理変数のみであるが、プログラミングをしやすくするために、いくつかの型の変数を導入する。

stable

stable（X）という述語を実行すると、その述語が実行されたインターバル中のすべての時刻で変数Xは同じ値をとる。stable（X）は、

```
stable（X）　：－　keep（@X＝X）.
```

で定義される。

global

global（［atom1，atom2，．．．，atomN］）という述語を実行すると、リスト中のアトムはglobal変数の変数名として登録される。それ以降、＊変数名という形式で、普通のTokio変数として使われる。global変数は、

```
a （X，Y）：－
    X＝1，Y＝1，b （X），c （Y）.
b （X）：－X　is．．．
c （Y）：－Y　is．．．
```

を

```
global（［x，y］）.
a ：－＊x ＝1，＊y ＝1，b，c .
b ：－＊x　is．．．
c ：－＊y　is．．．
```

のように記述することを可能にする。すなわち、global変数は、いくつかの節の間で引数として常に渡される同一の意味を持った変数の記述を減らすための前処理と考えてもよい。

static

static変数は、一度値が設定されると、次に値が設定されるまでその値を保つ。これはPascal 等の変数と同じような変数であるが、Tokioでは、インターバルの決め方に関するバックトラックがあるため、以前の値に戻る必要が生じるという点が異なる。

static変数は、値の設定をしないインターバルでその変数を明示的にstableにすることと等価であるが、その記述を減らしてプログラミングをしやすくするためにシステムでサポートされている。

static変数は、static（［atom1，．．．，atomN］）という述語で宣言され、それ以降、＊変数名という形式で使われる。

：＝，＜＝

また、static変数s に対する値の設定は、

```
＊s　：＝　1
＊s　＜＝　1
```

の２種類がある。前者は現在時刻で値を設定し、後者はインターバルの最後で設定する。それぞれTokio変数に対する＝，＜－に対応する。

2．6　インターバルの決定

インターバルＩは、２つの整数の対＜Ｉ．beg ，Ｉ．fin ＞であるが、Ｉ．fin の値は実行時に非決定的に決まる。

Tokioは現在の時刻をＩ．beg としてゴールを実行する。時相演算子をまったく含まないゴールは、実行終了とともにインターバルも終わり、Ｉ．finはＩ．beg ＋Ｎに値が決まる。ここでＮは０以上の整数で、非決定的に決まる。すなわち、バックトラックによってＮの値は様々な値をとる。

length

インターバルの長さを明示的に制限するために、length述語が用意されている。length（Ｎ）はＩ．fin の値をＩ．beg ＋Ｎに設定する。

empty

empty はlength（０）のことである。

時相演算子はその意味に応じてインターバルを決める働きをする。

chop演算子

まずchop演算子について考える。

```
P　&&　Q
```

というゴール全体のインターバルをＩ、PとQのインターバルをそれぞれＩp ，Ｉq とすると、直観的には、図 2.14 のようにインターバルが決まればよい。

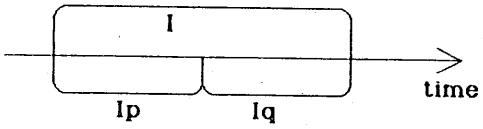


Fig 2.14 Interval generation
by chop operator

PとQは時相演算子を含まないものとし、Tokioプログラムの実行に即して考えると、次のようになる。

1）現在の時刻I．beg でP&&Qを実行しようとする。

2）Ip．beg をI．beg とし、Pを実行する。

3）Pは時相演算子を含まないので、インターバルは終了し、

Ip．fin ＝Ip．beg ＋Np

となる。ここでNp は非決定的に決まる。

4）Iq．beg をIp．fin とし、現在の時刻をIq．beg としてQを実行する。

5）Qは時相演算子を含まないので、インターバルは終了し、

Iq．fin ＝Iq．beg ＋Nq

となる。ここでNq は非決定的に決まる。

6）P&&Q全体の実行が終了し、I．fin はIq．fin になる。

次にlength指定のある場合を考える。例えば、

length（2），（P&&Q）

の場合、P&&Qは上の6）まで同じように実行され、I．fin が決まる。length指定があるので、I．fin の値がI．beg ＋2でないときは、バックトラックが生じて、Np，Nq の別の解をみつけにいく。

Prolog の非決定的動作は、節の出現順に次の解を求めるように決められている。出現順という順序づけには論理的な意味はないが、プログラムの実行という点からみると大きな影響を及ぼす。Tokioのlengthの決め方にも同様の問題がある。Np，Nq の解も論理的にはどのような順で決めてもよいはずだが、Tokioプログラムの実行には、その順序が大きく影響する。

現在のインタープリタでは、

1）length指定のある場合には、それに従う。他の解は無い。

2）length指定の無い場合には、1，2，3，．．．Nの順に決まる。ただし、Nは親のインターバル（Pのインターバルに対して、P&&Qのインターバルを親のインターバルとする）のlengthによって制限される。

3）0。ただし、2）でNの制限が無いときには3）には来ない。

と決められている。

例えば、

length（2），（P&&Q）

では、P，Qの実行されるインターバルの長さは、親のインターバルのlength指定により制限されて、1，2，0の順に決まる。そのためP&&Qは、

P	Q
1	1
2	0

の順で成功する。

P&&Q

のようにlength指定の無い場合には、P，Qのインターバルの長さは、1，2，3，．．．の順に決まるが、Pのインターバルの長さは1以外にはならない。P&&Qは次の順に成功する。

P	Q
1	1
1	2
1	3
1	4
：	：

@演算子

P，@Q

というゴール全体のインターバルをI、Qの実行されるインターバルをIq とすると、直観的には、図2.15 のようにインターバルが決まればよい。ここで、Pと@Qはchop演算子で区切られているわけではないので、同じインターバルIで実行されることに注意しなければならない。インターバルIでのPと@Qの実行は、インターバルIでのPの実行と、インターバルIq でのQの実行にわけられる。

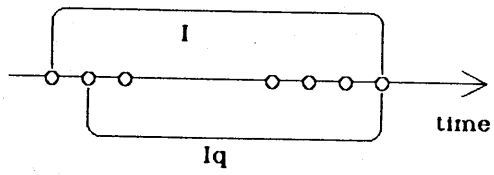


Fig 2.15 Interval generation
by next operator

PとQは時相演算子を含まないものとして、Tokioプログラムの実行を考えると、

1）現在の時刻I．beg でP，@Qを実行しようとする。

2）Pを実行する。しかし、同一インターバルで@Qも実行しなければならないので、I．fin はまだ決まらない。

3）現在の時刻Ｉ．beg で@Qを実行しようとする。

4）現在の時刻をＩq．beg ＝Ｉ．beg ＋1としてQを実行する。

5）Qは時相演算子を含まないのでインターバルは終了し、

Ｉq．fin ＝Ｉq．beg ＋Nq

となる。ここでNq は非決定的に決まる。

6）P，@Q全体の実行が終了し、Ｉ．fin はＩq．fin になる。

length指定があれば、Ｉ．fin は既に決まっていて、Nq はそれによって決まる。

#演算子

＃P

というゴール全体のインターバルをＩとすると、PはＩ，Ｉp 1，Ｉp 2，．．．Ｉpnで実行される。直観的には、図2．16のようにインターバルが決まればいい。

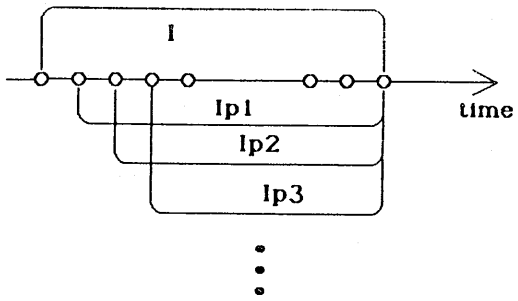


Fig 2.16 Interval generation by always operator

Pには時相演算子が含まれないものとして、Tokioの実行は、

1）現在の時刻Ｉ．beg で＃Pを実行しようとする。

2）現在の時刻Ｉ．beg でPを実行する。

3）Pには時相演算子が含まれないので、インターバルは終了し、

Ｉ．fin ＝Ｉ．beg ＋N

となる。ここでNは非決定的に決まる。

length指定があれば一意に決まる。

4）以下、図のようにＩp 1，Ｉp 2，．．．というインターバルを生成して、Pを実行する。

2．7　バックトラック

Tokioにおけるバックトラックには2種類のバックトラックがあると考えてよい。1つは普通のProlog におけるバックトラックと同じもので、もう1つはインターバルの長さの決め方に関するバックトラックである。

3．　Programming In Tokio

3．1　簡単な例

図3．1に、簡単なプログラムの実行例を示す。2章の説明から動作は明らかであろう。？－はProlog 同様、入力の要求であり、ｔ０，ｔ１等は、Tokioのデバッガが出力する現在時刻の情報である。

```
| ?- length(2),(write(0) && write(1)).

t0:0
t1:1
t2:
yes
| ?- length(2),(write(0) && write(1)) && fail.

t0:0
t1:1
t2:
t1:
t2:1
no
| ?- length(2),#write(0) && length(3),#write(1).

t0:0
t1:0
t2:01
t3:1
t4:1
t5:1
yes
| ?- length(2),@write(0) && length(2),#write(1).

t0:
t1:0
t2:1
t3:1
t4:1
yes
| ?- length(2),<>write(0).

t0:
t1:0
t2:
yes
| ?- length(2),<>write(0) && fail.

t0:
t1:0
t2:
t1:
t2:0
no
| ?- length(2),keep(write(0)).

t0:0
t1:0
t2:
yes
| ?- length(2),keep(write(0)) && length(3),#write(1).

t0:0
t1:0
t2:1
t3:1
t4:1
t5:1
yes
```

```
  ¦ ?- length(2),fin(write(0)).
 t0:
 t1:
 t2:0
 yes
 ¦ ?- length(3),X=1,#write(X).

 t0:1
 t1:_87
 t2:_387
 t3:_597
 X = 1 ;

 no
 ¦ ?- length(3),@X=1,#write(X).

 t0:_3
 t1:1
 t2:_400
 t3:_690
 X = _3 ;

 no
 ¦ ?- length(3),X=1,@X=X+1,#write(X).

 t0:1
 t1:2
 t2:_536
 t3:_826
 X = 1 ;

 no
 ¦ ?- length(3),X=1,X<-X+1,#write(X).

 t0:1
 t1:_99
 t2:_624
 t3:2
 X = 1 ;

 no

 ¦ ?- length(3),X=1,keep(@X=X+1),#write(X).

 t0:1
 t1:2
 t2:3
 t3:4
 X = 1 ;

 no
 ¦ ?- [user].

 t0:add1(X) :: X<-X+1.
 ¦ user consulted 104 bytes 0.116687 sec.

 yes
 ¦ ?- length(2),X=1,add1(X),#write(X).

 t0:1
 t1:_94
 t2:2
 X = 1 ;

 no
 ¦ ?- [user].

 t0:counter(X) :: #(@X=X+1).
 ¦ user consulted 120 bytes 0.100028 sec.

 yes

 ¦ ?- length(5),X=1,counter(X),#write(X).

 t0:1
 t1:2
 t2:3
 t3:4
 t4:5
 t5:6
 X = 1 ;

 no
```

Fig 3.1 Simple examples

3．2 メモリの記述

Tokioでメモリを記述すると図3．2のようになる。メモリの内容はリストで表現されている。変数Contents，Cmd，Data，Adrは、ほとんどの節に出現する。これらをいちいち引数の中に記述するのは、プログラムの読みやすさ書きやすさの点から好ましくない。これらの変数をglobal変数として扱うと、図3．3のようなプログラムになる。
図3．2と図3．3のプログラムの意味は同じである。

3．3 ユニファイプロセッサ

図3．4はTokioによるユニファイプロセッサ（UP）の記述である。UPは定義節とゴール説の単一化を行なう。プログラム中の変数名で、d _は定義節に関する変数、g _はゴール節に関する変数を表わしている。

単一化アルゴリズムは次のようになっている。まずゴールの長さを調べ、0ならば単一化は成功する。0以外の場合は次の要素を取り出す。変数の場合には変数のたぐりを行なう。ゴール側の要素と定義側の要素によって5つの場合にわかれる。

1）両方が未定義
2）3）一方が未定義
4）両方がリスト
5）それ以外

1）2）3）の場合は適当な変数の束縛が起こる。4）の場合は現在の状態をスタックに積んで、自分の要素の単一化を実行する。5）の場合は両者が同じ要素ならば成功し、それ以外は失敗する。

testという節によって、このプログラムは実行される。testは、appendの定義とゴールを含むようにメモリを初期化する。メモリには、INT，ATOM，LISR，VAR，UNDEFの5つのタグがある。節の最初の要素はその節の長さを表わす。次のインターバルでinitが実行される。initは最初のデータを読み、running flagをセットし、loop_unifを実行する。

loop_unifは節の長さを調べ、0ならばスタックの深さを調べる。スタックに何もなければ単一化は成功する。スタックに何かあれば、それを取り出して実行を続ける。次のインターバルで変数のたぐりを行なう。このインターバルではゴール側のメモリアクセスと定義側のメモリアクセスが衝突する可能性がある。これを避けるために、fetch _unif2ではg _bus とd _bus というフラグを調べている。

```
store(Adr,Value0,Contents) :-
    Value = Value0,
    Now = Contents,
    store0(0,Adr,Value,Now,Next),
    Contents <- Next.
store0(Adr,Adr,Value,[_|Tail],[Value|Tail]) :- !.
store0(I,  Adr,Value,[V|Now],[V|Next]) :-
    I1 is I+1,
    store0(I1,Adr,Value,Now,Next).

fetch(Adr,Value0,Contents) :-
    Now = Contents,
    fetch0(0,Adr,Value,Now),
    stable( Contents ),
    Value0 <- Value.
fetch0(Adr,Adr,Value,[Value|_]) :- !.
fetch0(I,  Adr,Value,[_|Now]) :-
    I1 is I+1,
    fetch0(I1,Adr,Value,Now).

memory(Adr,Cmd,Data,Contents) :- (
    if Cmd=read       then fetch(Adr,Data,Contents)
    else if Cmd=write then store(Adr,Data,Contents)
    else stable(Contents)) &&
        memory(Adr,Cmd,Data,Contents).

test :- store(0,0,Contents) && store(3,1,Contents) &&
        stable(Contents) &&
        fetch(0,A,Contents),fetch(3,B,Contents),
        fin(write( (A,B) )).

test2 :- (
    Cmd=write,Data=1,Adr=0 &&
    Cmd=write,Data=2,Adr=3 &&
    Cmd=off &&
    Cmd=read,Adr=0 &&
    Cmd=off &&
    Cmd=read,Adr=3 &&
    Cmd=off ),
    memory(Adr,Cmd,Data,Contents),
    #write((Adr,Cmd,Data)),#nl.
```

Fig 3.2 Memory system

```
% memory using global variable

global([cmd,adr,data,contents]).

memory :- (
    if *cmd=read       then fetch
    else if *cmd=write then store
    else stable(*contents)) &&
        memory.

store :-
    Value = *data,
    Now = *contents, Adr = *adr,
    Value = *data,
    store0(0,Adr,Value,Now,Next),
    *contents <- Next.

fetch :-
    Adr = *adr, Now = *contents,
    fetch0(0,Adr,Value,Now),
    stable( *contents ),
    *data <- Value.


test3 :- (
    *cmd=write,*data=1,*adr=0 &&
    *cmd=write,*data=2,*adr=3 &&
    *cmd=off &&
    *cmd=read,*adr=0 &&
    *cmd=off &&
    *cmd=read,*adr=3 &&
    *cmd=off ),
    memory,
    #{  Adr = *adr, Cmd = *cmd,
        Data = *data,
        write((Adr,Cmd,Data)),
        nl }.
```

Fig 3.3 Memory system using global variables

4. Tokio基礎論

4.1 Tokioと時相論理

TokioはLiner Time Temporal Logic（LTTL）[1]と呼ばれる論理に基づく。

LTTLは、時刻に対して真偽値の決まる論理であり、基本的な演算子は次の3つである。

- ○p　次の時刻でp が成立
- □p　以後のすべての時刻でp が成立
- p U q　q が成立するまでのすべての時刻でp が成立

LTTLは古典論理を含んでいる。LTTLは時間の進行に沿って展開していくことにより、検証が可能である。Tokioの実行が時間に沿って行なわれるのはこのためである。

LTTLの記述は事実上□とUを使った記述になる。□とUはどちらもすべての時刻でp が成立するという意味なので、宣言的なプロセスを構成する。しかし、プロセスの順序性を記述するためには、一つの順序について一つの変数を必要とする。

これに対し、Interval Temporal Logic（ITL）[2]は、より容易に順序性を記述することができる。

ITLの真偽値は、時間の区間に対して決まり、前半の区間でp が成立し、後半の区間でq が成立することを、p &&q という形で表現できる。しかし、ITLは決定不能である。

ITLを決定可能とするためには、変数に対してその値が時刻のみで決定するという制限を加えればよい。この性質はlocal と呼ばれる。LocalITL

```
static([
    length,memory(_),
    stack_depth,run,
    return_code,
    d_cell,d_addr,d_mem,
    g_cell,g_addr,g_mem,
    stack_ln(_),
    stack_ga(_),
    stack_da(_)
]).

global([
    g_bus,d_bus
]).
```

```
fetch_unif0g:-
    fetch(*g_cell,*g_addr,*g_bus),
    *g_mem <= *g_addr  &&
    if *g_cell..tag = 'VAR'
        then fetch_unif1g.

fetch_unif1g:-
    fetch(*g_cell,*g_cell,*g_bus),
    *g_mem <= *g_cell  &&
    if *g_cell..tag = 'VAR'
        then fetch_unif1g.

fetch_unif0d:-
    fetch(*d_cell,*d_addr,*d_bus),
    *d_mem <= *d_addr  &&
    if *d_cell..tag = 'VAR'
        then fetch_unif1d.

fetch_unif1d:-
    fetch(*d_cell,*d_cell,*d_bus),
    *d_mem <= *d_cell  &&
    if *d_cell..tag = 'VAR'
        then fetch_unif1d.
```

Fig 3.4 Unify processor

```
init:-
    *length <= (*memory(*g_addr)..data - 1,'INT',0),
    *d_addr <= (*d_addr..data + 2,'INT',*d_addr..map),
    *g_addr <= (*g_addr..data + 2,'INT',*g_addr..map),
    *stack_depth <= 0,
    *run <= 1
    &&  loop_unif.

loop_unif:-
    if *length..data > 0
        then fetch_unif1
        else if *stack_depth = 0
            then (*return_code <= (0,'INT',0),
                *run <= 0  && 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
                    &&  fetch_unif1.

fetch_unif1:-
    fetch_unif0g,
    fetch_unif0d &&
    *length <= (*length..data - 1,'INT',0),
    *g_addr <= (*g_addr..data + 1,'INT', *g_addr..map),
    *d_addr <= (*d_addr..data + 1,'INT', *d_addr..map) &&
    fetch_unif2.

fetch_unif2:-
    if *g_mem = *d_mem
        then loop_unif
           else case2([*g_cell..tag,*d_cell..tag],
          [[['UNDEF'],['UNDEF']],
            (store(*g_mem,*d_mem, *g_bus) && loop_unif)],
          [[['UNDEF'],[not,'UNDEF']],
            (store(*g_mem,*d_cell,*g_bus) && loop_unif)],
          [[[not,'UNDEF'],['UNDEF']],
            (store(*d_mem,*g_cell,*d_bus) && loop_unif)],
          [[['LIST'],['LIST']],
            (if *length..data > 0
                then ((*stack_depth <= *stack_depth + 1,
                        S<— *stack_depth,
                        *stack_ga(S) <= *g_addr,
                        *stack_da(S) <= *d_addr,
                        *stack_ln(S) <= *length,
                        *length <= (2,'INT',g),
                        *g_addr <= *g_cell,
                        *d_addr <= *d_cell) && fetch_unif1)
                else loop_unif)],
           [[otherwise],
            (if (*g_cell..tag =\= *d_cell..tag ;
                 *g_cell..data =\= *d_cell..data)
              then ((*return_code <= 'FAIL', *run <= 0) && idle)
              else loop_unif)]]).

fetch(Data,Adr,Bus):-
    Adr = (Address,_,Map),Map=Bus,
    D = *d_bus,G = *g_bus,
    if D==G
        then if Bus = d
            then (true && fetch(Adr,Data,Bus))
            else Data <= *memory((Address,_,Map))
        else Data <= *memory((Address,_,Map)).

store(Adr,Data,Bus):-
    (Address,_,Map)<—Adr,Map=Bus,
    D = *d_bus,G = *g_bus,
    if D==G
        then if Bus = d
            then true && fetch(Adr,Data,Bus)
            else *memory((Address,_,Map)) <= Data
        else *memory((Address, ,Map)) <= Data.
idle:-  dump,
    if *run = 1
        then (true && init).
```

にはＬＴＴＬへの変換が存在する。このような変換に基づいてＬｏｃａｌＩＴＬを実行していくのがＴｏｋｉｏである。すなわち、ＴｏｋｉｏはＬｏｃａｌＩＴＬをＬＴＴＬによって実行する系である。

４．２　Ｔｏｋｉｏの形式的定義

インターバルＩは２つの整数の対、＜Ｉ．ｂｅｇ，Ｉ．ｆｉｎ＞である。２つの整数の間には、

Ｉ．ｆｉｎ ≧ Ｉ．ｂｅｇ

が常に成り立っていなければならない。

３つのインターバルＩｐｑ，Ｉｐ，Ｉｑについて、次の関係を定義する。

Ｉｐｑ＝Ｉｐ＋Ｉｑ　≡
　Ｉｐｑ．ｂｅｇ＝Ｉｐ．ｂｅｇ ∧
　Ｉｐｑ．ｆｉｎ＝Ｉｑ．ｆｉｎ ∧
　Ｉｐ．ｆｉｎ＝Ｉｑ．ｂｅｇ

まず命題論理の範囲で考える。Ｐ，ＱをＴｏｋｉｏの命題とする。

Ｐが時相演算子を含まないとき、Ｐはインターバルｉｐについて真偽値を持つ。これを、

Ｉｐ ⊨ Ｐ

と表わす。Ｐが時相演算子を含まないとき、

Ｉｐ０．ｂｅｇ＝Ｉｐ１．ｂｅｇ　→
　Ｉｐ０⊨Ｐ＝Ｉｐ１⊨Ｐ

を公理として導入する。この公理によりＰ，Ｑの真偽値は、インターバルの最初で局所的に決まることになる。

∧，～に関しては、

Ｉ⊨Ｐ∧Ｑ　≡　Ｉ⊨Ｐ∧Ｉ⊨Ｑ
Ｉｐ⊨～Ｐ　≡　～Ｉｐ⊨Ｐ

インターバルの長さに関する述語ｌｅｎｇｔｈとｅｍｐｔｙは、

Ｉ⊨ｌｅｎｇｔｈ（Ｎ）　≡
　Ｉ．ｆｉｎ－Ｉ．ｂｅｇ＝Ｎ
Ｉ⊨ｅｍｐｔｙ　≡　Ｉ．ｆｉｎ＝Ｉ．ｂｅｇ
Ｉ⊨ｎｏｔ ｅｍｐｔｙ　≡　Ｉ．ｆｉｎ＞Ｉ．ｂｅｇ

時相演算子は、

Ｉｐｑ⊨Ｐ＆＆Ｑ　≡
　∃Ｉｐ，Ｉｑ　Ｉｐｑ＝Ｉｐ＋Ｉｑ
　∧Ｉｐ⊨Ｐ∧Ｉｐ⊨ｎｏｔ ｅｍｐｔｙ
　∧Ｉｑ⊨Ｑ
Ｉｐ⊨@Ｐ　≡
　∃Ｉ０，Ｉ１　Ｉｐ＝Ｉ０＋Ｉ１
　∧Ｉ０⊨ｌｅｎｇｔｈ（１）∧Ｉ１⊨Ｐ
Ｉｐ⊨＜＞Ｐ　≡
　∃Ｉ０，Ｉ１　Ｉｐ＝Ｉ０＋Ｉ１
　∧Ｉ１⊨Ｐ
Ｉｐ⊨＃Ｐ　≡
　∀Ｉ０，Ｉ１　Ｉｐ＝Ｉ０＋Ｉ１
　→Ｉ１⊨Ｐ
Ｉｐ⊨ｋｅｅｐ（Ｐ）　≡
　∀Ｉ０，Ｉ１　Ｉｐ＝Ｉ０＋Ｉ１
　→（Ｉ１⊨ｎｏｔ ｅｍｐｔｙ →Ｉ１⊨Ｐ）
Ｉｐ⊨ｆｉｎ（Ｐ）　≡
　∃Ｉ０，Ｉ１　Ｉｐ＝Ｉ０＋Ｉ１
　∧Ｉ１⊨ｅｍｐｔｙ ∧Ｉ１⊨Ｐ

と定義される。

次に一階述語論理への拡張を考える。それには、変数の値と関数の値の決め方を与えればよい。あるインターバルＩでの変数ｘ と関数ｆ の値を

Ｍ．Ｉ（ｘ），Ｍ．Ｉ（ｆ）

で表すと、変数の値について次の公理が成り立つ。

Ｉ ０．ｂｅｇ＝Ｉ １．ｂｅｇ　→
　Ｍ．Ｉ ０（ｘ）＝Ｍ．Ｉ １（ｘ）

関数の値については、

Ｍ．Ｉ（ｆ（ｘ０，．．．，ｘｎ））　≡
　Ｍ．Ｉ（ｆ）
　（Ｍ．Ｉ（ｘ０），．．．，Ｍ．Ｉ（ｘｎ））

となる。

４．３　ｎｅｘｔについて

以上のように定義すると、

＃Ｐ　⇔　～（＜＞（～Ｐ））

は成り立つが、

@～Ｐ　⇔　～@Ｐ

は成り立たない。

ｎｅｘｔ演算子は、長さ０のインターバルで次の時刻が存在しないときに真となるかどうかで２種類のｎｅｘｔがある。上の定義では長さ０のインターバルで@Ｐは偽となる。このようなｎｅｘｔをｓｔｒｏｎｇ ｎｅｘｔ と呼ぶ。また、長さ０のインターバルで真となるようなｎｅｘｔをｗｅａｋ ｎｅｘｔ といい、○で表わす。@と○の間には次のような関係がある。

○Ｐ　⇔　@Ｐ∨ｅｍｐｔｙ
@Ｐ　⇔　○Ｐ∧ｎｏｔ ｅｍｐｔｙ

また、このとき、

～○Ｐ　→　○～Ｐ
@～Ｐ　→　～@Ｐ

が成り立ち、さらに、

○Ｐ　⇔　～@～Ｐ
@Ｐ　⇔　～○～Ｐ

が成立する。

５．　Ｗｈｅｒｅ ｉｓ Ｔｏｋｉｏ?

５．１　Ｐｒｏｌｏｇ との比較

プログラミング言語としてのＰｒｏｌｏｇ の最大の特徴は論理変数である。論理変数は不定部分を含む構造体を扱うことを可能にし、Ｐｒｏｌｏｇ を従来のプログラミング言語より優れたものにしている。

一方、論理変数は、バックトラックを行なわない限り、一回の代入（束縛）しか許されない。そのため、Prolog では、変数の値を変更するような記述、例えば、

X＝X＋1

等の簡単な記述ができない。

Prolog の論理変数の良い性質を残したまま、より記述力のある変数に拡張するために、Tokioでは時価の概念を導入した。時間軸上に並んだ論理変数の列を一つのTokio変数として扱うことにより、多数回の代入が可能になる。

多数回代入できる変数による記述力の向上は単なる変数の値の変更に留まらない。変数の値を状態遷移と対応させることにより、様々な記述が可能になる。

まず、Prolog のrepeat－failループのような繰返しをより美しく書くことができる。repeat－failループは、バックトラックによって変数の値を変え、繰返す構造であるが、バックトラックを制御することはむずかしく、きれいなわかりやすいプログラムにはならない。Tokioでは、時間を進めることにより簡単に変数の値を変更できるので、＃とlengthを使った宣言的でわかりやすい記述が可能である。

一方、再帰呼び出しを使うと、Prolog でもrepeat－failループよりはきれいなプログラムを書くことができるが、毎回新しい変数を必要とする。例えば、

```
p(X) :- q(X, X1), p(X1).
```

というように、X1という変数が再帰呼び出しごとに生成される。Tokioでは、時間を進めることにより、

```
p(X) :- q(X) && p(X).
```

と書くことができて、新しい変数を必要としない。

再帰呼び出しのときに新しい変数を必要としないというTokioの特徴は、while 文のような抽象を可能にする。

```
while PdoQ  :-
  if  P then (Q&&while PdoQ)
        else  empty.
```

ここで、

```
if  G1 then G2 else G3
```

はProlog の

```
G1->G2;G3
```

と同じであり、while とdoはオペレータ宣言されている。また、P，Qにはゴールが束縛される（高階の述語）。

Prolog ではヘッドのP（下線）に含まれる変数とボディのP（下線）に含まれる変数が異ならないと再帰呼び出しが意味をなさないため、このような抽象は不可能である。

さらに、Tokioではstatic変数を導入しているため、手続き的なプログラミングが可能である。例えば、先にあげたユニファイプロセッサの例は、Prolog とはかなり異なる手続き的なプログラミングスタイルになっている。しかし、そのときでも、ある時刻たけをみれば、宣言的なプログラムのままである。このためTokioでは、短い時間で終わるようにプログラミングするとProlog に近づき、時間を進めるように書くと従来のプログラミング言語に近づく。

5．2　並列Prolog との比較

Tokioは並列性を記述する（もしくはシミュレートする）言語であって、並列実行の言語ではない。そのため、他の並列Prolog と直接比較することはできないが、記述の面での比較を行なう。

Prolog の並列処理方式には、AND並列、OR並列、ストリーム並列等がある。時相演算子を含まないToki プログラムは、AND並列をシミュレートしている。すなわち、そのプログラムは、Tokio時間内の同一時刻で実行される。OR並列は、時間軸に関するバックトラックによってシミュレートされる。

ストリーム並列については、Tokioの記述は他の並列Prolog の記述よりも優れている。concurrent prolog ［4］を例にとると、

```
?－producer(A), consumer(A?).
consumer([A|A´]) :-
    work(A),
    consumer(A´?).
```

というプログラムはTokioで、

```
?－#producer(A), #consumer(A).
consumer(A) :－work(A).
```

と書くことができる。ここで、workはAの現在の値を使うものとする。

Tokioもconcurrent prolog も、プロセス間の通信に共有変数を使う点は同じであるが、Tokioでは、ストリームを表現するリスト構造や読出し専用記号が必要なく、読出し専用記号をどこにつけるかで迷うこともない。

concurrent prolog では、プロセス間の同期にも変数を使う。concurrent prolog の

```
p(A), q(A?)
```

はTokioで、

```
p(A) && q(A)
```

と表現される。逆にTokioの

```
p &&q
```

をconcurrent prolog で書こうとすると、同期のた

めだけに変数が必要になる。

Tokioでは、状態遷移と変数の束縛ははっきり分離している。そのため、#を使ってプロセスを宣言的に書くことができる。

また、&&を使ってプロセスを記述することもできる。#を使ったプロセスの記述が、一時刻一時刻同期して動くプロセスをシミュレートするのに対して、&&を使ったプロセスの記述は、インターバルの決め方についての非決定性があるため、非同期に動くプロセスをシミュレートする。

5．3　時間の概念を持つ他の言語との比較

時間の概念を持つ他の言語としては、B．Moszkowskiの Tempura［3］や米崎のTemplog［5］などがある。

Tempuraは、ITLに基づく言語であるが、Tokioのような論理型言語とは異なる。Tempuraの変数は、従来のプログラミング言語の変数と同じである。また、Tempuraはインターバルの長さについての非決定性を持たず、インターバルの長さは常に与えられていなければならない。

Templogはassertによって時間が進む言語であり、Templogの論理変数は時間ごとに異なる値を持つことはない。

Tokioの特長はTokio変数に基づく部分が多い。すべての論理変数を、Tokio変数のように時間について拡張した言語は、Tokio以外には存在しない。

6．@Tokio

現在Tokio処理系としては、C－Prolog 上に作成したインタープリタが動いている。Tokioは並列性を記述できるが、スケジューリングは固定的であるため、Prolog のコンパイルコードを少し拡張することによって、コンパイラを生成することが可能である。コンパイラはC－Prolog 上に現在作成中であり、コンパイラが完成すれば十分な速度で動く処理系が得られるであろう。

高速な処理系が得られれば、Tokioで記述したハードウェアをTokio上で検証し実際のハードウェアにまでコンパイルする総合設計支援システムにまで発展させることが可能である。

ハードウェアの記述以外にも、時間の推移により新しい世界が得られることを利用して、可能世界の意味を持つ論理式をTokio上で直接実行することができるだろう。

参考文献

［1］P．Wolper，”Temporal logic Can Be More Expressive”，22nd Annual Symposium on Foundation of Computer Science，October 1981

［2］B．Moszkowski，”A Temporal Logic for Multilevel Reasoning about Hardware”，IEEE Computer Mabazine，February 1985

［3］B．Moszkowsky，”Executing Temporal Logic Program”，Technical Report No．55 University of Cambridge，Computer Laboratory 1984

［4］E．Shapiro，”A Aubset of Concurrent Prolog and its Interpreter”，TR－003，ICOT，1983

［5］米崎　直樹、”時間論理プログラミング言語Templog”、日本ソフトウェア科学会第1回大会論文集　1E－4