Tokio: Logic Programming Language Based on Temporal Logic and Its Compilation to Prolog

M.Fujita*, S. Kono**, H.Tanaka**, T.Moto-oka**

* FUJITSU LABORATORIES LTD.
** Department of Electronic Engineering, University of Tokyo

ABSTRACT

Tokio is a temporal logic programming language. It is a sophisticated extension of Prolog intended for specification of concurrent programs. Its basic execution is a resolution of Linear Time Temporal Logic [1]. Tokio also has an extension that can execute Interval Temporal Logic [2]. The resolution consists of three parts. These are: unification of temporal variables, reduction including temporal operators, and controlling intervals. We developed a Tokio compiler in Prolog.

1. Temporal logic from the point of view of logic programming

We have been studying hardware specification in a logic programming language [3]. Logic programming has many attractive points.

* Program validity is directly related to logical consequence.
* Unification serves as a powerful connection between procedures.
* It is easy to extract parallelism from a program.

However, logic programming is rather idealistic. In general, declarative meaning, which is the meaning of a logical formula, is somewhat independent of procedural meaning. Algorithm implementations are not described in detail in Prolog. For example, consider the difference between a loop and a recursion. Both repetitions have self-reference pointers for execution. The difference is that a recursion creates new environments in each cycle time, while a loop does not. In traditional languages, stack and frame are used explicitly.

For example, the loop is the one form of repetition available in Fortran. On the other hand, there is no loop in Prolog. This is because Prolog variables have no state, i.e. single assignment in nature. In order to make a loop meaningful, new variables must be created at each repetition. For this reason, loops cannot be directly expressed in Prolog. Needless to say, repeat-fail-loops destroy the relationship between declarative meanings and procedural meanings.

In logic programming, the order of execution is not strictly specified. Although this freedom is suitable for getting high concurrency from original program, detailed descriptions with full specification of complex timing are difficult.

Temporal logic is a promising tool for the design and description of hardware. A loop is represented using the state of a variable, and both parallelism and sequentiality are easily described using temporal operators. There are many kinds of temporal logic. Here we use two of them: Linear Time Temporal Logic (LTTL) [1] and Interval Temporal Logic (ITL) [2]. We have been used LTTL for hardware description [3]. Many researchers use propositional logic, because it has a decision procedure [4]. However, the compact descriptions of first order temporal logic make it desirable. Moszkowski developed a language called Tempura based on ITL, and implemented in lisp [5]. We here present a temporal logic programming language called Tokio based on a resolution of LTTL. Tokio Logic is an extension of LTTL containing some temporal operators of ITL. Tokio can be considered a logic programming version of Tempura.

Tokio is used primarily for hardware specification and simulation. Tokio is expressive and executable enough for hardware description. Execution of Tokio is not efficient in itself. Tokio consumes a lot of memory in backtracking and a lot of time in unification of temporal logic variables. Like other languages, fast, efficient execution is of primary importance. However, a much more important issue is to establish a method of translating Tokio to a real implementation (i.e. silicon compiler). We anticipate the output of the compiler to be Register Transfer Level hardware description language, or some other practical concurrent languages such as GHC [7]. Tokio supports simulation of early-time-specification, which is an important hierarchal design tool. Our final goal is logic circuit synthesis using Tokio.

In the rest of this chapter, we briefly introduce LTTL and ITL. In chapter 2, we discuss the basic execution mechanism of Tokio. The execution of Tokio consists of three parts: unification of temporal variable, reduction including temporal operators and an extension with ITL's operators. In chapter 3, we presents a sample hardware description. The example is a hardware sorter for a database machine [6]. The technique of compiling Tokio to Prolog is described in chapter 4. In the last chapter, we discuss the relation among GHC [7], Tempura, T-Prolog [15] and Tokio.

1.1. Linear Time Temporal Logic

Tokio is based on Linear Time Temporal Logic (LTTL). LTTL is based on discrete time concepts, i.e. it has a minimum unit of time. Variable and predicate meanings are determined for each state, i.e. instants in time. Therefore, variables may have many values depending on the time at which they are evaluated. There are three major temporal operators in LTTL: next or @, always and until.

First we discuss the @ operator. @P means P is true at the next state, i.e. next clock period.

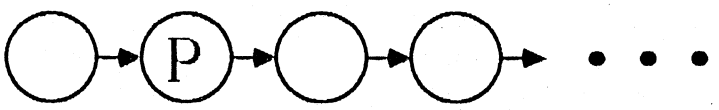

Fig. 1 @P next operator

There are two kinds of equality in Tokio. The first kind of equality is an identity. This type of equality means two values are equal in all states. This is also called temporal equality. The other kind of equality is single state equality, that is, the two variables' values are equal only at the present instant and not necessarily equal in future states (Fig. 2).

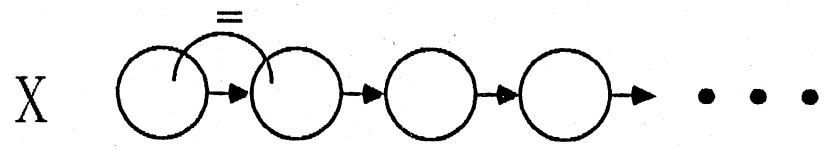

Fig. 2 Reference to Next Time Value

The syntax of Tokio is very similar to that of DEC-10 Prolog [8]. Words beginning with a capital letter are variables. In (1), = symbolizes single state equality. The two X's denote the same variable. These two X's are equal in all states. The value of X at present is equal to the value of X in the next state.

```
@X = X                                  (1)
```

The next major operator is the always operator. Always describes a predicate that is true in all states (Fig.3).

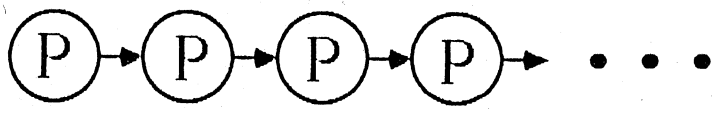

Fig. 3 #P Always Operator

Instead of the square notation used in [1], we symbolize always with #. # is defined recursively using the next operator. This operator is a little different from the @ operator. However as far as LTTL concerned, two operators are same.

```
#P :- P,next(#P).                       (2)
```

':-', ',' and '.' indicate implication, conjunction, and termination, respectively. In Tokio, a clause is an axiom, i.e. it is true for all states.

Finally we introduce the until operator.

```
p until q                               (3)
```

This means that p is true for all states until the state in which q becomes true (Fig.4).

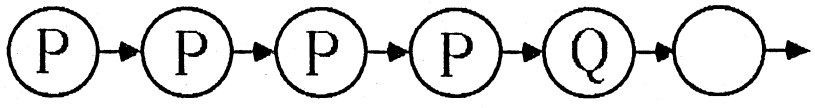

Fig. 4 P until Q

Until can also be defined using the @ operator.

```
P until Q :- Q,!.                       (4)
P until Q :- P,@(P until Q).
```

Note that the cut operator of first clause indicates negation of Q in a second clause. In order to satisfy the close world assumption, Q must not have temporal operator and undefined variables when this operator is evaluated. Tokio can be extented to handle Interval Temporal Logic. In the following section, we briefly review ITL.

1.2. Interval temporal logic

Interval temporal logic (ITL) was developed by B. Moszkowski [2]. In this logic, variable and predicate meanings are determined for each interval. An interval is a continuous finite sequence of states. In this section, five operators are introduced: chop, length, halt, fin and keep.

The chop operator is a binary operator. This operator splits an interval into two parts. '&&' symbolizes the chop operator. ';' expresses the chop operator used in [2], however, since ';' denotes the or operator in Prolog, we use '&&' to avoid confusion.

```
P && Q                                    (5)
```

The meaning of (5) is related to three intervals. Consider an interval I, diveded into two consecutive parts: Ip represents the former portion, and Iq, the later. If P is true during Ip, and Q is true during Iq, then (5) is true during I (Fig.5).

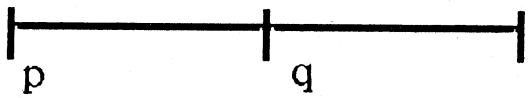

Fig. 5 p && q

The length operator specifies the length of an interval (Fig.6).

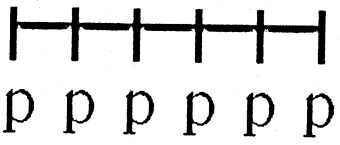

Fig. 6 length 5

The length of an interval is the number of states contained in the interval minus one.

```
p :- length(5).                           (6)
```

In (6), p is true on the interval which has 6 states. Length must be a nonnegative integer. Using chop and length, the next operator in ITL can be defined as follows (7).

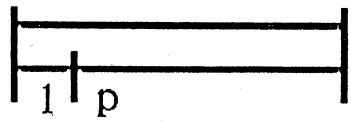

Fig. 7 @ operator in ITL

```
@P <-> length(1) && P                     (7)
```

This type of next is called strong in [5], because if the interval length is 0 then @P is false. In the other word, this operator extends an interval. An interval whose length is 0 is called empty. In the case of the LTTL next operator, it is succeeded even in an empty intervals. The next operator is independent of the interval.

```
next(P) <-> length(1) && P        (8)
or next(P) <-> empty
```

This kind of next is called weak. The conditional statement which is related to empty requires special treatment in Tokio. The following three statements are important.

```
halt(P) <-> #(P->empty, not(P)->not(empty)).
fin(P)  <-> #(empty->P).              (9)
keep(P) <-> #(not(empty)->P).
```

The halt(P) statement means that the interval length is determined by the formula P. Keep(P) means P is true for all states except the one at the end of the interval. Fin(P) means P is true only in the final state of the interval.

1.3. Tokio combines LTTL and ITL

LTTL can easily describe concurrency using conjunctions; however, sequentiality is not so easily described. ITL's chop operator is superior to LTTL's until operator for the description of sequentiality. p && q simply means that q will be executed after the termination of p. On the other hand, p until q means that p is true until q holds. This statement does not specify the termination of p.

The key concepts describing the relationship between ITL and LTTL are local and interval variables [10]. Local means ITL expressions that do not depend on when an interval ends. In another words, meanings of local variables or predicates are decided in the first time (state) of the interval. LTTL concepts are all translated into local concepts in ITL. In order to introduce the chop operator to Tokio, we use interval variables corresponding to each Tokio clause. These variables are all LTTL variables and they are generated by existential quantifiers. Tokio Variables have different meaning from ITL variables, because interval variables are only associated with clauses. This means that all the variables in Tokio are local.

```
p :- q && r.      ....   Ip
q.                ....   Iq      (10)
r.                ....   Ir
```

Ip, Iq and Ir are all interval variables. Each variable represents its corresponding interval. The Chop operator generates the intervals Iq and Ir from Ip using an existential quantifier. This corresponds to splitting an interval into two parts.

1.4. An extension of Horn clause for temporal logic

The Horn clause is a restricted form of logic formula. It consists of two parts: head and body.

```
head :- body.                     (11)
```

This means that the head is implied by the body. Variables in a head are all universally quantified and variables in a body are all existentially quantified. A head must be an atomic predicate. Roughly speaking, a Tokio Horn clause is a Horn clause including temporal operator in its body.

In Tokio, there are some restrictions. To avoid a circularly structured temporal variable, the @ function is only allowed in the expression =. The primitive temporal operators such as next or chop operator are not allowed in the head of Horn clause.

2. Resolution of Tokio

The execution of Tokio is a kind of resolution of LTTL. Unification and reduction generate a model incrementally. In LTTL, the model is generated for each state. It is also necessary to unify temporal variables.

* Reduction on multi-state model

* Unification on temporal variable

Our Tokio also has an ITL extension. Generation of interval variable is the third execution unit.

* Generation of interval variables

There are the three main execution units of Tokio.

2.1. Reduction to the future

If there are no temporal operators, the reduction of Tokio is identical to that of Prolog. A predicate including the next operator defines another state. These predicates are enqueued with the environment corresponding to the next state, and will be reduced later. Queuing is necessary to execute Tokio formula in the proper temporal order. We call this property a healthy execution. On the other hand, the execution order of a predicate referring to the same state need not be specified. In order to execute system predicates or numeric calculations, the execution order must be determined by the data dependency.

Tokio executes the predicates corresponding to a given state in the same way as Prolog does. This is only a practical solution. In this manner Tokio includes Prolog in itself. In Tokio there are two kind of reduction: Prolog-wise reduction and time-wise reduction (Fig. 8).

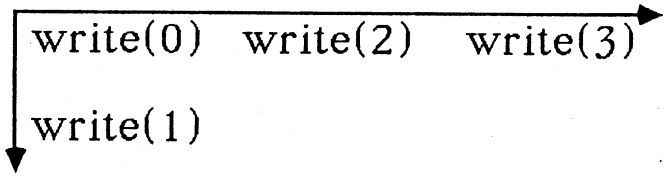

Fig. 8 Two way reduction

```
?-@ @write(3),@write(2),write(0),write(1).      (12)
0123
yes
```

'?-' means a goal in Tokio. Notice that the next statement is executed after the current state.

2.2. Unifier for temporal logic variable

A Tokio variable assumes many different values. These values are generated incrementally as time advances. Unification is a basic operation for accessing these values. Unification consists of two primitives.

1) to select current state value and successive state values.

This is very much like car and cdr.

2) to unify all the values of successive states.

The former type of unification is performed by an = and @. The later type of unification is executed in a head of clause. Consider a simple counter in Tokio.

```
counter(X) :- #(@X=X+1).                    (13)
?- X=0,counter(X),#write(X),length(3).
0123
yes
```

Counter increments the value of X in each state. The variable X in the goal is unified in all states with variable X in the counter clause. On the other hand, in the definition of counter = predicates unifies the next state of variable X and the incremented value of variable X in the current state.

2.3. Interval generator using interval variable

The interval variables in Tokio consists of two parts in our implementation. These are the ending time of the interval, and the flag of interval completion. This flag is generated in each time clock. Some ITL operators use this flag directly; for example, the fin and keep operators are not executed until this flag is determined. Tokio has a waiting queue for the fin and keep operators.

The interval variable is generated only by a chop operator. The consistency of the interval length is checked in every state. The length of an inner interval must be less or equal than the length of an outer interval.

One feature of Tokio is automatic interval length tuning. Tokio use a simple strategy for interval determination: the shortest possible interval is selected. Tokio tries to cut off the interval which does not have length specification. Controlling the interval length is performed by backtracking. This mechanism is known to useful in goal oriented simulation [15]. If there is a maximum interval length, Tokio can generate all the necessary length combination of the intervals. This is very useful when it is necessary to join processes.

```
qs(X) :- split(X,H,L) && qs(H),qs(L).       (14)
```

(14) is a part of quick sort. First qs splits a list X into two parts: H and L, and then performs qs for each part. Because length of parts may be different, two qs may use different lengths of time. In the predicate split, using a temporal operator such as temporal assignment which is not dependent on the length of interval, qs can be written in a form which is not depended on interval length. In such a case, the length of qs is automatically tuned and each qs has the same interval length.

3. Hardware description in Tokio

In this section, a sample description of a pipeline merge sorter is presented. The pipeline merge sorter is a component of the relational database machine GRACE [6]. A sorter prototype was already available. The pipelining details are described in Tokio.

3.1. Pipeline merge sorter in Tokio

Fig.10 is a sample description of a pipelineed merge sort.

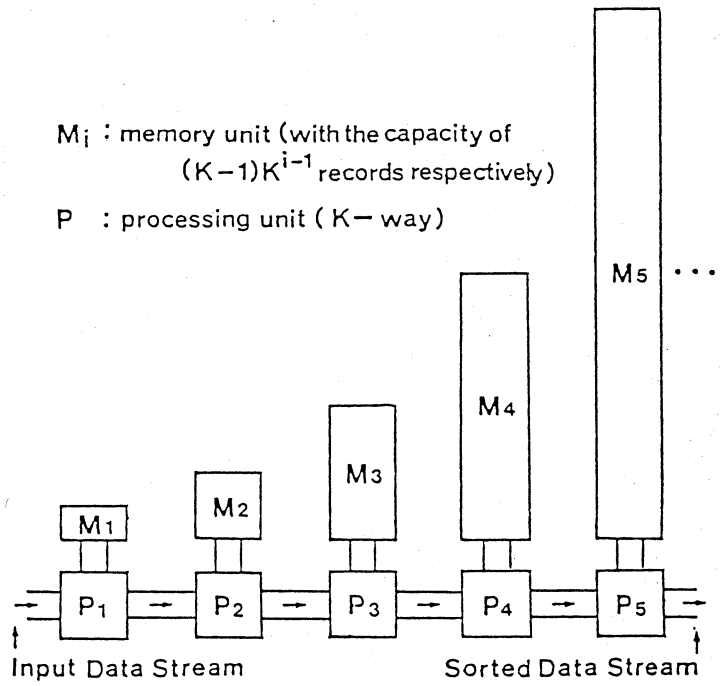

Fig. 9 Organization of Pipeline Merge Sorter

It has enough information to specify timing of the pipeline. This is a three-stage-pipeline. There are three sorters. proc describes a sorter. The first and second variables in proc are input and output of each pipeline sorter. X, Y, and Z are used as three different input buffers. T represents count of input data. P and PP are constants of each processor. It has three major parts. At first, if there is no data then proc is in a waiting state. In an active state, there are two parts: load and merge. Load and merge work concurrently in this program.

Load also has two phases. In the first phase, input data is stored into the input buffer: Z. In the second phase, input data is stored into the merge buffer: Y. These two phases are selected by the count of input data. The size of input data is always the power of 2, and fixed to each processor. The variable T is used to count up the input data.

In the merge process, the third buffer: X and the merge buffer: Y are merged into output. If either buffer is empty, according to the state of loading phase, input buffer: Z is changed to X.

Whole processes are combined using conjunctions, and work concurrently. In the program test, there are 4 processes: a data generator and a three stage pipeline merge sorter. The execution sample is shown in Fig.11. Cpu time is measured on VAX11/730.

```
| ?- tokio test.
[][][][][][][][][][][1][2][3][5][6][10][20][100][]
18 clock and 18.0834 sec.
```

Fig. 11 Sample Execution

4. Compilation to Prolog

The compilation to Prolog is the easiest and fastest way to imple-

```
test :- Strdata = [10,20,5,100,1,6,2,3],
    datagen(Strdata, Data),
    pipe(Data, Out),
    length(18),
    #write(Out).

        % Data Generator

datagen([],[]).
datagen([H|T],Out) :-
    Out = [H],
    @T = T, @datagen(T,Out).

        % Pipeline Merge Sorter

pipe(I0,Out) :-
    I1 = [], I2 = [], Out = [],
    proc_start(I0,I1, 2,1),
    proc_start(I1,I2, 4,2),
    proc_start(I2,Out,8,4).

    % Processor Unit

proc_start(I,O,P,PP) :-
    X = [], Y = [], Z = [], T = 1,
    #proc(I,O,X,Y,Z,T,P,PP).

proc(I,O,X,Y,Z,T,P,PP) :- X=[],Y=[],I=[],!,
    @X=X, @Y=Y, @Z=Z, @O=[], @T=1.
proc(I,O,X,Y,Z,T,P,PP) :-
     load(I,O,X,Y,Yn,Z,Zn,T,P,PP),
    merge(I,O,X,Y,Yn,Z,Zn,T,P,PP).

load(I,O,X,Y,Yn,Z,Zn,T,P,PP) :- T=<PP, !,
    append(Z,I,Zn), @Z=Zn, Yn=Y,
    @T=T+1.
load(I,O,X,Y,Yn,Z,Zn,T,P,PP) :-
    append(Y,I,Yn), @Z=[],
    if T<P then @T=T+1 else @T=1.

merge(I,O,X,Y,Yn,Z,Zn,T,P,PP) :-X=[],Yn=[],!,
    @O=[], @Y=Yn,
    if T=PP then @X=Zn else @X=X.
merge(I,O,X,Y,Yn,Z,Zn,T,P,PP) :- X=[A|L],Yn=[],!,
    @O=[A], @Y=Yn,
    if T=PP then @X=Zn else @X=L.
merge(I,O,X,Y,Yn,Z,Zn,T,P,PP) :-X=[],Yn=[B|N],!,
    @O=[B], @Y=N,
    if T=PP then @X=Zn else @X=X.
merge(I,O,X,Y,Yn,Z,Zn,T,P,PP) :-X=[A|L],Yn=[B|N],!,
    if A<B then
        @O=[A], @X=L, @Y=Yn
    else
        @O=[B], @Y=N, @X=X.

append(Nil,L,L1) :- Nil=[],L=L1.
append(X,L,Y) :-[H|T]=X,[H1|M]=Y,
     H=H1,append(T,L,M).
```

Fig. 10 Pipeline Merge Sort

assert or retract, they can be compiled efficiently by a Prolog compiler. The basic compilation method of Tokio is the same as that of GHC compiler to Prolog [7]. These use the queue structure for scheduling and in-line development of unifications. Our compiler also has a macro facility for the development of second order predicates such as meta-call. The main features of Tokio are listed below.

1) Multi-time queue structure for reduction

2) In-line development of unification and generating local unifier

3) Macro facility, which enables control abstractions

In the following section, details relating to these points are discussed.

4.1. Multi-time queue

Three main queues are used in Tokio. In this compiler the queue is generated incrementally, corresponding to one state. All these queues are D-list and make a list structure in state order.

```
[$t(N, F, K, C),          ...   1st state
 $t(N1,F1,K1,C1),         ...   2nd state
 $t(N2,F2,K2,C2)|         ...   3rd state
 Q]                       ...   rest of states
```

Fig. 12 Tokio queue structure

The queue N is for next queue, F is for fin queue, K is for keep queue. The fourth element: C keeps the interval variables and current time. Interval variables indicating ending time of the interval and end flag correspond to each clause in this way. Fig.13 shows simple compile examples.

```
% cprolog
C-Prolog version 1.5
Tokio consulted 57168 bytes 66.6166 sec.
| ?- com([tm,user],user).                % start compile
|: p :- q, r.                            % input clause
p(_0,_1):-q(_0,_2),r(_2,_1).            % compile output
|: p :- next(q), r.
p([$t((q(_0,_1),_2),_3,_4,_5)|_0],_6):-
   r([$t(_2,_3,_4,_5)|_1],_6).
|: p :- next(next(next(r))).
p(
    [_0,_1,$t((r(_2,_3),_4),_5,_6,_7)|_2],
    [_0,_1,$t(_4,_5,_6,_7)|_3]).
|: p(X,Y) :- X=Y.
p(_0,_1,_2,_2):-
    unifyNow(_0,_3),unifyNow(_1,_3).
|: eq(X,X).
eq(_0,_1,_2,_2):-unifyAll(_0,_1).
|: p(X,Y) :- @X=Y.
p(_0,_1,_2,_2):-
    unifyNext(_0,_3),unifyNow(_3,_4),
    unifyNow(_1,_4).
```

Fig. 13 Sample Compile

In our interpreter [9], one state queue structures is used. The

1) Nested next operator is compiled into single enqueue.

2) Compilation algorithm becomes more symmetric to current state and next state.

4.2. Unifier generation

In order to express multi-state variables, the stream representation is used. These states are created incrementally, and there must be an incomplete part at its tail, which contains the rest of states. Tokio uses node: $t for this list (Fig. 14).

```
| ?- Tokio length(5),X=1,
counter(X),#write(X).
123456
X = $t(1,$t(2,$t(3,
    $t(4,$t(5,$t(6,_))))))
```

Fig. 14 Temporal Variable Representation

There are two types of constant in Tokio. The first one is a constant in a single state. The second one is a constant in all states. The object of Tokio is a mixture of these two constants. The unification of Tokio must be applied to these structures.

There are two kinds of unification in Tokio.

1) Unification on head of clause: Unification is performed for all successive states.

2) Unification in the = predicate: Unification is performed for the current state.

The Tokio variable is a list of logical variables. The second type of unification selects the value of a variable in the current state. This unification is very much like car in a list structure. The part corresponding to the cdr selector is also available in Tokio. This is done by next operator.

3) Unification in the next predicate: Unification is performed for next successive states.

In the Tokio compiler these primitives are compiled into the following predicates.

```
unifyAll(X,Y)  ...  unifies all state values of X and Y
unifyNow(X,Y)  ...  Y is a current state value of X
unifyNext(X,Y) ...  Y contains all successive state of X
```

Temporal variable structure is analyzed using these predicates. The unifyAll statement includes a double loop: a loop for structures and a loop for time structure. If both unifyNow and unifyNext are necessary in a single clause, the combined clause unifyNowNxt is used.

Because of the double loop of unifyAll, structure in a head slows down operation of Tokio considerably. Like a GHC to Prolog compiler, in-line development of head unification is effective. In GHC, only a list structure is expanded, because the most important structure in GHC is a stream which is a list. In Tokio, structure in the head is used as a type of variable, and list has no special meaning. So Tokio expands all head unifications in-line.

4.3. Control abstraction in temporal logic

In Prolog, there is no while-do statement. Prolog cannot abstract control structures containing loops, because it is difficult to express how to create new variables in recursion. On the other hand, Tokio can express while-loops very easily.

```
while P do Q :- P,!,(Q && while P do Q).          (15)
while P do Q :- empty.
```

4.4. Compiler Statistics

Fig.15 details the statistics of this compiler.

	clause size (byte size)		execution time (sec) on Vax11/730 (0.2 Mips)		
program	source	object	Interpreter	Compiler	Prolog
append (30list) (all state)	3 (49)	3 (177)	10.6	0.8	0.13
append (single state)	3 (92)	3 (224)	27.9	2.0	0.13
pipeline merge sorter	15 (1380)	27 (5726)	154.4	18.0	----

Fig. 15 Tokio compiler

C-Prolog [13] is used as a basic Prolog. This Tokio compiler is 5 times faster than the Tokio interpreter in Prolog, and 7 times slower than original Prolog if Tokio is used as a Prolog.

5. Comparison with related languages

In this chapter, Tokio is compared with Tempura, GHC and T-Prolog.

5.1. Comparison with Tempura

Tokio is a logic programming version of Tempura. Useful ITL operators come from Tempura. According to logic programming concepts, there are several features in Tokio.

1) Backtracking to the past: Tokio has a backtracking mechanism as just like that of Prolog. These backtracking mechanisms enable automatic interval length tuning. Actually Tokio is a kind of branching time temporal logic.

2) Unification over the temporal logic variables: Tokio is actually a pseudo concurrent programming language. The execution order in one state is uncertain. The bidirectional assignment of unification makes the concurrent programming easy.

3) The program is the extension of a Horn clause.

Tempura does not have backtracking and unification. Actually Tokio can execute a somewhat wider range of temporal logic formula than Tempura. On the other hand, the execution of Tempura is faster, and consumes less memory than that of Tokio. Tempura use some lambda expression for its representation. It needs temporal logic type expression such as stbtype (stable type). In Tokio, such a type is expressed as a structure in head.

Tempura has two kinds of variables: stable and non-stable. The corresponding element in Tokio is a static variable which is implemented with a record statement, that is, assert. These variables are needed for the description of circuit containing many registers. These variables are not good elements as far as formal verification is concerned. Notice that in the description of the pipeline merge sorter, only list structures are used.

Finally, Tokio is based on LTTL. Verification of the synchronization part of Tokio will be possible using the LTTL verification method. ITL operators in Tokio is translated into LTTL operators using interval variables.

5.2. Comparison with GHC

Guarded Horn Clause [7] is a stream parallel logic programming language. Tokio is used for specification, while GHC is used for actual representation of parallel programming.

The main difference between GHC and Tokio is a guard. Guard is a selection point of disjunction. In Tokio, selection is performed by backtracking as in Prolog. On the other hand, GHC's guard selects the clause whose execution of guard parts is completed fastest. It cuts off other selections, so GHC has no backtracking. Backtracking is a powerful verification tool.

Another difference is the concept of time. Tokio has a minimum unit of time. GHC has no such minimum unit. In a clocked circuit or in a systolic programming, Tokio is superior to GHC. For example, consider a simple producer-consumer.

Tokio

```
?-X=1, #produce(X), #consume(X).
produce(X) :- @X=X+1.                          (17)
consume(X) :- write(X).
```

GHC

```
?-produce(1,X), consume(X).
produce(X,T)    :- X1 := X+1 |
           T=[X|T1], produce(X1,T1).           (18)
consume([X|T]) :- write(X)  |
           consume(T).
```

In GHC, synchronize of two processes is difficult; for example, synchronization requires structure in head for the suspension of 'consume' and guard. On the other hand, Tokio has built-in synchronized clock. It is very easy to write synchronized programs. The previous pipeline merge sorter is one example.

5.3. Comparison with T-Prolog

T-Prolog [15] is a Simula like simulation system on Prolog. All the process in T-Prolog is represented by a Prolog-like description which has predicates relating to time concept such as during, or after. Using these operators T-Prolog can express useful temporal relations ships among parallel process. In the following example [15], Dick first waits decision of which safe should be open, then sends tools for his friends.

T-Prolog

```
DICK_GETS_THE_MONEY(*BANK,*SAFE):
     WAIT(IDENTIFIER(*SAFE)),          (19)
```

Tokio

```
dick_gets_the_money(Bank, Safe, Tools):-
    keep(Safe=undef) &&                    (20)
    has(Safe, Tools).
```

A process is executed in serial way. However T-Prolog uses complicated primitives to describe communications, for example, SEND and WAIT. Assignment of a variable in T-Prolog can be used as a synchronization tool in the same way of GHC. In Tokio, all the communications are represented by a formula including temporal variables. T-Prolog cannot express communications by its variables, because its variables have no state. Only a truth value of predicate has states in T-Prolog. On the other hand, Tokio variables have states. It is suitable for the hardware description, because the state of connection lines are changed in each time. The synchronization mechanism is separated from variables in Tokio. The basic mechanism of synchronization is a chop operator.

T-Prolog has a backtracking in time. This is a good tool for the goal oriented simulation. The backtracking mechanism of Tokio can work like T-Prolog. However our backtracking is based on each clock period, not on each event.

The current implementation of Tokio is not suitable for actual parallel execution. In a real situation, the problem is healthy execution, i.e. relationship between actual time and Tokio time. Once Tokio is running under some parallel machine, it is difficult to execute Tokio in a healthy way, i.e. execution in the correct time order. To execute Tokio program in a healthy way, it is better to translate it to other hardware or parallel programming language, because the description of Tokio is an abstract one and it includes the difficult task of temporal variable unification.

6. Conclusion

Tokio is temporal logic language suitable for specification of concurrent algorithms and simulation. The compilation to Prolog is an efficient implementation of Tokio. We have written Tokio interpreter and compiler in C-Prolog [13]. There are many hardware description examples which include the unify-processor of a Parallel inference engine: PIE [14] and systolic array matrix multiplication. We plan to develop a Tokio verifier. This is based on an LTTL verifier using Prolog [11].

Many thanks to those people. Mr. Okano wrote a pipeline merge sorter. Vince help us with his English. Dr. Tsang give us many useful discussion.

References

[1] Z. Manna and A. Pnueli, "Verification of Concurrent Programs, Part 1: The Temporal Framework", Dept. of Computer Science, Stanford Univ. Report STAN-CS-81-836, June 1981.

[2] B.C. Moszkowski, "Reasoning about Digital Circuit", Rep. No.STAN-CS-83-970 Dept. of Computer Science, Stanford Univ. July 1983.

[3] M. Fujita, H. Tanaka and T. Moto-oka, "Logic Design Assistance with Temporal Logic", IFIP 7th Computer Hardware Description Languages and their Applications, August 1985.

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

[5] B.C. Moszkowski, "Executing Temporal Logic Programs", Rep. No.55 Computer Laboratory, Univ. of Cambridge, 1984.

[6] M. Kituregawa, H. Tanaka and T. Moto-oka, "Relational Algebra Machine GRACE", Lecture Notes in computer Science 147, Springer-Verlag, March, 1983.

[7] K. Ueda, "Guarded Horn Clauses", TR-103, ICOT, 1985.

[8] W.F Clockskin and C.S Melish, "Programming in Prolog", Springer-Verlag, New York, 1981.

[9] S. Kono T. Aoyagi, M. Fujita, H. Tanaka, "Implementation of temporal logic programming language Tokio", to be appeared as "Proceedings of LPC'85", Lecture Notes in Computer Science Springer-Verlag.

[10] M. Fujita, S. Kono, H. Tanaka and T. Moto-oka, "Assistance in Hierarchical and Structured Logic Design Using Temporal Logic and Prolog", to be appeared in IEE proceedings-E COMPUTER AND DIGITAL TECHNIQUES.

[11] M. Fujita, H. Tanaka and T. Moto-oka, "Specifying Hardware in Temporal Logic & Efficient Synthesis of State-Diagrams Using Prolog", Proc. of FGCS '84, Tokyo Japan, November 1984.

[12] M. Fujita, "Logic Design Assistance with Temporal Logic", Doctoral Dissertation, Information Engineering, University of Tokyo, 1984.

[13] F. Pereira, "C-Prolog Users Manual Version 1.5",EdCAD, Edingburh Univ. 1984.

[14] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata and T. Maruyama, "The Architecture of a Parallel Inference Engine -PIE-", Proc. of FGCS '84, Tokyo, Japan, November 1984.

[15] I. Futo, J. Szeredi, "T-PROLOG A VERY HIGH LEVEL SIMULATION SYSTEM GENERAL INFORMATION MANUAL", 1011 Budapest I. Iskcla utca, April 1981.