IMPLEMENTATION OF TEMPORAL LOGIC PROGRAMMING LANGUAGE Tokio

S. Kono, T. Aoyagi, M. Fujita*, H.Tanaka

Department of Electronic Engineering, University of Tokyo
* Currently assigned to Fujitsu laboratories

ABSTRACT

The temporal logic programming language, "Tokio" can be executed by a resolution of Interval Temporal Logic. The resolution consists of three parts, which are: the unification of the temporal variable, reduction including temporal operator, and interval control. The implementation of Tokio includes automatic interval length determination and stream-like temporal variable representation. At the end of this report, an abbreviated version of a Tokio interpreter written in Prolog will be shown.

1. TEMPORAL LOGIC PROGRAMMING LANGUAGE Tokio

Tokio is a concurrent logic programming language designed for hardware description, based on first order linear time temporal logic (LTTL) (Wolper 1981) and first order local interval time temporal logic (ITL) (Moszkowski 1983). Since ITL fully embraces first-order predicate logic, Tokio includes the capabilities of Prolog.

When compared to Prolog's unification and reduction processing, Tokio processing consists of the following three elements:

-- Unification of temporal logic variables that possess different values at different times.

-- Ordinary reduction and future reduction

-- Division of time intervals

The following chapters summarizes ITL used as an extension of LTTL, and discuss the methods of implementing the above three elements.

2. LOGIC OF Tokio

Tokio executes Local ITL on the basis of LTTL. This chapter describes ITL and Tokio logic. In non local ITL, the value of a variable can be determined for time intervals. On the other hand in the Local ITL the value of a variable can be determined for time axis.

Local indicates that the value of variables is only determined at the beginning of an interval, and does not depend on the final time of the interval. In this sense, LTTL and local ITL are equivalent. But ordering descriptions is easier in ITL than in LTTL because times are treated as intervals. The use of LTTL is also convenient, if it includes an automatic synthesis system and a verification system for logical circuits (Fujita 1983). The language to be proposed, therefore, must permit ITL descriptions by expanding LTTL's capabilities.

In Tokio, the LTTL operator @ (next) is a basic temporal operator. @p means that p is true at the next time, that is LTTL has a discrete time concept, as does Tokio. In ITL's view, @p creates a new interval and p is true in this interval. The other important operator is "&&" chop of ITL. p && q means that p is true in some interval and q is true in the succeeding interval.

To execute chop by a next operator, Tokio uses two variables. One is for the fin time of time interval, and the other for the indicator of interval terminating. Tokio propositions are generally determined for time axis, and variable values are associated with the interval. For an atomic predicate, except for the few temporal operators, the truth value of the predicate is dependent only on the time. Previous "@" operator sets the later variable to "not empty". That is, there must be a next time. The "next" operator of this type is called a "strong next". There is a "weak next" (wnext), which does not set the variable of interval terminating. In the weak next, if there is no next time that is the end of the interval, the whole formulae is true.

The Tokio program is a kind of Horn clause. The primitive temporal operators such as next or chop are not allowed in head of the Horn clause. This is a useful subset of first order theory. Using these Horn clauses, the other LTTL operators are defined easily. For example, using weak next the operator "#", "always" is defined as follows. This operator corresponds to the square of LTTL.

```
#P :- P,wnext(#P).
```

The syntax of the Horn clause is that of C-Prolog (Pereira 1984) In the following sections we discuss the unification and reduction of Horn clauses of ITL.

3. UNIFICATIONS FOR TEMPORAL VARIABLES

For a first-order predicate, the truth value is always determined from the meaning of its argument for all times. Accordingly, unification in Tokio is executed at all times. Operator "=" is only used to fetch the current value of a variable. The "@" (next) function is used to fetch a future variable value*. To avoid a circularly structured

* This next function should be distinguished from the next operator, even though we use the same symbol. The next operator is prefixed to a predicate, while the next function is prefixed to a variable.

temporal variable, the "@" function is only allowed in the expression "=". In the leading discussion, there are two kinds of unification in Tokio. One is unification in the head of a Horn clause, which we call a full time unification. The other is a unification in a "=" predicate, and we call this a one time unification.

In principle, a temporal logical variable has values for all time (may be infinite). Specific finite values are only referenced, however. In Tokio, the value of a variable is generated incrementally according to time advance. When the value of some time is generated, that variables is called differentiate. In a full time unification, the unification for non differentiated variable is a simple assignment for it. The differentiated variable is represented by the following list.

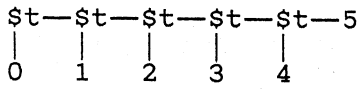

Fig 3.1 temporal variable structure

The leaves of 0-4 means the value at times 0-4. The leaf of 5 represents non differentiated parts. $t is a tag or functor for differentiated variables.

examples

```
eq(X,X).
?-@A = 1,@ @B = 2, @ @ @(eq(A,B)).
```

This results in the temporal logic variable structures shown below. Underscored number represents uninstanciated variable.

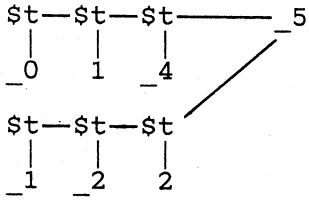

Fig 3.2 temporal variable structure

In the Prolog interpreter of Tokio, a variable is expressed by the $t(Now,Next) structures by using Prolog functors. unfiyAll specifies unification at all times. unifyNow is the predicate used for obtaining the current portion of the variable. Similarly, unifyNext obtains the future portion. Both unifyNow and unifyNext generate a new structure if the specified structure is unified with a differentiated variable. Specifically, repetition by functor and arg generates copies of the structure. During the actual, more detailed implementation stage, such copy generation is achieved by structure sharing. This method, however, does not improve storage efficiency, because of its multiple environments. Unification at the current time only, represented by "=", is enabled by unifyNow.

Note how unification is done for structures that are static in terms of time, and variables vary according to time. By using this method, the execution speed is maximized when variables are not differentiated. The other characteristic of this method is that Tokio can be implemented in a form similar to Prolog. But treatment of the constants on an interval is not so efficient in this implementation. This method is effective because structures do not usually exist at the clause head in actual Tokio programming in our experience.

4. FUTURE REDUCTION

In Tokio, reduction is done in two directions: future and current axis. Reduction in the current direction is identical to reduction in Prolog. Reduction in the future direction occurs when a new interval is generated by "@" (next) or "&&" (chop). Tokio executes these two types of reductions in the order of time values. In other words, a reduction generated by "next" is performed after all current reductions are completed. This necessitates the preparation of a queue, called "next quee", which consists of the literals to be reduced. In the simplest implementation, the "next queue" is organized as a FIFO queue.

The next queue contains literals with assigned variable for future times, that have been created by unifyNext. The next queue will be closed with backtracking in the current axis direction. A backtrack may be performed beyond the current time, which is called "backtrack to the past". "Cut" may be considered when backtracking in Tokio. For a backtrack in the current axis direction, the cutting feature can be supplied in the same way as in Prolog. Implication and negation without temporal operators are executed by cutting, although another mechanism is required for negation related to time intervals. The five primary negation operations related to intervals are listed below.

```
empty       :- wnext(false).
notEmpty    :- not(empty).
halt(P)     :- #(if P then empty else notEmpty).
fin(P)      :- #(if empty then p).
keep(P)     :- #(if notEmpty then p).
```

These operations are all related to interval determination. The definitions of empty and notEmpty using the weak next definition are somewhat technical, and are not suited for implementation. The end of an interval is directly used to determine whether to execute "fin" or "keep". In other words, the execution of "fin" or "keep" cannot start until completion of the interval has been verified. Therefore, both a fin queue and a keep queue are required.

The end of an interval is signified by the execution of either an "empty" or a "notEmpty". Execution of theses predicates is done by setting a flag that indicates interval completion. The formula fails, of course, if a conflict is caused by the flag. The flag must be reset by a backtrack. If neither "empty" nor "notEmpty" is executed, "empty" flag is set currently. In this case, the interval completes in this time. If backtrack is reached at this point, the flag is reset and the length of the interval is extended. In this way, the length of an

interval is adjusted to gradually increase, starting from 1.

In the Tokio interpreter, t reduce generates i reduce for each time. The time precedes until only true remains or exceeds the end of the interval. At i reduce, reduce operation is done in the same way as in Prolog, which reduces the literals with the exception of the temporal operators. Then, force finite is used to cut unlimited intervals. Depending on the result of the cutting operation, either "fin" or "keep" is executed by exec fin keep. The queue structure represented by Q is always retained by reduce. Q contains queues for "fin", "keep", and "next", and a flag indicating "empty". It also retains the current time and "fin" time.

5. DIVIDING INTERVALS

The chop operator is used to divide an interval. Specifically, it divides interval p&&q into two shorter intervals during which p and q will be executed respectively. These shorter intervals are executed independently of each other by i reduce. "fin" and "keep" are executed during each interval. Each interval is identified by its interval variables, i.e., the flag indicating the end of interval and fin_time indicating the end time. This means that each interval can be expressed using the following structures:

```
$i(p,q,fin_time)
```

where p is executed during that interval and q will be executed during the following interval. q is executed at "fin" of p, which differs from fin(q).

6. THE INTERPRETER

The Tokio interpreter written in Prolog shown here can execute basic Tokio programs. The execution speed is approximately 40 times slower than that of the original Prolog. The compiler of Tokio to Prolog has also been written, which operates 5 times faster than the interpreter. At present, an interpreter and a compiler written in the C language, are being developed. The interpreter will be the almost as fast as C-Prolog, and the compiler will be 5-10 times faster.

7. DIFFERENCE FROM TEMPURA

Tempura is a Lisp-like language based on ITL, and is written in Lisp. When compared to Tokio, the method of executing Tempura is simpler. However Tokio can handle a wider range of ITL executions than Tempura. The following restrictions apply to Tempura:

- Tempura programs are written in Lambda format, including temporal operators. It has no unfications.
- Tempura does not support the backtrack feature.
- Two or more consecutive "next" operations, such as @ @ A, are not allowed.

- Cross-references, such as exchange of two variables' values, are not allowed.

Tempura, however, is provided with more intrinsic functions than Tokio. Tempura has better storage efficiency than Tokio written in Prolog, mainly because it does not perform backtracking.

8. CONCLUSION

The Tokio language has been designed to execute ITL on the basis of LTTL. It supports both mathematical fundamentals and actual simulations for hardware description. Several students in our university are using Tokio. Programs written in Tokio include the Unify Processor which is a prototype of PIE, Parallel Inference Engine, and systolic array for matrix multiplication and pipeline merge sorter. We are now planning to develop a system based on Tokio to verify and synthesis logical circuits that include functional relationship.

REFERENCES

[1] M. Fujita, "Temporal Logic Based Hardware Description and Its Verification with Prolog", New Generation Computing, Vol. 1, No. 2, pp. 195-203 1983.

[2] B. Moszkowski, "Reasoning about Digital Circuits", Report No. STAN-CS-83-970, Department of Computer Science, Stanford University, July 1983.

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

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

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

[6] D. Warren, "AN ABSTRACT RPOLOG INSTRUCTION SET", Technical Note 309, SRI International, October 1983.

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

```
/* Tokio interpreter Reduced Version */

t(A):-t_reduce(A,0,Fin).      % entry

/* temporal reducer
       t_reduce(Formula,Now,Fin) */

t_reduce(true,Now,Fin):-integer(Fin),Now>Fin,!.
t_reduce(Formula,Now,Fin):-
       nl,write((t)),write(Now),write(':'),
```

```
        i_reduce(Formula,Next,Now,Fin,_),
        NextT is Now+1,
        t_reduce(Next,NextT,Fin).

/* interval reducer
        i_reduce(Formula,Next,Now,Fin,OuterFin)
        reduce now formula and
        execute either
        keepQueue or finQueue */

i_reduce(Formula,Next1,Now,Fin,OuterFin):-
        qcl(Next,EmptyFlag,Q1,Q2),
        reduce(Formula,Q1,Now,Fin),
        force_finite(EmptyFlag,Now,Fin,OuterFin),
        exec_fin_keep(Next,Next1,Q1,Q2,Now,Fin).

/* interval terminate works */

exec_fin_keep(Next,true,Q1,Q2,Now,Fin):-
        Now==Fin,!, get_fin(F,Q1),
        reduce(F,Q2,Fin,Fin).
exec_fin_keep(Next,Next,Q1,Q2,Now,Fin):-
        get_keep(K,Q1),
        reduce(K,Q2,Now,Fin).

/* cut off infinite interval
        force_finite(EmptyFlag,Now,Fin) */

force_finite(free,Now,Fin,OuterFin):-
        var(Fin),Fin is Now+1,
        (nonvar(OuterFin),OuterFin=Fin,! ; true).
force_finite(_,Now,Fin,OuterFin).

/* reducer
        reduce(F,Q,Now,Fin) */

reduce((A,B),Q,Now,Fin):-qap(Qa,Qb,Q),!,
        reduce(A,Qa,Now,Fin),
        reduce(B,Qb,Now,Fin).
reduce(A,Q,Now,Fin):-systemp(A),!,
        exec(A,Q,Now,Fin). % system predicate
reduce(A,Q,Now,Fin):-
        t_clause(A,B),reduce(B,Q,Now,Fin).

t_clause(A,B):-
        functor(A,Head,N),functor(AA,Head,N),
        clause(AA,B),
        unify_all(A,AA).

/* system predicate exec(Formula,Q,Now,Fin) */

exec(true,Q,_,_):- !,nonq(Q).
exec(length(N),Q,Now,Fin):- !,
        nonq(Q),Fin is Now+N.
```

```
exec(empty,Q,Now,Fin) :- nonq(Q),!,Now=Fin.
exec(notEmpty,Q,Now,Fin) :- !,
        enq_nonEmpty(true,Q).
exec(halt(F),Q,Now,Fin):-reduce(F,Q,Now,Fin),
        !,Now=Fin.
exec(halt(F),Q,Now,Fin):- unifyNext(F,Fn),
        enq_nonEmpty(halt(Fn),Q).
exec(#F,Q,Now,Fin):-qap(Q1,Q2,Q),enq_nxt(#Fn,Q1),
        unifyNext(F,Fn),!,
        reduce(F,Q2,Now,Fin).
exec(@F,Q,Now,Fin):-!,              % strong next
        enq_nonEmpty(Fn,Q),
        unifyNext(F,Fn).
exec(wnext(F),Q,Now,Fin):-          % weak next
        enq_nxt(Fn,Q),
        unifyNext(F,Fn),!.
exec(fin(F),Q,Now,Fin):-!,(
        (Now==Fin,!,reduce(F,Q,Now,Fin)) ;
        qap(Q1,Q2,Q),enq_fin(F,Q1),
        unifyNext(F,Fnn),enq_nxt(fin(Fnn),Q2)).
exec(keep(F),Q,Now,Fin):-!,
        qap(Q1,Q2,Q),enq_nfin(F,Q1),
        unifyNext(F,Fnn),enq_nxt(keep(Fnn),Q2).
exec((A && B),Q,Now,Fin):-!,
        sub_exec(A,B,Q,Now,Fin,SubFin).
exec($int(A,B,SubFin),Q,Now,Fin):-!,
        sub_exec(A,B,Q,Now,Fin,SubFin).
exec(A=B,Q,Now,Fin):- qap(Q1,Q2,Q),!,
        eval(A,Va,Q1,Now,Fin),
        eval(B,Vb,Q2,Now,Fin),
        unifyNow(Va,V),
        unifyNow(Vb,V).
exec(A<--B,Q,Now,Fin):-!,eval(B,Vb,Q,Now,Fin),
        unifyNow(Vb,A1),unify_all(A,A1).
exec(S,Q,Now,Fin):-unifyNow(S,SN),!,
        SN,nonq(Q).

/*      subinterval execute
        sub_exec(Former,Later,Q,Now,Fin,SubFin) */

sub_exec(F,L,Q,Now,Fin,SubFin):-
        i_reduce(F,Fnext,Now,SubFin,Fin),
        sub_exec_later(L,Fnext,Q,Now,Fin,SubFin).

sub_exec_later(L,Next,Q,Now,Fin,SubFin):-
        Now==SubFin,!,
        reduce(L,Q,Now,Fin).
sub_exec_later(L,Next,Q,Now,Fin,SubFin):-
        unifyNext(L,Ln),
        enq_finNext(L,$int(Next,Ln,SubFin),Q).


/*      function evaluate
        eval(Formula,Value,Queue,Now,Fin) */
```

```
eval(Atom,Atom,Q,Now,Fin):- (atomic(Atom) ; var(Atom)
        ; functor(Atom,$t,2)),
        nonq(Q),!.
eval(@Var,Next,Q,Now,Fin):-!,
        eval(Var,NowValue,Q,Now,Fin),
        unifyNext(NowValue,Next).
eval(A+B,V,Q,Now,Fin):-!, qap(Q1,Q2,Q),
        eval(A,Va,Q1,Now,Fin),
        eval(B,Vb,Q2,Now,Fin),
        unifyNow(Va,Vna),unifyNow(Vb,Vnb),
        V is Vna+Vnb.
eval(S,S,Q,_,_) :- nonq(Q).

/* queue structures
        1       2       3       4
q(Next-Next,Fin-Fin,Keep-Keep,EmptyFlag)
*/

qcl(Next,EmptyFlag,
    q(Next-Next1,Fin0-true,Keep0-true,EmptyFlag),
    q(Next1-true,Fin1-true,Keep1-true,EmptyFlag)).

qap(q(Next2-Next1,Fin2-Fin1,Keep2-Keep1,EmptyFlag),
    q(Next1-Next ,Fin1-Fin ,Keep1-Keep ,EmptyFlag),
    q(Next2-Next ,Fin2-Fin ,Keep2-Keep ,EmptyFlag)).

nonq(q(Next-Next,Fin-Fin,Keep-Keep,EmptyFlag)).

enq_nxt(N,                          % weak next
    q((N,Next)-Next,Fin-Fin,Keep-Keep,EmptyFlag)).
enq_nonEmpty(N,                     % strong next
    q((N,Next)-Next,Fin-Fin,Keep-Keep,nonEmpty)).
enq_fin(F,
    q(Next-Next,(F,Fin)-Fin,Keep-Keep,EmptyFlag)).
enq_nfin(K,
    q(Next-Next,Fin-Fin,(K,Keep)-Keep,EmptyFlag)).
enq_finNext(F,N,
    q((N,Next)-Next,(F,Fin)-Fin,Keep-Keep,nonEmpty)).

get_fin(F,Q):-arg(2,Q,F-_).
get_keep(K,Q):-arg(3,Q,K-_).
get_nonEmpty(EmptyFlag,Q):-arg(4,Q,EmptyFlag).

nonEmpty(Q,Now,Fin):-Now==Fin,!, fail.
nonEmpty(Q,Now,Fin):-get_nonEmpty(nonEmpty,Q).

/* unifiers */

unify_all(G,D):-
        (var(G) ; var(D)),!,G=D.
unify_all(Fl,D):-
        functor(Fl,$t,2),!,
        unify_flt(Fl,D).
unify_all(D,Fl):-functor(Fl,$t,2),!,
```

```
        unify_flt(Fl,D).
unify_all(Sa,Sb):-
        functor(Sa,H,N),functor(Sb,H,N),
        unify_arg(N,N,Sa,Sb).

unify_arg(0,N,_,_):-!.
unify_arg(M,N,Sa,Sb):-
        arg(M,Sa,Aa),arg(M,Sb,Ab),
        unify_all(Aa,Ab),M1 is M-1,!,
        unify_arg(M1,N,Sa,Sb).

unify_flt($t(Now,Nxt),$t(Now,Nxt1)) :-!,
        unify_all(Nxt,Nxt1).
unify_flt($t(Now,Nxt),S) :-
        unifyNow(S,Now),
        unifyNext(S,Nxt1),
        unify_all(Nxt,Nxt1).

unifyNow(X,X1):-atomic(X),!,X=X1.
unifyNow($t(Now,_),Now1):-!,Now=Now1.
unifyNow(S,Sn):-
        functor(S,H,N),functor(Sn,H,N),
        unifyNowArg(N,N,S,Sn).

unifyNowArg(0,_,_,_).
unifyNowArg(M,N,Sa,Sb):-
        arg(M,Sa,Aa),arg(M,Sb,Ab),
        unifyNow(Aa,Ab),M1 is M-1,!,
        unifyNowArg(M1,N,Sa,Sb).

unifyNext(X,X):-atomic(X),!.
unifyNext($t(_,Next),Next1):-!,Next=Next1.
unifyNext(S,Sn):-
        functor(S,H,N),functor(Sn,H,N),
        unifyNextArg(N,N,S,Sn).

unifyNextArg(0,_,_,_).
unifyNextArg(M,N,Sa,Sb):-
        arg(M,Sa,Aa),arg(M,Sb,Ab),
        unifyNext(Aa,Ab),M1 is M-1,!,
        unifyNextArg(M1,N,Sa,Sb).

/* predefined utilities */

<>(F)           :- true && F.
A gets B        :- keep(B = @A).
% stable(A)     :- A gets A.
stable(A)       :- keep(A = @A).
% B <- A        :- stable(C),A=C,fin(B=C).
B <- A          :- C<--A,fin(B=C).
skip            :- length(1).
```