Application of Temporal Logic to the Assistance of Hardware Logic Design

Masahiro FUJITA

FUJITSU LABORATORIES LTD.

1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Abstract

Temporal logic is an extension of classical logic to express temporal sequences. This paper presents application techniques of temporal logic to hardware logic design assistance: specification of digital systems in temporal logic, a verification method between specification and gate-level designs, and a synthesis method of state diagrams from specification. Timing relations shown usually in timing diagrams can be described, and verification and synthesis against those can automatically be done.

A Hardware Description Language, called **Tokio**, which is based on temporal logic and is an extension of Prolog, is also presented. The above techniques can be applied to **Tokio**.

1 Introduction

Computer assistance is inevitable for the correct design of large digital systems. However, there are few tools to support hardware logic design: the only one is a simulator based on the descriptions in some hardware description languages (HDL's). So some tools to support synthesis and verification of logic design are highly required, especially in designing VLSI chips which have more than 100,000 gates, because errors in logic design may cause fatal delay of production. Establishing some methods to formally specify hardware is the first step to develop some synthesis and verification tools.

Hardware designers usually use timing diagrams to specify complex timing relations among signals. Generally speaking, diagrams are easily understood by them. However, the strict meaning is not defined, so the direct assistance of timing diagrams is not easy. From the point of both computer support and human readability, the use of some formal languages is good compromise. The requirements for such languages are:

(1) can express all relationships shown usually in timing diagrams,

(2) have enough mathematical backgrounds and are easy to be handled by computers.

Logic has strict mathematical background and is a good candidate for the method. However, normal logic (not extended logic, we call it "classical logic" from now on) has no time concept, or expresses facts of only one time (or state). In classical logic, once a variable has a value, it can no longer change the value. This means classical logic can not express sequential circuits directly. To express relations among states, each variable should be able to change their values as time proceeds. Thus temporal logic is introduced as an extension of classical logic to have that property. It has several additional operators concerning temporal sequences, which are called "temporal operators". Values of variables may change their values among states, and those changes are controlled by temporal operators.

Many kinds of temporal logic can be defined by introducing different kinds of temporal operators. Those include Linear Time Temporal Logic (**LTTL**) [11], Interval Temporal Logic (**ITL**) [13], Extended Temporal Logic (**ETL**) [16], Branching Time Temporal Logic (**BTTL**) [4], etc. Application of **LTTL** and **ITL** to the assistance of logic design of synchronous circuits is presented here. As for other temporal logics and their applications, please see references [16,4].

In section 2, **LTTL** is briefly introduced, and the method to describe various timing properties, such as timing diagrams, are shown. In section 3 and 4, verification and synthesis method and their experimental results, which handle mainly control part of hardware, are presented. In section 5, application of **ITL** to HDL's is shown. Finally section 6 gives concluding remarks.

2 Specification in Temporal Logic and its Reasoning

2.1 LTTL and its Descriptions of Timing Diagrams

In this section, we first briefly introduce temporal logic, then

0195-623X/88/0000/0254$01.00 © 1988 IEEE

show how to translate it into a state diagram, and finally give a decision procedure using the state diagram. In temporal logic, behaviors of systems are defined on infinite sequences of states; systems change their internal states as time advances, in other words, timing concept of temporal logic is discrete. Therefore, there must be next state of each state. In synchronous circuits, next time (or state) corresponds to next clock. **LTTL** [11] is an extension of the classical logic with four temporal operators, that is, ○ (next), □ (always), ◇ (sometime), and U (until). The first three are unary operators and the last is a binary operator. Each has the following meanings.

- P (without temporal operators): P is true at current state.
- $\bigcirc P$: P is true at the next state.
- $\Box P$: P is true in all future states.
- $\Diamond P$: P is true in some future state.
- P U Q: P is true for all states until the first state where Q is true.

To specify the property of concurrent systems, □ and ◇ are basically used as follows:

□ (*bad things do not happen*),

◇ (*good things happen*).

The former is so called safeness property and the latter is so called liveness property. For example,

□ (∼ deadlock),

or,

◇ (finish).

In describing hardware, in addition to the above, timing relations shown usually in timing diagrams must be specified. For example, the relation,

> Every state (clock) where signal P is active is immediately followed by a state in which signal Q is active.

which is shown in figure 1, is described as

$\Box(P \rightarrow \bigcirc Q)$ (1).

(→:IMPLY ∼:NOT ∨:OR ,:AND ∧:AND both "," and "∧" mean logical AND)

Similarly,

> Every state where signal P is active is followed (not always "immediately") by a state in which signal Q is active.

is described as

$\Box(P \rightarrow \Diamond Q)$ (2).

(1) or (2) guarantees "If P is active, then Q is active", but P may be active in other conditions. If the specification is "Q becomes active if and only if P is active", the following should be added.

$\Box(\sim Q \rightarrow ((\bigcirc \sim Q) U P))$ (3).

The basic timing relationship among signals can be described with the above expressions [5,7]. (1) means that $P \rightarrow \bigcirc Q$ is satisfied in all states. A different property $OneTime(P, Q)$: "From now on, if P, then Q on the next state, but thereafter there is no specification", is described in a different and a little complex way:

$$OneTime(P,Q) \equiv (P \rightarrow \bigcirc Q) \wedge ((\bigcirc(P \rightarrow \bigcirc Q)) U P).$$

The above means $(P \rightarrow \bigcirc Q)$ is satisfied until the first state where P is true. For example, an error handling for a microprocessor: "If an error occurs, then on the next state the system enters into a kind of idling state and remains there until the restarting signal becomes active. If so, then the system state becomes WORK on the next state", is described as:

$$\Box(ERROR \rightarrow \bigcirc(OneTime(RESTART, WORK))).$$

The above is a kind of specification of hardware. In contrast, designs are usually described in gate-level or some HDL's whose meanings are defined by state machines. Those design descriptions can also be described in temporal logic, mainly by using □ and ○ operators, i.e.,

$\Box(state1 \wedge cond1 \rightarrow \bigcirc state2)$,

etc. So, verification of logic design reduces to satisfiability check of temporal logic formulas. In the next section, a method to check the satisfiability is presented.

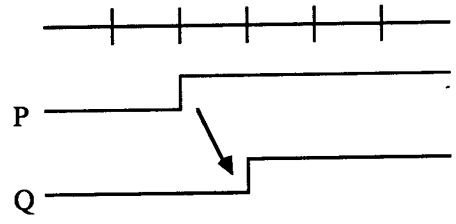


Figure 1 Relation of $\Box(P \rightarrow \bigcirc Q)$

2.2 Temporal Logic Decision Procedure

In this section, we show a method to translate **LTTL** formulas into state diagrams on which we can determine the satisfiability of the formulas. The basic idea of this method is that **LTTL** formula can be decomposed into sets containing formulas which are either atomic (that is, without temporal operators) or that have $\bigcirc$ as their main operator, where atomic sets are transitional conditions and remains except outmost $\bigcirc$ operator are conditions in next state (details are described in [16]). Decompositions are repeated until every condition in next states produced during decompositions are the same as those conditions already treated. Decomposition rules are as follows.

- $\Box F = F \wedge \bigcirc\Box F$
- $\Diamond F = F \vee (\sim F \wedge \bigcirc\Diamond F\{F\})$
- $F1\ U\ F2 = F2 \vee (F1 \wedge \sim F2 \wedge \bigcirc(F1\ U\ F2))$

For example, let P, Q and R are atomic and translate

(A) $\Box P$

(B) $\sim((P \wedge \bigcirc\Box Q) \rightarrow \bigcirc\Box R)$

into state diagrams using above rules.

(A) $\Box P = P \wedge \bigcirc\underline{\Box P}$

Since the condition in the next state "$\Box P$" (underlined) is same as the condition at present state, decomposition is completed and state diagram is as shown in figure 2.

(B) $\sim((P \wedge \bigcirc\Box Q) \rightarrow \bigcirc\Box R)$

$= (P \wedge \bigcirc\Box Q \wedge \sim(\bigcirc\Box R))$

$= (P \wedge \bigcirc\Box Q \wedge \bigcirc\Diamond(\sim R))$

$= (P \wedge \bigcirc\underline{(\Box Q \wedge \Diamond(\sim R))}$

$(\Box Q \wedge \Diamond(\sim R))$ is the next condition and decomposed as follows.

$(\Box Q \wedge \Diamond(\sim R))$

$= Q \wedge \bigcirc\Box Q \wedge (\sim R \vee (R \wedge \bigcirc\Diamond(\sim R)\{\sim R\})$

$= (Q \wedge \sim R \wedge \bigcirc\Box Q) \vee (Q \wedge R \wedge \bigcirc(\Box Q \wedge \Diamond(\sim R)\{\sim R\})$

And so, state diagram is as shown in figure 3.

LTTL formula is satisfiable iff it has an infinite sequence of state transitions when it is translated into state diagrams. Logic formula (A) is satisfiable because it has infinite sequences of state transition $< 1 >, < 1 >, < 1 >, \cdots$. Similarly, logic formula (B) is satisfiable because it has infinite sequences of state transition $< 5 >, < 5 >, \cdots$. Here, sequence $< 4 >, < 4 >, \cdots$ is not infinite because it does not satisfy eventuality $\{\sim R\}$. Eventuality $\{P\}$ means that P must eventually be true in all sequences of future states which follow the state P exists. Since "$\sim R$" is never true in sequence $< 4 >, < 4 >, \cdots$, this sequence cannot be infinite.

It is easy to check satisfiability of products of some logic formulas by tracing each state diagram concurrently. For example, we check the satisfiability of the formula

$$\Box(Q \wedge R) \wedge \sim((P \wedge \bigcirc\Box Q) \rightarrow \bigcirc\Box R).$$

To do so, we have only to trace state diagrams figure 3 and figure 4 concurrently. The results is shown in figure 5. Since there is no loop (even sequence $< 4, 6 >, < 4, 6 >, \cdots$ does not satisfy eventuality), this formula is unsatisfiable.

Figure 2 State Diagram for $\Box P$

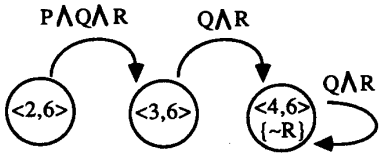


Figure 3 State Diagram for $\sim((P \wedge \bigcirc\Box Q) \rightarrow \Box R)$

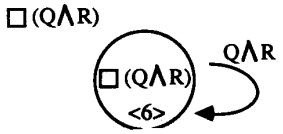


Figure 4 State Diagram for $\Box(Q \wedge R)$

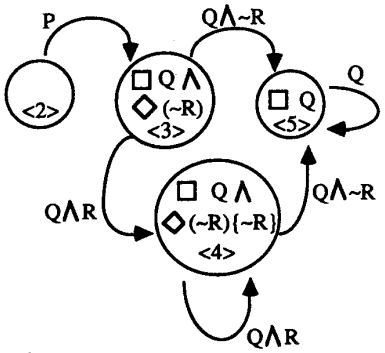


Figure 5 State Diagram for the Conjunction of Figure 2 and Figure 3

3 Verification

3.1 Verification Method

We verify that hardware designs really satisfy specifications. Let D be the temporal logic expression for hardware design and S be the temporal logic expression for the specification. We must investigate the following formula:

$$D \rightarrow S$$

is valid. To do so, we show that the negation of the formula, that is,

$$D \wedge \sim S$$

is unsatisfiable. In order to check it, we have only to do the following.

(1) To make state diagrams for $\sim$ S.

(2) To make state diagrams for D.

(3) To check whether there is any infinite sequence of state transitions for both state diagrams $\sim$ S and D.

The procedure of (3) is shown in figure 6 [14]. In the figure, NS is a state diagram translated from the negation of a specification. If there exists an infinite sequence, the design does not satisfy the specification (contradiction), and if not, the design is correct with respect to the specification S.

(1) of verification steps above is essentially a synthesis of state diagrams from temporal logic formulas. In fact, by translating S instead of $\sim S$, the result is the state diagram that we want. And so, implementation of (1) is discussed in section 4. As for (2) and (3) of verification steps, we have implemented it in two ways: one is to use Prolog's automatic backtracking and pattern matching mechanisms [7], and the other is to use cover (sum-of-products) expression on C language [14].

The former is simple and easy to understand, and the latter has much more execution efficiency than the former, especially in large circuits. The verification speed is heavily depend on the speed of reasoning of combinational circuits,

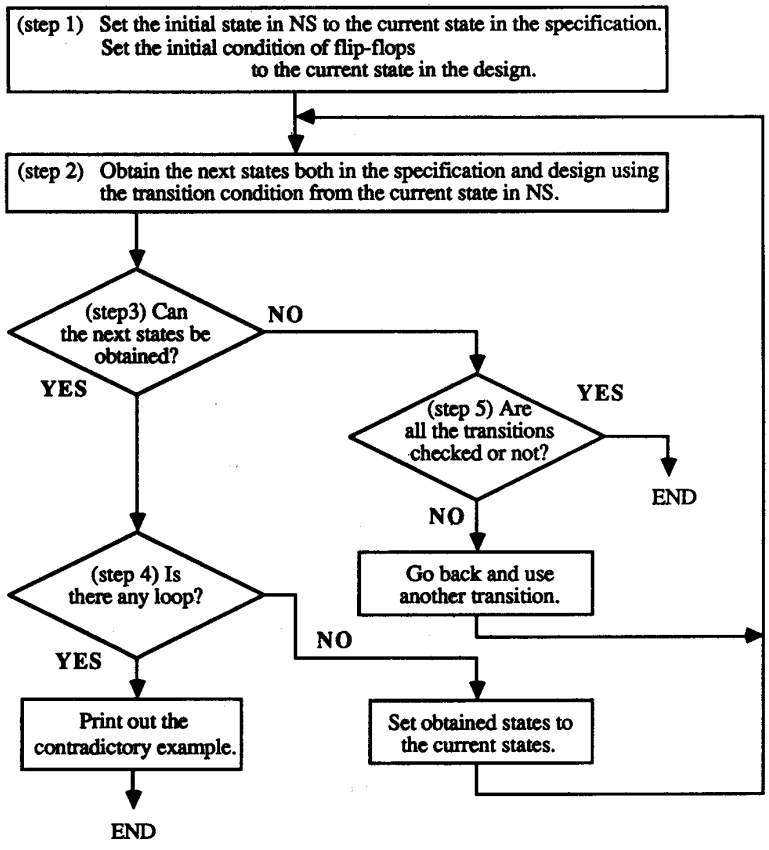


Figure 6 Verification Procedure

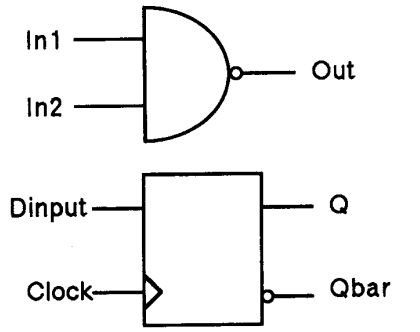


Figure 7 NAND Gate and D Flip-flop

that is, the speed to check whether a propositional formula of classical logic is satisfiable. There is another and promising method to check that: translating a combinational circuit into a decision graph [2]. We are now evaluating the method. The experiments show that large circuits, whose combinational part has more than 1,000 gates, can be verified in reasonable time.

3.2 Implementation in Prolog

Sequential circuits are directly converted into Prolog programs in the form of state transition tables [7]. Any hardware components, including flip-flops, can be described as a table containing current inputs, current outputs, current internal state, and next internal state. For example, a NAND gate and a D flip-flop (figure 7) are described as follows.

```
% dff([Dinput,Clock],[Q,Qbar],CurrentState,NextState).
  dff([D,0],[0,1],0,0).
  dff([D,0],[1,0],1,1).
  dff([D,1],[0,1],0,D).
  dff([D,1],[1,0],1,D).
% nand([In1,In2],[Out],CurrentState,NextState).
  nand([In1,0],[1],0,0).
  nand([0,In2],[1],0,0).
  nand([1,1],[0],0,0).
```

Here we follow the syntax in [3], a string beginning with a capital letter is a variable. The first line of "dff" shows that if Clock signal is 0, then the next internal state is the same as the current state regardless of Dinput. The third line of "dff" shows that if Clock signal is 1, then the next internal state is the same as the current Dinput. "nand" is described in the same way, except that it has the same internal state all the times, because "nand" is a combinational circuit and it has no internal states.

Networks of circuits are described as a clause of Prolog. For example, an EOR gate, consisting of four NAND gates and shown in figure 8, is described as follows.

```
eor([In1,In2],[Out],0,0):-
        nand([In1,In2],[N1],0,0),
        nand([In1,N1],[N2],0,0),
        nand([N1,In2],[N3],0,0),
        nand([N2,N3],[Out],0,0).
```

In the above, networks are described using the same variables as terminals connecting that networks.

Once a circuit is described in Prolog, we can calculate any input-output relations by simply executing those descriptions. Moreover, state diagrams which are translated from temporal logic formulas can also be described in the above form, and so verification can be done by repeated execution of those descriptions. Details of Prolog implementation are found in [5].

In examing gates, we must consider the case that the current input values are not fixed, i.e., undefined values of inputs. For example, let us consider that In1 = 1, and the value of In2 is not fixed in a NAND gate of figure 7. In this case, the value of O1 is unified to 1 at first because the first definition for NAND gates successes. Even if there does not exist a loop in this case, since there still remains the possibility of contradiction, the verification backtracks and the value of O1 is unified to 0 (the third definition for NAND gates successes), and the verification continues. Therefore, in this system, all the gates should be traced every time on obtaining the next condition of the flip-flops and there also exist many backtrackings, which lead to low efficiency of the verification. To raise the efficiency of the verification, the number of backtracking and tracing gates be reduced. Thus, we suggest two approaches for the efficiency;

(1) Use triple-valued logic and handle undefined value.

(2) Trace the combinational part of the design only once.

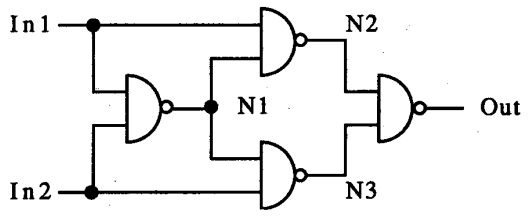


Figure 8 EOR Gate

3.3 Implementation in C by Cover Expression

Synchronous circuit is divided into combinational part and flip-flop part like figure 9. In C language implementation, combinational part is simulated using cover (sum-of-products) expression [14].

First covers and cubes are briefly explained. Let p be a product term associated with a sum of products expression of a logic function with n inputs $(x_1, ..., x_n)$ and m outputs $(f_1, ..., f_m)$. Then p is specified by a row vector $c = [c_1, ..., c_n, c_{n+1}, ..., c_{n+m}]$, where

$$c_i = \begin{cases} 10 & \text{if } x_i \text{ appears complemented in } p, \\ 01 & \text{if } x_i \text{ appears not complemented in } p, \\ 11 & \text{if } x_i \text{ does not appear in } p, \\ 0 & \text{if } p \text{ is not present in the representation of } f_{i-n}, \\ 1 & \text{if } p \text{ is present in the representation of } f_{i-n}, \end{cases}$$

c is called a cube. For example, suppose a logic function with 4 inputs and 2 outputs. For $f_1 = x_1x_2\overline{x_4}$, we have c = [01 01 11 10 1 0].

A set of cubes is said to be a cover C associated with a sum of products expression. For

$$f_1 = x_1x_2 + \overline{x_2}x_3 + x_1x_3;$$

$$f_2 = \overline{x_2}x_3 + \overline{x_3}\overline{x_4};$$

$$\text{we have C} = \begin{bmatrix} 01 & 01 & 11 & 11 & 1 & 0 \\ 11 & 10 & 01 & 11 & 1 & 1 \\ 01 & 11 & 01 & 11 & 1 & 0 \\ 11 & 11 & 10 & 10 & 0 & 1 \end{bmatrix}$$

The intersection (logical and) of two cubes c and d, written as $c \cdot d$, is a cube e. The entries e_i of the cube e are obtained from bit-and operation between cube c and cube d.

$$\text{Example.} \quad \begin{matrix} x_1x_2 & [01\ 01\ 11\ 1] \\ x_2x_3 & [11\ 01\ 01\ 1] \\ \Downarrow & \Downarrow \\ x_1x_2x_3 & [01\ 01\ 01\ 1] \end{matrix}$$

For a certain output variable f_i, the set of all cubes which makes f_i be 1 is called on-cover for the output variable f_i; similarly the set of all cubes which makes f_i be 0 is called off-cover for the output variable f_i.

To verify a circuit, combinational part of it is first translated into cover expression, and then verification is processed according to figure 6. For example, to verify the circuit shown in figure 10, the following on-cover and off-cover are generated. We describe each cube in the form of [Call, CY, Hear-i, Call-o, Hear-o, Hear]. (input variables are Call, CY, and Hear-i; output variables are Call-o, Hear-o, and Hear)

$$\text{Con} = \text{(on-cover)} \begin{bmatrix} 01 & 11 & 11 & 1 & 0 & 0 \\ 01 & 10 & 11 & 0 & 1 & 0 \\ 01 & 01 & 01 & 0 & 1 & 0 \\ 11 & 11 & 01 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Coff} = \text{(off-cover)} \begin{bmatrix} 10 & 11 & 11 & 1 & 0 & 0 \\ 10 & 11 & 11 & 0 & 1 & 0 \\ 11 & 01 & 01 & 0 & 1 & 0 \\ 11 & 11 & 10 & 0 & 0 & 1 \end{bmatrix}$$

Using on-cover and off-cover for the combinational part, two points for increasing the efficiency, which are mentioned in the last section, are realized.

The performance of this method is much higher than that of the last section, especially for large circuits (about 1,000 times faster). Although size of cover expressions remains small for control circuits, that for functional circuits, such as parity checker, becomes large (large number of rows in a cover). Therefore, this method can not be always applied to circuits which have fairly large functional units. To verify such circuits, more powerful method to check satisfiability of propositional formulas in classical logic is required. Promising one, we think, is a method to translate circuits into a decision graph [2]. The evaluation is in progress. Initial experiment shows it can verify circuits whose combinational circuits have more than 1,000 gates in reasonable time.

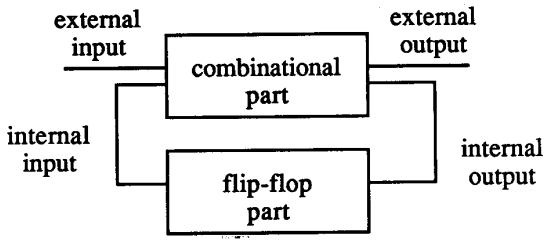


Figure 9 Synchronous Circuits

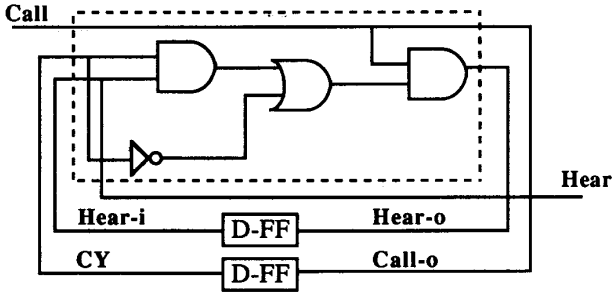
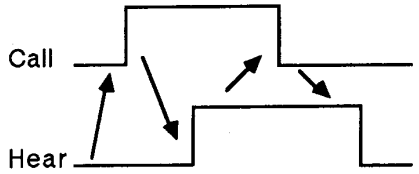


Figure 10 A Circuti Example



$\Box(\sim Hear \rightarrow \Diamond Call) \wedge \Box(\sim Call \rightarrow \sim Call\ U \sim Hear) \wedge$
$\Box(Hear \rightarrow \Diamond \sim Call) \wedge \Box(Call \rightarrow Call\ U Hear) \wedge$
$\Box(Call \rightarrow \Diamond Hear) \wedge \Box(\sim Hear \rightarrow \sim Hear\ U Call) \wedge$
$\Box(\sim Call \rightarrow \Diamond \sim Hear) \wedge \Box(Hear \rightarrow Hear\ U \sim Call)$

Figure 11 Handshake Sequence and its Description in Temporal Logic

4 Synthesis of State Diagrams

4.1 Prolog Implementation

The translation method presented in section 2 is easily implemented in Prolog [8]. If the temporal logic expressions are described in lists:

<1> $\Box P$ = [alw,P],
<2> $\Diamond P$ = [eve,P],
<3> $P1\ UP2$ = [unt,P1,P2],
<4> $\sim$ P = [not,P],
<5> $P1 \wedge P2$ = [and,P1,P2],
<6> $P1 \vee P2$ = [or,P1,P2],

the predicate "expand", which corresponds to <1> and <2> of the above, is like:

```
expand([alw,F], [[alw,F] | Fnn],E,En,P):-
                if_exist_delete(F,E,E1),
                expand(F,Fn,E1,En,P),
                simplify(Fn,Fnn).
expand([eve,F],Fnn,E,En,P):-
                if_exist_delete(F,E,E1),
                expand(F,Fn,E1,En,P),
                simplify(Fn,Fnn).
expand([eve,F],[[eve,F] | Fnn],E,[Ff | En1],P):-
                simplify(F,Ff),
                if_exist_delete([not,F],E,E1),
                expand([not,F],Fn,E1,En1,P),
                simplify(Fn,Fnn).
```

The first argument of "expand" is the expression to be expanded, the second is the expression in the next state, and the next two are the present and the next eventualities. The last is the list of the present values of variables. All the conditions in the next state are acquired by backtracking them compulsorily, and so state transitions are easily obtained using the procedure described above.

However, state transitions obtained from "expand" may have much redundancy. So, optimization of state transitions must be required. The optimization process is:

(1) eliminate states having no transitions from it,

(2) merge states having the same transitions,

(3) simplify conditions for transitions.

(2) and (3) are repeated as long as there is any improvement.

We apply this Prolog program to various temporal logic expressions. The results are shown in table 1. We used C-Prolog on VAX11/780, which was developed by Edingburgh University [15].

No.	*Temporal Logic Formula*	*(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*
1	$\Box(P \rightarrow \bigcirc Q)$	2	1.2	2	1.1	1	0
2	$\Box(P \rightarrow \bigcirc\bigcirc Q)$	4	2.6	4	2.7	1	0
3	$\Box(P \rightarrow \bigcirc Q \vee \bigcirc\bigcirc Q)$	4	3.4	3	2.5	1	0
4	$\Box(P \rightarrow \Diamond Q)$	2	2.0	2	1.4	1	1
5	$\Box(\sim Q \rightarrow \sim Q\ UP)$	2	2.0	2	1.4	1	0
6	$\Box(\sim Q \rightarrow (\bigcirc \sim Q)\ UP)$	3	3.7	2	2.4	2	0
7	$\Box(P \rightarrow \Diamond Q) \wedge$ $\Box(\sim Q \rightarrow \sim Q\ UP)$	8	27.7	4	18.2	3	1
8	$\Box(P \rightarrow \bigcirc Q) \wedge$ $\Box(\sim Q \rightarrow (\bigcirc \sim Q)\ UP)$	7	17.4	3	9.2	3	0
9	$\Box(S \rightarrow (P \rightarrow \bigcirc Q)\ UE)$	5	12.4	3	14.6	3	0
10	$\Box(S1 \rightarrow (S2 \rightarrow$ $(P \rightarrow \bigcirc Q)\ UE2)\ UE1)$	15	184.2	6	428.2	4	0
11	$\Box(P \rightarrow \bigcirc Q) \wedge$ $\Box(\sim Q \rightarrow (\bigcirc \sim Q)\ UP) \wedge$ $\Box(\sim P \rightarrow \bigcirc \sim Q) \wedge$ $\Box(Q \rightarrow \bigcirc Q\ U \sim P)$	11	108.7	2	20.7	4	0
12	$\Box(P \rightarrow \Diamond Q) \wedge$ $\Box(\sim Q \rightarrow \sim Q\ UP) \wedge$ $\Box(\sim P \rightarrow \Diamond \sim Q) \wedge$ $\Box(Q \rightarrow Q\ U \sim P)$	15	228.1	7	51.5	3	2
13	$\Box(P \rightarrow \Diamond Q) \wedge$ $\Box(\sim Q \rightarrow \sim Q\ UP) \wedge$ $\Box(\sim P \rightarrow \Diamond \sim Q) \wedge$ $\Box(Q \rightarrow Q\ U \sim P) \wedge$ $\Box(\sim Q \rightarrow \Diamond P) \wedge$ $\Box(\sim P \rightarrow \sim P\ U \sim Q) \wedge$ $\Box(Q \rightarrow \Diamond \sim P) \wedge$ $\Box(P \rightarrow P\ UQ)$	13	1140.3	5	27.8	3	4

(1): Number of States after Expansion
(2): CPU Time for Expansion
(3): Number of States after Simplification
(4): CPU time for Simplification
(5): Number of Repetition of Simplification
(6): Number of Eventuality
Executed by C-Prolog on VAX11/780

Table 1 Result of Synthesis of State Diagrams

We have not yet implemented temporal logic level optimization, i.e., $\Diamond\Diamond P \Rightarrow \Diamond P$. Also there are several points to be improved in implementation as well as the above point, so this program can be improved much faster.

4.2 Incremental Synthesis Method

There is another way to improve the efficiency. Specifications for hardware are usually expressed in the conjunction of fairly simple temporal logic expressions [2,3]. That is, a specification T is described as

(C) $T = T1 \wedge T2 \wedge \ldots \wedge Tn$,

where each Ti is fairly simple temporal logic expression.

For example, handshake sequence [1] is described by the conjunction of the eight expressions shown in figure 11.

This specification can be derived from the two temporal logic expressions: $\Box(A \rightarrow \Diamond B)$ and $\Box(\sim B \rightarrow\sim B\ UA)$ by changing the classical (not temporal) boolean expression, A and B, appropriately (i.e., $A \Rightarrow Call$ and $B \Rightarrow Hear$, etc.). Therefore, once $\Box(A \rightarrow \Diamond B)$ and $\Box(\sim B \rightarrow\sim B\ UA)$ are expanded, the state transitions for the eight expressions in figure 11 can be acquired easily. Moreover, the state transitions for $\Box(A \rightarrow \Diamond B) \wedge \Box(\sim B \rightarrow\sim B\ UA)$ be acquired by calculating the product of the two state transitions for $\Box(A \rightarrow \Diamond B)$ and $\Box(\sim B \rightarrow\sim B\ UA)$ respectively. Here, production means the intersection of all combinations of states and transitions. So the state transitions for the handshake sequence are calculated as the product of the eight state transitions. We call this method incremental synthesis. The incremental synthesis method basically has the same processing order in the worst case, but in practice, it is very efficient compared to the original method, because of the following two points:

(1) The incremental method only calculates products of state transitions. Although calculation of products has the same processing order as the expansion of a whole expression, there is no need of expanding temporal logic expressions.

(2) The optimization of state transitions in each expansion of temporal logic expressions reduces the number of state in the state transitions. So the results of production are usually smaller than those obtained from the expansion of a whole expression.

The flow of the incremental synthesis method is as follows:

(STEP1) The specification is assumed to be expressed in the form of (C). Expand temporal logic expressions which appear in Ti's and have different combination of temporal operators from each other, by using the program shown in section 2.

(STEP2) If necessary, replace variables by appropriate expressions (in classical logic) to express the real Ti's.

(STEP3) Calculate the production of each state transitions obtained from (STEP2), and check eventualities of each state transition, if some state transitions do not satisfy eventualities, delete them.

(STEP4) Simplify the state-diagram acquired from (STEP3). That is, put together the states having the same transitions and simplify conditions for transitions.

The state transitions obtained from the algorithm above can also be used in (STEP2), so the state transitions for a large number of expressions can be incrementally synthesized by repeatedly applying the algorithm above.

This method is implemented in Prolog. All cases must be checked to calculate the products of state transitions, which is easily programmed by using the Prolog's automatic backtracking mechanism. While making a production of state transitions, each eventuality attached to states are merged to attach the produced state.

The required CPU time for the incremental synthesis of the receiver of handshaking sequence (the first four temporal logic expressions in figure 11) is 22.2 seconds, which is much smaller than that shown in table 1 (279.6 seconds). We have already used this method for the automatic synthesis of state diagrams for control part of hardware. Please see [8] for the details.

5 Temporal Logic as a Hardware Description Language

5.1 Disadvantage of LTTL for HDL

Using **LTTL**, declarative specification, such as shown usually in timing diagrams and presented in section 2, are easily described. A complete specification is constructed as the conjunction of fairly simple expressions which should be satisfied in parallel. However, to describe sequential behavior, i.e., first execute A, and then execute B, we must specify the followings in **LTTL**:

- First A is executed,
- If A is finished, then B is executed,
- If A is in execution, B must not be executed,
- If B is in execution, A must not be executed.

In contrast, the above can be easily described in programming languages, such as C and Prolog.

When we use temporal logic as a HDL, the above point becomes serious. In **LTTL**, parallelism is easy to describe, but sequentiality is not. To overcome it, an idea of "interval" is developed, and a temporal logic with interval, called Interval Temporal Logic (**ITL**), is introduced [13,12].

5.2 ITL and Temporal Logic Based Language: Tokio

ITL is developed by B. Moszkowski. In this logic, meanings of variables and predicates are determined for each interval, while those of **LTTL** are determined in each state. An interval is a continuous finite sequence of states. In **ITL**, behaviors of a system is described from two points: relations within an interval and relations among intervals. The former is described in just the same way as **LTTL** using temporal operators introduced in section 2. To describe the latter, another temporal operator, "chop" is introduced. Chop operator is a binary operator. It splits an interval into two parts. Here && indicates chop operator.

(D) P && Q

has a relation among three intervals. Consider some interval $IALL$, and $IALL$ is divided into two parts which are Ip: the former part of $IALL$ and Iq: the later part of $IALL$. (D) is true for the interval IALL, if P is true for Ip and Q is true for Iq (figure 12). P and Q are any expressions containing any temporal operators in **LTTL** and &&.

Using && operator together with temporal operators in section 2, any relations which appears in describing hardware can be easily described. We developed a HDL, called **Tokio**, which is based on those temporal operators [10,6], as an extension of Prolog. **Tokio** has the same syntax as Prolog, and the relation in (5) is described as:

```
all :- p && q.
```

The above means that to execute all, execute p and then execute q. p and q may have temporal operators in **LTTL**.

Several hardware systems were described in **Tokio**, including a special purpose micro processor, and were simulated by **Tokio** simulator, which compiles **Tokio** programs into Prolog. Using logic programming languages like **Tokio**, structure informations as well as behavioral informations can be described in an unified manner. Details of **Tokio** are found in [10,6].

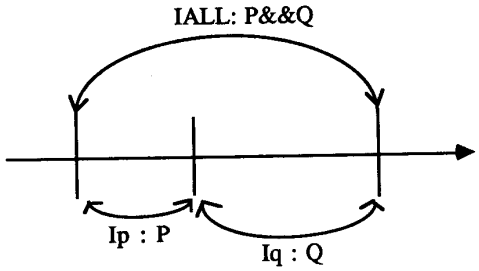


Figure 12 Relation of Intervals for $P\&\&Q$

6 Conclusions

Hardware specification methods in temporal logic, verification and synthesis methods which are based on the temporal logic decision procedure, and a temporal logic based HDL: **Tokio** are presented. C-version of the verification system has enough practical power for control part of hardware, such as PLA-based state machines. We are now implementing the synthesis and verification methods in C language using the decision graph [2]. Initial estimation shows that this new system can handle large circuits more than 1,000 gates.

We are also developing a supporting environment for **Tokio**, which includes a formal verifier utilizing the theorem proving techniques like in [9]. To design a large system, hierarchical and structured design must be applied. The key point of assistance to such designs is to check logical equivalence of two descriptions: one is for upper level of hierarchy and the other is for lower level of hierarchy. The formal verifier tries to check it as a theorem proof, and will be an essential tool.

Acknowledgement

The author would like to thank Mr. S. Kono and Mr. H. Nakamura of Tokyo University for their deep contribution to the implementation. The author would also like to thank Mr. S. Sato, Mr. J. Tanahashi, Mr. M. Ishii, and Dr. N. Kawato for their continuous encouragements.

References

[1] G.V. Bochmann. Hardware specification with temporal logic: an example. *IEEE Trans. Computer*, C-31(3):223–231, March 1982.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computer*, C-35(8):667–691, August 1986.

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

[4] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuit using temporal logic. In *Proceedings. Pt. E*, pages 276–282, IEE, September 1986.

[5] M. Fujita. *Logic Design Assistance with Temporal Logic.* PhD thesis, Dept. of Information Engineering, University of Tokyo, 1984.

[6] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: logic programming language based on temporal logic and its complication to prolog. In *3rd ICLP*, London, 1986.

[7] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *IFIP 7th Computer Hardware Description Languages and their Application*, IFIP, August 1985.

[8] M. Fujita, H. Tanaka, and T. Moto-oka. Specifing hardware in temporal logic & efficient synthesis of state-diagrams using prolog. In *FGCS '84*, November 1984.

[9] M.J.C. Gordon and J. Herbert. Formal hardware verification methodology and its application to a network interface chip. In *Proceedings. Pt. E*, pages 255–270, IEE, September 1986.

[10] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of temporal logic programming language tokio. In *LPC'85, Lecture Notes in Computer Science*, 1985.

[11] Z. Manna and A. Pnueli. *Verification of Concurrent Programs, Part 1: The Temporal Frame work.* Technical Report STAN-CS-81-836, Dept. of Computer Science, Stanford University, June 1981.

[12] B.C. Moszkowski. *Executing Temporal Logic Programs.* Computer University Press, 1984.

[13] B.C. Moszkowski. *Reasoning about Digital Circuits.* Technical Report STAN-CS-83-970, Stanford University, June 1983.

[14] H. Nakamura, M. Fujita, S. Kono, and H. Tanaka. Temporal logic based fast verfication system using cover expressions. In *VLSI '87*, IFIP, August 1987.

[15] F. Pereira. C-prolog users manual version 1.5. 1984.

[16] P. Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundation of Computer Science*, October 1981.