PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

SPECIFYING HARDWARE IN TEMPORAL LOGIC & EFFICIENT SYNTHESIS OF STATE-DIAGRAMS USING PROLOG

Masahiro FUJITA, Hidehiko TANAKA, Tohru MOTO-OKA

Department of Electrical Engineering
The University of Tokyo
7-3-1 Hongo Bunkyo-ku, Tokyo 113 Japan

ABSTRACT

We have already proposed a new verification method for hardware logic design with temporal logic and Prolog (Fujita et al. 1983a, 1983b, 1984a), which supports hierarchical design consistently. Specifications are written in temporal logic (Manna and Pnueli 1981, Wolper 1981), which is an extension to traditional logic and can easily describe timing relations among variables. And the verifier for gate and state-diagram designs is easily implemented using Prolog.

In this paper, in addition to the techniques for specifying hardwares in temporal logic, we present the efficient method for automatic synthesis of state-diagrams from temporal logic specifications. We also show it has much practical power by applying it to real hardwares and comparing to the manual design. State-diagrams can be synthesized by expanding the specifications to the conditions at present and the ones in the next time using the temporal logic decision procedure (Wolper and Manna 1981, Clarke and Emerson 1981). As the required time for the original procedure increases exponentially with the number of temporal operators (Wolper and Manna 1981, Clarke and Emerson 1981), we show the efficient method for implementing the procedure with the automatic backtracking and the pattern matching mechanisms of Prolog. The method regards state-diagrams already synthesized as a knowledge and synthesizes the global state-diagrams incrementally. The synthesis time is drastically reduced (to the polynominal order) and is kept small enough with fairy large hardwares.

1 INTRODUCTION

In recent years, computer systems become larger and more complicated, and they have got the most essential part in our lives. This means hardware designs must have much reliability and one can not make logic designs without computer assistances. Some methods and tools supporting hierarchical designs smoothly, that is, with which a designer can formally specify hardwares and verify designs, or designs are automatically synthesized from specifications, are indispensable for reliable designs.

We have already proposed a new verification method for hardware logic design with temporal logic and Prolog (Fujita et al. 1983a, 1983b, 1984a), which supports hierarchical design consistently. Specifications are written in temporal logic (Manna and Pnueli 1981, Wolper 1981), which is an extension to traditional logic and can easily describe timing relations among variables. And the verifier for gate and state-diagram designs is easily implemented using Prolog.

Hardware systems are divided into two parts: function part, which actually executes logic and arithmetic functions, and synchronization part, which controls timings for data transfer among function units. Synchronization part requires a designer to consider the whole design working in parallel, may have many design errors, and should be automatically verified. The method proposed in (Fujita et al. 1983a, 1983b, 1984a) is concentrated on the verification of synchronization part. Designs are verified by automatically examining all the cases truly needed. Several methods for increasing the efficiency of verification are also proposed (Fujita et al. 1984a), and they keep the verification time for complex and large systems small enough.

In this paper, we first show the techniques for specifying hardwares in temporal logic and second present the efficient method for automatic synthesis of state-diagrams from those temporal logic specifications. We also show it has much practical power by applying it to real hardwares and comparing to the manual design. State-diagrams can be synthesized by expanding the specifications to the conditions at present and the ones in the next

the conditions at present and the ones in the next time using the temporal logic decision procedure (Wolper and Manna 1981, Clarke and Emerson 1981). As the required time for the original procedure increases exponentially with the number of temporal operators (Wolper and Manna 1981, Clarke and Emerson 1981), we show the efficient method for implementing the procedure using Prolog. We specify hardwares in the form of logical AND of simple temporal logic expressions. The method regards state-diagrams already synthesized as a knowledge and synthesizes the global state-diagrams incrementally. The synthesis time is drastically reduced (to the polynominal order) and is kept small enough with fairy large hardwares.

Since the synthesis method translates any temporal logic expressions to state-diagrams, it is also possible to check whether or not hardware designs in temporal logic really satisfies some other temporal logic expressions. Moreover, together with the verification method for gates and state-diagrams in (Fujita et al. 1983a, 1983b, 1984a), any designs in temporal logic, state-diagrams, and gates can be verified in the same way, and then the hierarchical design is smoothly supported.

Section 2 informally introduces temporal logic and explains how to describe hardware with it by showing an example of real hardwares. It also presents the techniques for behavior descriptions in temporal logic. Section 3 shows the techniques of how to describe state-diagrams in Prolog. Section 4 presents the synthesis method based upon the temporal logic decision procedure and its efficient implementation in Prolog. Section 5 shows the synthesis example, and section 6 discusses the power of the synthesis method. The last section has concluding remarks.

2 HARDWARE SPECIFICATION IN TEMPORAL LOGIC

2.1 Specification of Synchronization Parts in Temporal Logic

This section briefly introduces temporal logic and explains how to specify synchronization parts of hardware with it. The detailed discussions about temporal logic can be found in (Manna and Pnueli 1981, Wolper 1981).

Temporal logic is an extension to traditional logic with four temporal operators: ○ (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: P is true in the next time (in sequential circuits, next clock),

□P: P is true at present and all the future times,

▽P: P is true at least on a time at present or in the future,

P U Q: P is true all the times until the first time where Q is true.

Temporal logic can describe temporal sequences and therefore can express timing relations among variables shown usually in timing diagrams. 'If the signal P is active, then the signal Q is active on the next time' is described as

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

(→: IMPLY, ∧: AND, ∨: OR, ~: NOT)

If the time when Q is active is not definite, ○ is replaced by ▽.

$$\Box(P \rightarrow \nabla Q) \quad (2)$$

(1) and (2) guarantee 'If P is active, then Q is active', but P may be active otherwise. If it is desired 'Q becomes active if and only if on the next time when P is active', the following expression should be added to (1).

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

The basic timing relationship among signals can be described with (1) [or (2)] and (3).

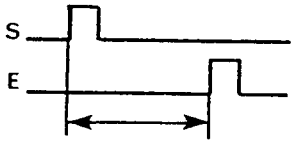


Fig.1 A Timing Chart

During this interval

P→○Q

For example, the relationship shown in fig.1 (Fujita et al. 1984a): 'During the period from the time when the start signal S is active till the time when the end signal E is active, if P is active, then Q is active on the next time' is described as

$$\Box(S \rightarrow ((P \rightarrow \bigcirc Q)\ U\ E)). \quad (4)$$

[(4) assumes that S and E are externally given as a pulse like in fig.1. This is shown in temporal logic as follows.

$$\Box(S \rightarrow ((\bigcirc\sim S))\ U\ E)),$$
$$(\bigcirc\sim E)\ U\ S,$$
$$\Box(E \rightarrow ((\bigcirc\sim E)\ U\ S)) \ ]$$

(expressions marked off by a comma mean logical AND of those.)

As seen from (4), a complex specification requires temporal operators nested many times. However, if an interval I such that, I is active during the period from the time when S is active till the time when E is active and inactive during the period from the time when E is active till the time when S is active, is introduced, the specification (4) is described with the logical AND of the following simple expressions.

$\Box(\sim I \rightarrow ((\sim I)\ U\ S))),$
$\Box(S \rightarrow I),$
$\Box(I \rightarrow (I\ U\ E)),$ (5)
$\Box(E \rightarrow (\sim I)),$
$\Box((I \wedge P) \rightarrow \bigcirc Q)$

The first four expressions guarantee that I is active only in the period from the time when S is active till the time when E is active. And the last expression means that all the time when I is active, if P is active, Q is active on the next time.

As a rule, introducing an interval like I in (5), complex specifications can be described with logical AND of simple expressions. Moreover, as shown in 2.3, we can easily describe behavioral (procedural) descriptions such as algorithms using intervals (see 2.3). Therefore, not only declarative but also procedural descriptions can be done in temporal logic. Although an interval I shows an internal state, which is not necessarily observable from the outside, it makes specifying a module much easilier. Also, as there are many simple expressions of the same form as seen in (5), the synthesis time of complex specifications is kept manageably small as described in the later sections.

2.2 Specification Example of Synchronization Part

This section shows the specification example of the real hardware: the 'sequencer controller' of the Unify Processor (UP) of the parallel inference machine PIE. PIE (Moto-oka et al. 1984) is a highly parallel inference machine. UP (Yuhara et al. 1984) is a processor executing unification by hardware and its pilot machine is being constructed. The pilot UP machine (Yuhara et al. 1984) has the scale of 500 TTL ICs and some memory ICs and microprogrammed control. Its internal structure is shown in fig.2. The goal registers and definition registers are respectively connected to the goal bus and definition bus, and the buses are connected through the switch to the memory divided into four banks. The goal registers and definition registers access one of the four banks. If they access the same bank, the goal registers get the right of the access. However, some micro instructions require successively accesses to memory depending on the data in the memory (we call this kind of micro instructions 'successive access instruction'). In that case, if the other registers try to access to the same bank, it should be waited. Also, in the case of executing the 'successive access instruction', the micro program sequencer should be informed to continue the current micro instruction in the later micro cycles. These are controlled by the 'sequencer controller' in fig.2, and we specify it.

First, we describe the meanings of the variables appeared in the specifications.

fet: signal expressing whether the 'successive access instruction' is executed or not,
wait: signal informing the sequencer to continue the current micro instruction,

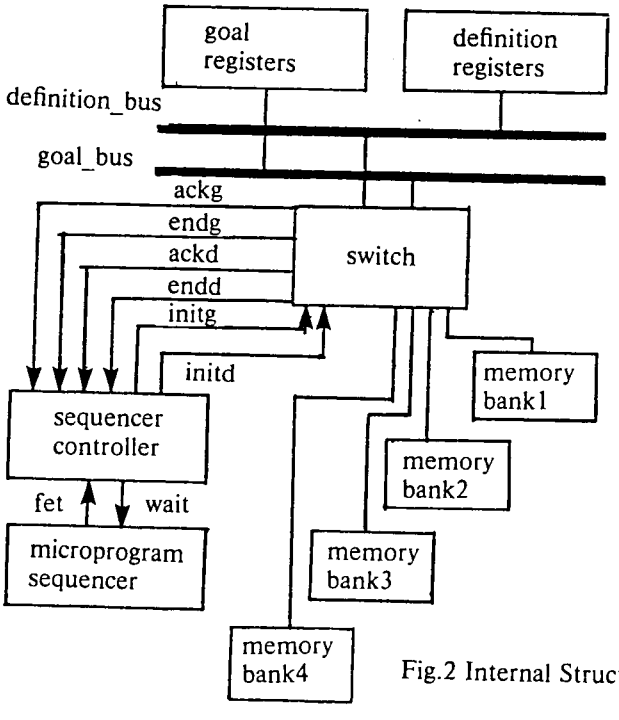


Fig.2 Internal Structure of UP

initg: signal expressing the first access of the goal's registers in the current 'successive access instruction',

endg: signal expressing whether the successive access of the goal's registers is ended or not,

ackg: signal expressing whether the goal registers really get the right of access to memory or not,

initd: signal expressing the first access of the definition's registers in the current 'successive access instruction',

endd: signal expressing whether the successive access of the definition's registers is ended or not,

ackd: signal expressing whether the definition registers really get the right of access to memory or not,

Now we specify the 'sequencer controller'. As the logical AND of the temporal logic expressions (1) and (3) above is used very often, the next abbreviations are defined.

IFF_NEXT(a,b) =
$\Box(\bar{a} \rightarrow \bigcirc b) \wedge \Box(\sim b \rightarrow ((\bigcirc \sim b)\ U\ a))$ **(6)**

IFF_PRESENT(a,b) =
$\Box(\bar{a} \rightarrow b) \wedge \Box(\sim b \rightarrow (\sim b\ U\ a))$ **(7)**

The 'sequencer controller' accepts 'ackg', 'ackd', 'endg', 'endd', and 'fet' as inputs and controls 'initg', 'intd', and 'wait'. The specifications are described in temporal logic as follows.

(S-1) The goal registers' access of the 'successive access instruction' goes from the first access (initg) to the second access (~initg), if and only if on the next time when the 'successive access instruction' is being executed and the goal registers really get the right of access (ackg).
IFF_NEXT((initg∧fet∧ackg), ~initg)

(S-2) The same fact must be satisfied for the definition registers.
IFF_NEXT((initd∧fet∧ackd), ~initd)

(S-3) The goal registers' access of the 'successive access instruction' goes to the first access (initg) of the next 'successive access instruction', if and only if on the next time when the registers cease to access to memory (~fet), or the 'successive access instruction' is being executed (fet) and both the goal and definition registers end the accesses (endg∧endd) and the access is not the first one (~initg).
IFF_NEXT((~fet∨(fet∧endg∧endd∧~initg)), initg)

(S-4) The same fact must be satisfied for the definition registers.
IFF_NEXT((~fet∨(fet∧endg∧endd∧~initd)), initd)

(S-5) The sequencer is begun to wait, if and only if the first access of the 'successive access instruction' is being executed.
IFF_PRESENT(((initg∨initd)∧fet), wait)

(S-6) The sequencer ceases to wait on the same conditions of (S-3) and (S-4).
IFF_PRESENT((~fet∧(fet∧endg∧endd∧~initg∧~initd)), ~wait)

The switch in fig.2 can also be specified in the same way. See (Fujita et al. 1984b) for the details.

2.3 Behavior Descriptions Using Intervals

Not only declarative but also procedural descriptions must be necessary to smoothly describe behaviors of hardwares. Procedural descriptions have two aspects: parallel and sequential. It is easy to describe parallelisms in temporal logic. Parallelisms are described simply in the form of logical AND of each action. For example, 'the two actions: $P \rightarrow \bigcirc Q$ and $R \rightarrow \bigcirc S$ are working in parallel' is described as
$(P \rightarrow \bigcirc Q) \wedge (R \rightarrow \bigcirc S)$. **(8)**

However, it is tedious and not easy to describe sequentialities. For example, when we describe the fact such that 'first execute P and then execute Q', that is, in PASCAL,

```
begin
  P;
  Q      (9)
end
```

the following things must be specified.

First P is executed.
All the time when P is executed, Q is not executed.
If P is ended, then Q is started to execute.
All the time when Q is executed, P is not executed. **(10)**

If we use some signals expressing whether P and Q is executed or not, the conditions above are easily described. That is, it is very useful to introduce intervals that are exactly active during the period when actions such as P and Q are executed like section 2.1. Let Ip and Iq be interval signals respectively expressing whether P and Q are executed or not, and let Ip.beg and Iq.beg be signals expressing the beginning time of those intervals and Ip.fin and Iq.fin be signals expressing the ending time of those intervals. Since Ip.beg, Ip.fin and Ip are the signals for the same interval, the following conditions are assumed.

First Ip.beg and Ip is active.
In the following some period Ip is active.
Last Ip.fin is active.

[These are described in temporal logic as follows.
□(Ip.beg→((Ip∧○~Ip.beg) U (Ip∧○Ip.fin))]

Then, **(10)** is described as follows.
□(Ip→P), □(Iq→Q)
Ip.beg,
□(Ip.fin→Iq.beg), **(11)**
□(~(Ip∧Iq)).

Note that Ip.beg, Ip.fin, Iq.beg and Iq.fin have almost the same meanings as in Interval Temporal Logic by Moszkowski (Moszkowski 1983). However, we describe them only in Linear Time Temporal Logic (Manna and Pnueli 1981, Wolper 1981) for ease mechanical handlings.

Interval signals are additional variables and indicate internal states of the hardware specified, and therefore, they are not necessarily observable the outside. However, the idea of intervals makes both parallel and sequential descriptions much easier. Moreover, using intervals, we can specify behaviors of hardwares in logical AND of simple temporal logic expressions, which drastically reduces the required time for synthesis and verification (Fujita et al. 1984a). We are now studying on a hardware description language, which is based upon intervals and Linear Time Temporal Logic, and its assistant tools using Prolog. Some results are already reported (Fujita et al. 1984c). The details will be appeared elsewhere.

3 STATE-DIAGRAM DESCRIPTIONS IN PROLOG

This section presents how to describe state-diagrams in Prolog and how to get the global state-diagram from given state-diagrams. The syntax of Prolog used here is that in the book 'Programming in Prolog' (Clocksin and Mellish 1981).

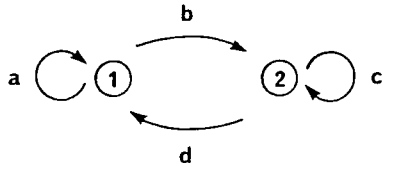
State-diagrams are expressed with the present values of variables, the present state, and the next state, and so, they are described in Prolog with the table of those values.

For example, a state-diagram shown in fig.3 (a) is described in Prolog as in (b). 'a', 'b', 'c', and 'd' in fig.3 (a) are some logical expressions (traditional logic) for state transitions, and '1' and '2' are state names. The first two arguments of 'state_diagram1' in (b) mean the present state and the next state, and the last argument (P) is the list of the values of variables. Each definition of 'state_diagram1' corresponds to state transitions. 'logic' is a predicate expressing the logical expressions for 'a', 'b', 'c', and 'd'. Assigning adequate logical expressions to 'a', 'b', 'c', and 'd', fig.3 (b) can express various state-diagrams. For example, in order to express the state-diagram in fig.3 (c), the 'logic' in fig.3 (d) should be added to (a). The first argument of 'logic' is the type of conditions and the second is the list of the values of variables. As seen above, the same forms of state-diagrams (the states and the relations of state transitions are the same) are expressed in a single clause of Prolog using 'logic'.

The global state-diagram of given state-diagrams is easily acquired with Prolog. For example, in order to get the global state-diagram of the two state-diagrams (fig.4 (a)), a new predicate for the global state-diagram whose head is the global state-diagram's name and whose bodies are the two descriptions in Prolog for state-diagrams, 'state_diagram1' and 'state_diagram2' (c), is defined as in (b). The execution :

```
?-state_diagram_all(S,Sn,P),print([S,Sn,P]),fail.
```

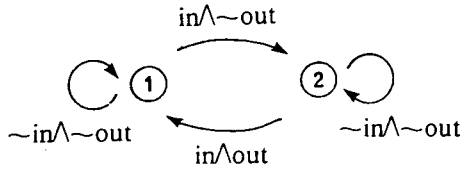
gives the global state-diagram in fig.4 (a). Using the descriptions for state-diagrams as bodies, the global state-diagram is easily acquired.



(a) A State-diagram

```
state_diagram1(1,1,P):- logic(a,P).
state_diagram1(1,2,P):- logic(b,P).
state_diagram1(2,2,P):- logic(c,P).
state_diagram1(2,1,P):- logic(d,P).
```

(b) Prolog Descriptions for (a)

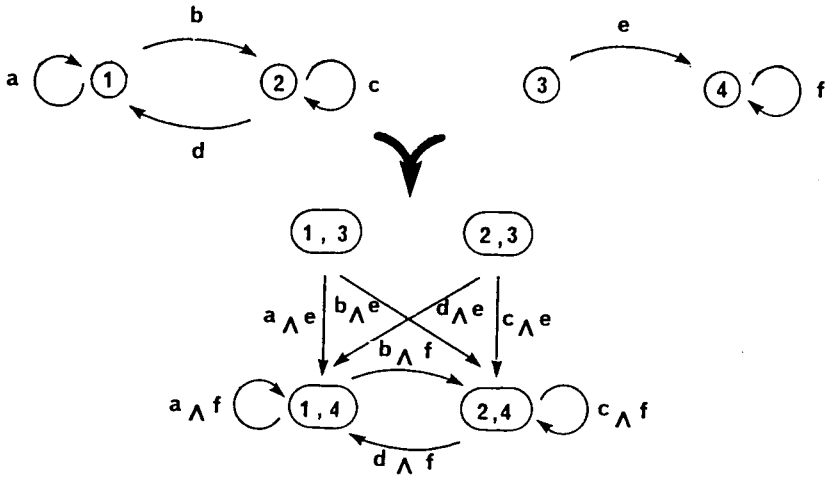


(c) A State-diagram Specialized from (a)

```
logic(a,[0,0]).
logic(b,[1,0]).
logic(c,[0,0]).
logic(d,[1,1]).
      in out
```

(d) 'logic' Added to (b)

Fig.3 A State-diagram and its Descriptions in Prolog



(a) State-diagrams and their Global State-diagram

```
state_diagram_all([S1,S2],[Sn1,Sn2],P):- state_diagram1(S1,Sn1,P),
                                         state_diagram2(S2,Sn2,P).
```

(b) Prolog Descriptions for (a)

```
state_diagram2(3,4,P):- logic(e,P).
state_diagram2(4,4,P):- logic(f,P).
```

state_diagram1 is the same as in fig.3 (b)

(c) State-diagram Descriptions for (a)

Fig.4 State-diagram and their Global State-diagram in Prolog

4 THE SYNTHESIS METHOD AND ITS EFFICIENT IMPLEMENTATION IN PROLOG

This section briefly explains the synthesis method based upon the temporal logic decision procedure (Wolper 1981) and shows its efficient implementation method in Prolog. See (Wolper and Manna 1981, Clarke and Emerson 1981) for the details of the decision procedure.

The synthesis method in (Wolper and Manna 1981, Clarke and Emerson 1981) is intuitively as follows.

First, the specifications in temporal logic expressions are expanded to the conditions at present and the ones in the next time using the temporal logic expansion rules. Next, the outmost ○ operators of the conditions in the next time are removed and the rest conditions are also expanded to the conditions at present and the ones in the next time. This expansion is repeated until the conditions in the next time are the same as the ones already treated. After treating all the conditions appearing in the expansion procedure, a state-diagram, whose states correspond to the conditions and whose state transitions correspond to the expansion process, is obtained.

The expansion rules for each temporal logic operator are shown in table 1. Table 1 is acquired from the temporal logic axioms and, <1> in table 1 indicates that '□P means that P should be satisfied at present and □P should also be satisfied in the next time'. And <2> indicates '▽P means that P is satisfied at present, or P is not satisfied at present and ▽P should be satisfied in the next time'. However, if ~P∧○▽P of <2> is always taken, ▽P will not be satisfied (□~P is satisfied). Therefore, when ~P∧○▽P of <2> is taken, the condition that P will eventually be satisfied must be added. This is called 'eventuality' and {P} behind ~P∧○▽P of <2> indicates that.

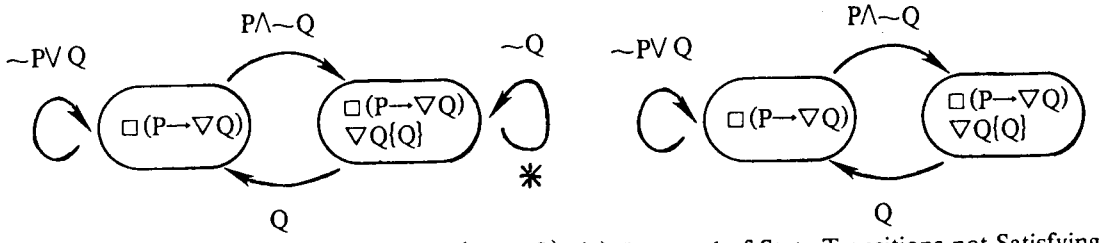
Using this table, any temporal logic expressions can be expanded into the conditions at present and the ones in the next time. After treating all the conditions appearing in expansion process, if some eventualities appear, the state-

<1> □P=P∧○□P
<2> ▽P=P∨(~P∧○▽P{P})
<3> P1 U P2=
P2∨(P1∧~P2∧○(P1 U P2))
<4> ~□P=~P∨(P∧○(~□P){~P})
<5> ~▽P=~P∧○(~▽P)
<6> ~(P1 U P2)=(~P1∧~P2)
∨(~P2∧○(~(P1 U P2)){~P1})

Table 1 Expansion Rules for Temporal Logic Operators

<1> □P=[alw,P]
<2> ▽P=[eve,P]
<3> P1 U P2=[unt,P1,P2]
<4> ~P=[not,P]
<5> P1∧P2=[and,P1,P2]
<6> P1∨P2=[or,P1,P2]

Table 2 List Expressions for Temporal Logic



(a) State_diagram Synthesized from □(P→▽Q) (b) Removal of State Transitions not Satisfying Eventualities

Fig.5 State-diagram Synthesized from □(P→▽Q)

present conditions
next conditions
present and next eventualities
present values of variables

```
expand([alw,F],[[alw,F]|Fnn],E,En,P):-
        if_exist_delete(F,E,E1),        <- if F is in E, E is deleted from F
        develop(F,Fn,E1,En,P),             and the rest is E1
        simplify(Fn,Fnn).               <- simplification of expressions
expand([eve,F],Fn,E,En,P):-
        if_exist_delete(F,E,E1),
        develop(F,Fn,E1,En,P).
expand([eve,F],[[eve,F]|Fnn],E,[F|En1],P):-
        simplify(F,Ff),
        if_exist_delete([not,F],E,E1),
        develop([not,F],Fn,E1,En1,P),
        simplify(Fn,Fnn).
```

Fig.6 'expand' Corresponding to Table 1

diagram is checked whether each state transition path really satisfies those eventualities. If not, it is deleted.

We expand □(P → ▽Q) as an example. First, □(P → ▽Q) is expanded as follows, referring to <1> and <2> of table 1.

□(P → ▽Q)
= (P → ▽Q)∧○□(P → ▽Q) (by <1>)
= (~P∨(P∧▽Q))∧○□(P → ▽Q)
= (~P∨(P∧(Q∨(~Q∧○▽Q{Q}))))
∧○□(P → ▽Q) (by <2>)
= ((~P∨Q)∧○□(P → ▽Q))
∨(P∧~Q)∧○(□(P → ▽Q)∧▽Q{Q}) **(12)**

As □(P → ▽Q)∧▽Q{Q} is appeared as the conditions for the next time, it is expanded next in the same way.

□(P → ▽Q)∧▽Q{Q}
=(Q∧○□(P → ▽Q))
∨(~Q∧○(□(P → ▽Q)∧▽Q{Q})) **(13)**

(12) and **(13)** are the expressions already treated. The state-diagram obtained from **(12)** and **(13)** is shown in fig.5 (a). However, the continuous taking the state transition * in (a) does not satisfy the eventuality {Q}, so this state transition is deleted as shown in (b).

The procedure above is easily implemented in Prolog. If the temporal logic expressions are described in lists as seen in table 2, the predicate 'expand', which corresponds to <1> and <2> of table 1, is shown in fig.6. The first argument of 'expand' is the expression to be expanded, the second is the expressions in the next time, and

the next two are the present and the next eventualities, which are used for the 'eventuality check' described above. The last is the list of the present values of variables. All the conditions in the next time are acquired by backtracking them compulsorily, and so a state-diagram is easily obtained using the procedure described above.

However, the required time for this procedure increase exponentially with the number of the temporal operators in the specifications. So, some methods for increasing the efficiency must be required for the practical use.

Seen from section 2, specifications are expressed with logical AND of simple expressions, and each simple expression has the same form.

$T = T1 \wedge T2 \wedge \ldots \wedge Tn$ **(14)**

T is the specification and each Ti is a fairly simple expression. If each Ti is expanded in advance, the state-diagram for the specification T can be synthesized by making the global state-diagram from state-diagrams for Ti. In fact many Ti have the same form of temporal logic expressions (in the sense that those are all expressed in the same Prolog descriptions like in fig.3 (a)). Using state-diagrams for Ti as a knowledge, this method drastically reduces the required time. The efficient synthesis algorithm is as follows.

(STEP1) The specification is assumed to be expressed in the form of **(14)**. Expand each Ti using the predicate 'expand' in fig.6.

(STEP2) Make the global state-diagram from the state-diagrams for Ti in the same way as in section 3. If necessary, check eventualities of each state transition. If some state transitions do not satisfy eventualities, delete them.

(STEP3) Simplify the state-diagram acquired from (STEP2). That is, put together the states having the same transitions and simplify conditions for transitions.

The state-diagram obtained from the algorithm above can also be used in (STEP2), so the state-diagram for the large hardware can be incrementally synthesized by repeatedly applying the algorithm above. If the number of states of the state-diagrams synthesized is not so big, this strategy has enough practical power. Also, the state-diagrams used in (STEP2) may be ones obtained from a register transfer level hardware description language like DDL or gate level descriptions (Fujita et al. 1983a), the state-diagram for the new hardware which is an extension to the already designed hardware by extra specifications can be synthesized in the same way.

The algorithm above is implemented in C-Prolog which is developed at Edinburgh University (Pereira 1984). The program size is about 1400 steps.

As for the verification, in order to check whether a temporal logic specification satisfies the design or not, one has only to expand the negation of the specification into the state-diagram by the algorithm above, and then check whether the design satisfies the negation of the specification or not as shown in (Fujita et al. 1983a, 1983b, 1984a).

5 SYNTHESIS EXAMPLE

This section presents a synthesis result of the 'sequencer controller' in section 2.2 as an example. In order to reduce the synthesis time, the global state-diagram is synthesized incrementally. The synthesis flow is as follows.

(1) As the specifications (S-1)~(S-6) have the form of IFF_NEXT and IFF_PRESENT, IFF_NEXT and IFF_PRESENT are expanded first using 'expand' in fig.6, according to (STEP1) of the synthesis algorithm.

(2) (STEP2) is executed next. 'a' and 'b' in IFF_NEXT and IFF_PRESENT are replaced to fit the expressions (S-1) ~ (S-6), using 'logic' in section 3. Since (S-1) ~ (S-6) have a conflict of themselves, specifications for the 'switch' in fig.2 (Fujita et al. 1984b):
$\Box$(endg $\rightarrow$ (endg U (initg$\vee\sim$fet))) **(15)**,
$\Box$(endd $\rightarrow$ (endd U (initd$\vee\sim$fet))) **(16)**,
are added to (S-1) ~ (S-6). Then, instead of state-transitioning all these specifications together, they are divided into two groups: (S-1)$\wedge$(S-3)$\wedge$(S-5)$\wedge$(S-6)$\wedge$**(15)** and (S-2)$\wedge$(S-4)$\wedge$**(16)**. The former controls 'initg' and 'wait', and the latter controls 'initd'. Each group is state-transitioned as (STEP2) respectively.

step of the synthesis flow	1	2	3	4	5	total
CPU time (seconds) on HITACHI M-280H C-Prolog	0.88	1.03	1.68	0.36	0.82	4.77

Table 3 Timings for the Synthesis of 'Sequencer Controller' in fig.2

(3) (STEP3) is executed to reduce the redundancy of the two state-diagrams obtained from (2).
(4) Again, (STEP2) is executed to make the global state-diagram for the two state-diagrams obtained from (3).
(5) Again, (STEP3) is executed to reduce the redundancy of the state-diagram.

The execution times for the synthesis flow above are shown in table 3. The state-diagram synthesized has 8 states, and so it requires 3 flip-flops. As compared to this, in the manual design, 'sequencer controller' is divided into two parts, because the state-diagram for all the 'sequencer controller' is too complex for a designer to manage. Each is designed with 3 states (i.e. 2 flip-flops), and so the manual design for all the 'sequencer controller' has 4 flip-flops. Seen from table 3, the required time is fairly short, and the results are satisfactory as compared to the manual design.

6 DISCUSSIONS

Both the synthesis time and the quality of the state-diagram synthesized in section 5 are satisfactory. As a rule, the required time for the original decision procedure (Wolper and Manna 1981, Clarke and Emerson 1981) grows exponentially with the number of temporal operators. However, the incremental synthesis method proposed in section 4 grows polynominally both with the number of variables and with the number of states in the synthesized state-diagram.

Specifications are assumed to be expressed in the form of **(14)**. Let Ni and No be the numbers of input and output variables respectively. As seen from section 2.2,

$n \propto No$

If the global state-diagram is synthesized by adding a expression one by one, the number of repeat of (STEP2) and (STEP3), Np, is

$Np \propto n \propto No$

Let Ns be the number of states for the final state-diagram synthesized. Since the number of states of the state-diagrams is not considered to exceed several times of Ns during the incremental synthesis procedure, the required time for one cycle of (STEP2) and (STEP3), Tc, is considered to be

$$Tc \propto Ns^{2*}(Ni+No)$$

Then, the synthesis time, Ts, is

$$Ts \propto Tc*Np \propto Ns^{2*}(Ni+No)*No$$

Therefore, the synthesis time grows polynominally both with the number of variables and with the number of states of the state-diagram synthesized.

7 CONCLUSIONS

We have presented the efficient method for automatic synthesis of state-diagrams from temporal logic specifications, and we have also shows it has much practical power by applying it to real hardwares and comparing to the manual design. As the required time for the original procedure increases exponentially with the number of temporal operators (Wolper and Manna 1981, Clarke and Emerson 1981), we have presented the efficient method for implementing the procedure with Prolog. The method regards state-diagrams already synthesized as a knowledge and synthesizes the global state-diagrams incrementally. The synthesis time is drastically reduced (to the polynominal order) and is kept small enough with fairy large hardwares.

The synthesis method translates any temporal logic expressions to state-diagrams, so it is also possible to check whether or not hardware designs in temporal logic really satisfies some other temporal logic expressions. Moreover, together with the verification method for gate and state-diagrams in (Fujita et al. 1983a, 1983b, 1984a), any designs in temporal logic, state-diagrams, and gates can be verified in the same way, and then the hierarchical design is smoothly supported.

The powerful pattern matching and automatic backtracking mechanisms of Prolog make the implementations much easier and simpler. It is concluded that together with the verification method in (Fujita et al. 1983a, 1983b, 1984a), a powerful CAD system for logic design, which smoothly supports hierarchical designs and has enough practical power, can be constructed with Prolog.

Finally, we are studying on a hardware description language which is based upon intervals and Linear Time Temporal Logic (Manna and Pnueli 1981, Wolper 1981). Its results will be appeared elsewhere.

REFERENCES

Fujita, M., Tanaka, H., and Moto-oka T: " Verification with Prolog and Temporal Logic", IFIP 6th Computer Hardware Description Languages and their Applications, Pittsburgh, May 1983a.

Fujita, M., Tanaka, H., and Moto-oka, T.: "Temporal Logic Based Hardware Description and its Verification with Prolog", New Generation Com-

puting, Vol.1 No.2, Ohmsha and Springer-Verlag, New York, 1983b.

Fujita, M., Nishiyama, S., Tanaka, H., and Moto-oka, T.: "Efficient Verification Methods for Hardware Logic Design and their Implementation with Prolog", Proc. of the Logic Programming Conference '84, Tokyo, March, 1984a.

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

Wolper, P.: "Temporal Logic Can Be More Expressive", 22nd Annual Symposium on Foundation of Computer Science, October 1981.

Wolper, P. and Manna, Z.: "Synthesis of Communicating Processes from Temporal Logic Specifications", Proc. of Logics of Programs, New York, May 1981.

Clarke, E.M. and Emerson, E.A.: "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic", i.b.i.d.

Clocksin, W.F. and Mellish, C.S.: "Programming in Prolog", Springer-Verlag, New York, 1981.

Moto-oka, T., Tanaka, H., Aida, H., Hirata, K., and Maruyama, T.: "The Architecture of a Parallel Inference Engine —**PIE**—", Proc. of FGCS'84, Tokyo, 1984.

Yuhara, M., Koike, H., Tanaka H., and Moto-oka, T.: "A Unify Processor Pilot Machine for **PIE**", Proc. of the Logic Programming Conference '84, Tokyo, Japan, March 1984.

Fujita, M., Tanaka H., and Moto-oka, T.: "Specifying Synchronization Parts of Hardware with Temporal Logic and Automatic Synthesis of State-Diagrams", Technical Research Report, EC83-59, IECE of Japan, 1984b (Japanese).

Pereira, F.: "C-Prolog User's Manual Version 1.5", EdCAD, Edinburgh University, Edinburgh, February, 1984.

Moszkowski, B.: "A Temporal Logic for Multi-Level Reasoning about Hardware", IFIP 6th Computer Hardware Description Languages and their Applications, Pittsburgh, May 1983.

Fujita, M., Tanaka H., and Moto-oka, T.: "Hardware Functional Description Based upon Temporal Logic and its Interpreter", Technical Research Report, EC84-23, IECE of Japan, 1984c (Japanese).