Computer Hardware Description Languages
and their Applications
T. Uehara and M. Barbacci (Editors)
North-Holland Publishing Company

Verification with Prolog and Temporal Logic

Masahiro Fujita, Hidehiko Tanaka, Tohru Moto-oka
Graduate Course of Information Engineering,
University of Tokyo
7-3-1 Hongo Bunkyoku Tokyo, Japan 113

ABSTRACT- Verification techniques of gate- and DDL-level designs with Prolog and temporal logic are presented by the handshaking example. Assertions (specifications) are made by temporal logic and this makes it possible to express state sequences (e.g. timing charts). Designs in both gate- and DDL-level are converted into the relation between the present state and next state in Prolog, and are verified state by state. Prolog has very powerful pattern matching and automatic backtracking facilities allowing easy to write programs which answer temporal logic assertions using forward reasoning and backward reasoning. We conclude that a total verification system handling various design levels - specification to gate - can be constructed with temporal logic and Prolog.

1. Introduction

The growing complexity of systems and escalating costs for implementing engineering changes to products are calling for tools and methods to detect design errors as early as possible in the hardware development process.

We are proposing a CAD system for complex hardware. Its characteristics are:

(1) Handles structured modules; a module is composed of several submodules.
(2) Uses temporal logic descriptions for specifying hardware modules.
(3) Verifies the logic of a module and its component submodules at every design cycle.
(4) Automatically constructs small modules based on the temporal logic descriptions specified.
(5) Uses Prolog/KR programming.

As a system becomes more complex, hierarchical designs become increasingly imperative, and a formal method for specifying modules is required to verify descriptions at every design stage of hierarchical designs. Thus temporal logic has been proposed as an extension to traditional logic and predicate calculus, which is convenient for specifying all possible system states at a given time. Temporal logic can add possible state sequences resulting from system evolution.

Instead of conventional timing charts, temporal logic provides timing relations to ease precise descriptions of hardware modules. The relational operators are: 'eventually', 'always', 'next', and 'until'.

Hardware verifications using several methods have been studied, including the software verification technique and DDL verifier [1]. But the former is difficult to make verification conditions or assertions, and the latter cannot handle temporal sequences. Based on those ideas, we are presenting a new approach to hardware verification and automatic synthesis by using temporal logic and its decision procedure modified from [2]. As temporal logic can express temporal sequences, several works on logic verification using temporal logic have been performed [11]. But these are theoretical approaches. We are introducing a new approach to construct a CAD system for high-level design using temporal logic and Prolog/KR [6,7]. The temporal logic specifications of a module are used for assertions of divided submodules or module design verifications. For small modules, temporal logic specifications are input to the automatic synthesis program. The CAD system can verify gate, DDL [3] descriptions, temporal logic specifications, their combined descriptions, and can automatically synthesize DDL descriptions from temporal logic specifications for small modules.

The CAD system is installed on Prolog/KR. Prolog has very powerful pattern matching and automatic backtracking capabilities which are useful for examing all the state transition paths. Prolog/KR, which is an extension of Prolog, has several functions available to aide in the construction of knowledge-based systems.

This paper introduces the basic ideas and techniques about temporal logic specifications, and gate- and DDL-level verifications in the CAD system. Verification of temporal logic levels and automatic synthesis are discussed later.

Chapter2 presents hardware specification concepts using temporal logic. Chapter3 is an overview of the CAD system, concentrating on hierarchical designs. Chapter4 shows techniques of describing gate- and DDL-level designs in Prolog/KR. Chapter5 presents the verification algorithm and the verifier program in Prolog/KR including the synthesis of verifier programs. Chapter6 presents a verification example and the last chapter has concluding remarks.

2. Hardware specifications with temporal logic

2.1. Temporal logic

This section introduces temporal logic which is used to describe hardware specifications. A more precise discussion of this topic may be found in [2,5].

While traditional logic uses such operators as $\vee$, $\wedge$, $\sim$, $\supset$, etc., temporal logic introduces additional operators for dealing with temporal sequences. While an expression of traditional predicate calculus is assumed to specify properties of the system state at a given time, called the 'present' time, an expression of temporal logic is assumed to specify properties of all possible execution sequences that may evolve from the present system state.

Propositional temporal logic is propositional logic extended with four 'temporal' operators, $\Box$, ∇, $\bigcirc$, and U. The first three are unary operators, and the last is a binary operator. Intuitively, for a sequence, $\Box$F is true if F is true in all future states of that sequence; ∇F is true if F is true in some future state; $\bigcirc$F is true if F is true in the next state in the sequences, and F1 U F2 is true if F1 is true for all states until the first state where F2 is true.

Temporal operators may be combined. For example, the assertion

$B \supset \Box\nabla A$

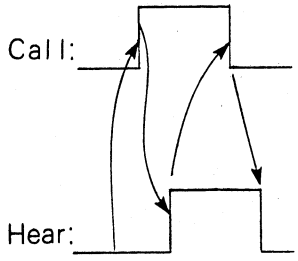
means that if B is true at present,then at every future point in time, A will be true at some point following that point and, thus, A will be true infinitely often, possibly without change.

2.2. Timing chart

Temporal logic can describe timing charts expressing timing relations among hardware modules. For example, the rising signal P is expressed as $\uparrow P = \sim P \wedge \bigcirc P$ and the falling signal P as $\downarrow P = P \wedge \bigcirc \sim P$.

Handshaking sequence is shown in fig.1 [11,12]. Call is a request signal from a calling module to a called module and Hear is a response. The timing chart (a) says that if Hear is low, then Call rises; if Call rises, Hear rises; if Hear rises, Call falls; and if Call falls, Hear falls. This relations are converted into temporal logic as (b). $\Box(\sim Hear \supset \nabla Call)$ and $\Box(\sim Call \supset \sim Call\ U \sim Hear)$ show that Call rises if and only if Hear is low. The until operator is used for specifying that Call and Hear are treated as registers.

As seen above, temporal logic can express various timing relations and hardware specifications.



(a) timing chart

$\Box(\sim Hear \supset \nabla Call)$
$\Box(Hear \supset \nabla \sim Call)$
$\Box(\sim Call \supset \sim Call\ U \sim Hear)$
$\Box(Call \supset Call\ U\ Hear)$
Peter (Sender)

$\Box(Call \supset \nabla Hear)$
$\Box(\sim Call \supset \nabla \sim Hear)$
$\Box(\sim Hear \supset \sim Hear\ U\ Call)$
$\Box(Hear \supset Hear\ U \sim Call)$
Paul (Receiver)

(b) temporal logic

fig.1 handshaking sequence between modul-Peter and Paul

2.3. Specification based on temporal logic

A digital system interact with its environments in some way, usually by performing a sequence of actions (e.g., passing data and control signals back and forth through shared registers and buses). Thus the behaviors of the system can be expressed with all possible sequences of events of data input/output through external terminals. At some point in time, an event is to read a value from terminal (input), to write a value to terminal (output), or to do nothing (null). The descriptions of events of all external terminals decide the behavior at that time, and the descriptions of all possible sequences of events decide the entire behavior of the system.

On the other hand, a system can be divided into two parts: synchronization and function. The function part concerns actual calculation processing (e.g., ALU in CPU), and the synchronization part concerns controlling interface timing relations among modules.

We now introduce a specification language describing the synchronization part with temporal logic and the function part as input-output relation tables. For example, the data transfer system is specified as in fig.2. The data in Infout are sent to Infin through the terminal Message using handshaking sequence shown in fig.1. The declaration <register> and <terminal> have the same meaning as in DDL [3]. I(Hear) represents the input of a value '1' from the register Hear, and O(~Call) represents the output of a value '0' to the register Call. In this way, input and output are clearly separated for system specifications. Systems containing the function part are described as follows:

$I(X=x) \wedge I(Y=y) \supset \bigcirc O(Z=fun(x,y))$

and 'fun' is given with an input-output relation table.

```
<system> Data-Transfer;
  <register> Call, Hear, Infout, Infin;
  <terminal> Message;
  <module> Peter;
     <spec> I(~Hear) ⊃ ∇ O(Call),
        I(Hear) ⊃ ∇ O(~Call),
        I(Call)∧I(Infout = m) ⊃ O(Message = m);
  <end> Peter;
  <module> Paul;
     <spec> I(Call) ⊃ ∇ O(Hear),
        I(~Call) ⊃ ∇ O(~Hear),
        I(Call)∧I(Hear)∧I(Message = m) ⊃ ○O(Infin = m);
  <end> Paul;
<end> Data-Transfer;
```

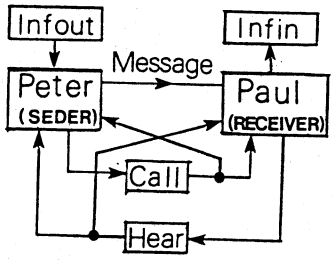


fig.2 specification of the data transfer system

Though the assertions for verification should be distinguished between I (read) and O (write), our system can currently handle only the form shown in fig.1, that is, there is no distinction between I and O.

3. CAD system overview

We are planning the total CAD system for hardware design -- specification, design or synthesis, and verification. Specifications are written with temporal logic (see chapter2) and design outputs with DDL-[3] or gate-level descriptions. Systems are designed hierarchically. Modules are specified and designed with lower-level submodules. The verification is to insure that divided submodules satisfy the upper-level module's specifications.

One design cycle in hierarchical design is:

(1)Specifies with temporal logic.

(2)Divides into and specifies submodules.

or

Designs with DDL or gate level descriptions

(3)Verifies that submodule specifications, or DDL- (or gate-level) submodule descriptions satisfy the upper-level module's specification

This cycle is repeated until all submodules are small enough to design with DDL or primitive gates easily or to be able to be synthesized automatically. When all modules

are written with DDL or gate-level descriptions, those descriptions are sent to another lower-level CAD system such as shown in [4].

Here is an example of hardware specification and hierarchical design. It is the data transfer system in chapter2.

Fig.3 is the hierarchical specification of the system. Top-level specification means that if adequate initial conditions are satisfied, then data in Infout are sent to Infin, and initial conditions are infinitely satisfied. At the time of specifying the 2nd-level design, we use the handshaking knowledge, and then get the specifications (designs) as seen in fig.2.

The 3rd-level system design is the DDL-level descriptions of module-Peter and Paul. Both are shown in fig.4.

If necessary, the gate-level designs are made in the 4th-level design.

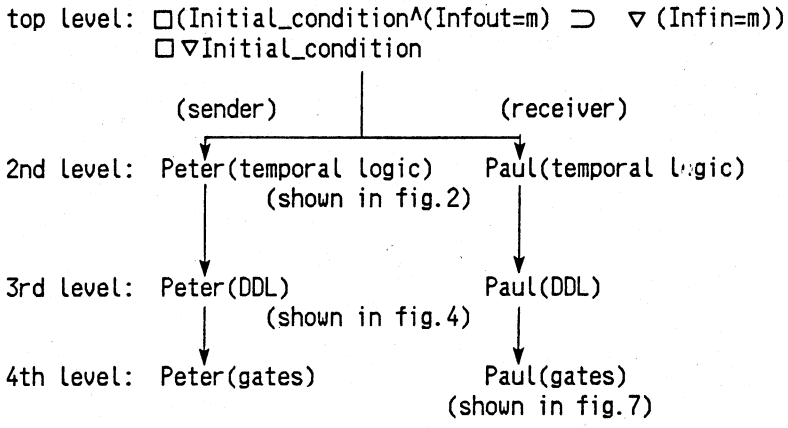


fig.3 hierarchical design of the data transfer system using handshaking sequence

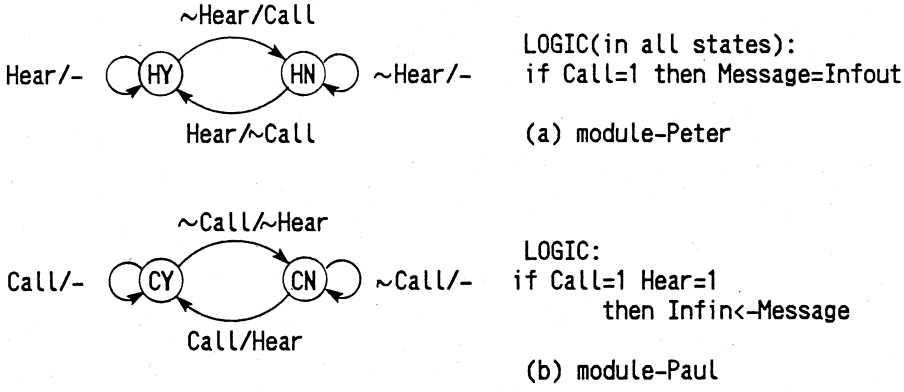


fig.4 DDL level design of module-Peter and Paul

Verifications are made by assertions and corresponding answers. The assertions are upper-level specifications.

All programs supporting this CAD system are written in Prolog/KR (see chapter4). The rest of this paper presents basic ideas and techniques of the CAD system in gate- and DDL-level designs.

4. Gate, DDL level descriptions with Prolog/KR

4.1. Prolog/KR and its expression for gate level circuits

Prolog/KR [6] is an interactive programming system designed to support programs manipulating symbols and structures, especially AI programs embedding knowledge (including control knowledge); KR stands for Knowledge Representation. Prolog/KR is currently being developed at the Univ. of Tokyo.

Prolog/KR differs from other Prolog systems in control structures and list expressions. In representing knowledge, a procedural way and declarative way are often distinguished, but this is a matter of degree. In Prolog/KR, both ways can be used. Prolog/KR can be used as a procedural language as well as Lisp by providing each control explicitly. In such a case, no automatic backtracking occurs and every computation is deterministic. If no controls are provided, automatic backtracking is performed and the computation is non-deterministic. In normal cases, programming is done using a mixture of these two extremes.

Prolog/KR's objects (including programs) are expressed as lists or atoms. Thus, every object except variables follows Lisp's syntax (UTILISP [7]: developed at the Univ. of Tokyo). Variables are atoms beginning with '*'.

Some Prolog/KR programs for gate-level description are presented. We describe these programs in two ways - popular Prolog [8] and Prolog/KR to make the difference clear.

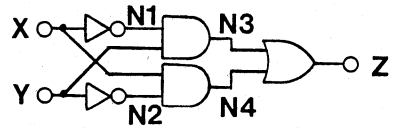
We treat two values (0,1), which are easily extended to more values. Primitive gates -- AND, NOT etc., are easily described in the input-output relations (fig.5). The last value of the table is the output value and others are input values. We assume that combinational circuits have no time delay.

The last example in fig.5 is an EXclusive-OR gate which is composed of NOT, AND and OR gates. In Prolog/KR, the first element of the list is a goal and the rest is the condition. Thus, a more complex gate can be constructed using the same variables as connections among gates.

```
(ASSERT (FAND 0 0 0))
(ASSERT (FAND 0 1 0))
(ASSERT (FAND 1 0 0))
(ASSERT (FAND 1 1 1))
(ASSERT (FNOT 0 1))
(ASSERT (FNOT 1 0))
(ASSERT (FEOR *X *Y *Z)
        (FNOT *X *N1) (FNOT *Y *N2)
        (FAND *N1 *Y *N3)
        (FAND *N2 *X *N4)
        (FOR *N3 *N4 *Z))
```

(a) Prolog/KR

```
fand(0 0 0).
fand(0 1 0).
fand(1 0 0).
fand(1 1 1).
fnot(0 1).
fnot(1 0).
feor(X,Y,Z):-fnot(X,N1),fnot(Y,N2),
             fand(N1,Y,N3),fand(N2,X,N4),
             for(N3,N4,Z).
```



(b) popular Prolog

fig.5 primitive gates descriptions in Prolog

Flip-flops are described with the relations between the present state and next state. For example, D-flip-flop is shown in fig.6. The first argument is D-input, the next two are the present internal state (and its negation), the next two are the next internal state (and its negation), and the last is the clock signal. The first ASSERT shows if the internal state is reset and the clock signal has not arrived (clock=0), then the next internal state is reset regardless of the D-input ('*' means 'don't care').

```
(ASSERT (FDFF * 0 1 0 1 0))
(ASSERT (FDFF 1 0 1 1 0 1))
(ASSERT (FDFF 0 0 1 0 1 1))
(ASSERT (FDFF * 1 0 1 0 0))
(ASSERT (FDFF 1 1 0 1 0 1))
(ASSERT (FDFF 0 1 0 0 1 1))
```

D-input Q ~Q Qn ~Qn clock

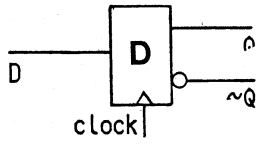
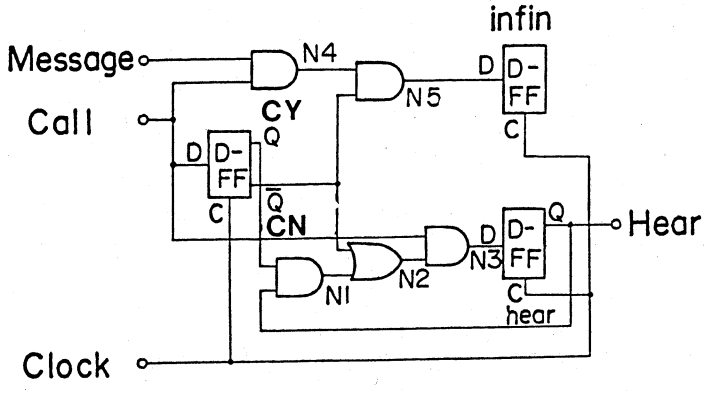


fig.6 D-flipflop in Prolog/KR

Finally we show an example of sequential gate level circuit: module-Paul of the data transfer system in chapter3. Fig.7(a) is an implementation example of module-Paul with primitive gates and is translated into the Prolog/KR program:(b). Variable names which begin with '@' have the next state values and those with no '@' have the present state values.

Present state values and next state values are connected using the D-flipflop table (FDFF in Prolog/KR). Thus, gate-level circuits are converted to the relations between the present and next state in Prolog/KR.



(a) circuit diagram

```
(ASSERT (PAULC (*MESSAGE *CALL *HEAR *INFIN *CY *CN)
               (*@MESSAGE *@CALL *@HEAR *@INFIN *@CY *@CN)
               *CL)
        (FAND *MESSAGE *CALL *N4) (FAND *N4 *CN *N5)
        (FDFF *N5 *INFIN *~INFIN *@INFIN *@~INFIN *CL)
        (FDFF *CALL *CY *CN *@CY *@CN *CL)
        (FAND *CY *HEAR *N1) (FOR *N1 *CN *N2) (FAND *CALL *N2 *3)
        (FDFF *N3 *HEAR *~HEAR *@HEAR *@~HEAR *CL))
```

(b) in Prolog/KR

fig.7 a gate-level design of module-Paul

```
(SYSTEM DATA-TRANSFER
  (DCL (CONTROL-REGISTER   (CALL HEAR))
       (DATA-REGISTER      (INFOUT INFIN))
       (DATA-TERMINAL      (MESSAGE))
       (STATE-NAME         (HY HN CY CN)))
  (AUTOMATON PETER
       (LOGIC (IF (= CALL 1) (= MESSAGE INFOUT)))
       (HY
          (IF (= HEAR 1) (-> HY) (DO (<- CALL 1) (-> HN))))
       (HN
          (IF (= HEAR 0) (-> HN) (DO (<- CALL 0) (-> HY)))))
  (AUTOMATON PAUL
       (LOGIC (IF (AND (= CALL 1) (= HEAR 1)) (<- INFIN MESSAGE)))
       (CN
          (IF (= CALL 0) (-> CN) (DO (<- HEAR 1) (-> CY))))
       (CY
          (IF (= CALL 1) (-> CY) (DO (<- HEAR 0) (-> CN))))))
```

fig.8 the data transfer system in DDL-S

```
(ASSERT (PETER (*STATE *CALL *HEAR *INFOUT *MESSAGE)
               (*@STATE *@CALL *@HEAR *@INFOUT *@MESSAGE)
               *CL)
      (IF (EQ *CL 0)
          (TRUE)
          (AND (OR (AND (= *STATE HY)
                        (OR (AND (FEQ *HEAR 1) (= *@STATE HY))
                            (AND (FEQ *HEAR 0) (TRAN *@CALL 1) (= *@STATE HN))))
                   (AND (= *STATE HN)
                        (OR (AND (FEQ *HEAR 0) (= *@STATE HN))
                            (AND (FEQ *HEAR 1) (TRAN *@CALL 0) (= *@STATE HY)))))
               (OR (AND (FEQ *CALL 1) (TRAN *MESSAGE *INFOUT)) (FEQ *CALL 0))))))
```

fig.9 DDL-S level design of module-Peter in Prolog/KR

```
(ASSERT (DATA-TRANSFER ((*PETER *PAUL) *CALL *HEAR *INFOUT *INFIN)
                       ((*@PETER *@PAUL) *@CALL *@HEAR *@INFOUT *@INFIN))
     (PETER (*PETER *CALL *HEAR *INFOUT *MESSAGE)
            (*@PETER *@CALL *@HEAR *@INFOUT *@MESSAGE)
            1)
     (PAUL (*PAUL *CALL *HEAR *INFIN *MESSAGE)
           (*@PAUL *@CALL *@HEAR *@INFIN *@MESSAGE)
           1)
     (REGUNCHANGED (*CALL *HEAR *INFOUT *INFIN) (*@CALL *@HEAR *@INFOUT *@INFIN))
```

fig.10 DDL-S level design of the data transfer system in Prolog/KR

4.2. DDL-level expression in Prolog/KR

Like in gate-level, DDL descriptions are translated into the relations between the present and next state in Prolog. As we concentrate on synchronization parts of the system, the subset of DDL (DDL-S) is sufficient. We do not show the syntax of DDL-S but show an example. Fig.8 is the DDL-S descriptions of the data transfer system and is equivalent to fig.4. DDL-S has two types of variables: control and data. The important restrictions are that a control variable is a single bit and a data variable can not appear in conditions. This simplifies translations of DDL-S descriptions into Prolog/KR.

Every DDL-S automaton is translated into Prolog/KR, for example, module-Peter of the data transfer system is translated into fig.9.

Each module are described to do nothing ((TRUE) in Prolog/KR) when the clock has not arrived (*CL=0). FEQ in fig.9 is a equality check program between two variables. TRAN is a program for data transfer. The data transfer system (fig.8) in Prolog/KR is shown in fig.10, using Peter (fig.9) and Paul. We assume that the two modules work with the same clock. Instead of gate-level, the internal expression (not shown in fig.9, 10) of every variable of DDL-S is set to the form (*val *c): *val is the actual value of the variable and *c is additional information for data transfer. At every clock cycle, if *c is instanciated to 'atom', the variable is data-transferred, and if not, it has not been data-transferred yet. This makes it very easy to handle variables as registers, that is, any register has the value of the past state unless data-transferred. This is realized by the system predicate REGUNCHANGED.

5. Method of verification

5.1. Forward reasoning and backward reasoning

Assertions in temporal logic can be verified using forward reasoning and backward reasoning [1] in temporal sequences. In both ways, verification step is divided into two parts.

(1) Gets Prolog/KR program for the system from informations concerning circuit connections or DDL-S descriptions (see cahpter4).

(2) Verifies by means of a reduction to absurdity.

The verifier program for forward reasoning is easily acquired from the temporal logic assertions using the temporal logic decision procedure [2] (see next section), but that for backward reasoning must be written manually for each assertion at present and much the same as in [13]. Therefore, in this paper we discuss only forward reasoning. We explain the verifier program by example: $\Box(A \supset \nabla B)$ (A, B have no temporal operators). $\Box(A \supset \nabla B)$ is verified through the verifications for all cases of $(A \supset \nabla B)$. Using a reduction to absurdity, we get negation first of all;

$\sim(A \supset \nabla B) = (A \wedge \Box \sim B)$ ($\because \sim \nabla = \Box \sim$ by axiom)

Then we check whether there is a state transition path satisfying $(A \wedge \Box \sim B)$, and if any, then print it.

In order to investigate all paths, we begin with state INIT satisfying the condition A and not B, and repeat the following until all paths are found.

Look up the state transition table in Prolog and get the next state N.

{(STTRAN * *) in Prolog/KR}

If N satisfies B, then this path is OK and backtrack for another path.

{(LOGIC B *N)}

If the state transition path is in a loop and every state in the path does not satisfy B, then this path satisfies $A \wedge \Box \sim B$ and is a counter example, so print this path (history) and backtrack (FALSE) for another path.

{(IF (STEQU *H *P) (AND (PRINT (*H TYPE<> *P))(FALSE)) (recursive call))}

This reasoning program is shown in fig.11 with Prolog/KR using explicit controls. The check whether the current state is the same as the past one (there is a loop) is done by investigating if all of the internal states (e.g. flip-flops, state-names in DDL etc.) and external variables have the same values as the past values. When there is an error or a loop, compulsory backtracking ((FALSE) in Prolog/KR) occurs to investigate all paths. The Prolog automatic backtracking facility makes programs very simple.

```
(ASSERT (VER *INIT) (LOGICA INIT) (LOGIC~B *INIT) (STTRAN *INIT *N)
                    (LOGIC~B *N) (VER1 (*INIT) *N))
(ASSERT (VER1 *H *P)
      (IF (STEQU *H *P)
          (AND (PRINT (*H TYPE<> *P)) (FALSE))
          (AND (STTRAN *P *N) (LOGIC~B *N) (VER1 (*P . *H) *N))))
```

fig.11 forward reasoning program for $\Box(A \supset \nabla B)$

5.2. Synthesis of verifier from temporal logic assertions

Verifier programs for forward reasoning are easily acquired from temporal logic assertions using the temporal logic decision procedure [2]. For example, the verifier of the last section is acquired in the following manner:

Since negation of the assertion is $(A \wedge \Box \sim B)$, we start with $(A \wedge \Box \sim B)$. According to the decision procedure, $\Box P$ is decomposed into P and $\bigcirc P$, that is, $\Box P$ means that P should currently be satisfied and that $\Box P$ should also be satisfied in the next state. Therefore, $A \wedge \Box \sim B$ require that the current state satisfies $A \wedge \sim B$ and that the next state satisfies $\Box \sim B$. So if $\sim A$ or $A \wedge B$ is satisfied in the current state, $A \wedge \Box \sim B$ can not be satisfied, and that case is not a counter example. If $A \wedge \sim B$ is satisfied, the condition for the next state is $\Box \sim B$. Then $\Box \sim B$ is decomposed in the next step.

This decomposition continues until all paths are in a loop (conditions are the same as the past ones). Finally, we get the state diagram for negation of the assertion $(A \supset \nabla B)$ (fig.12). It is very easy to get verifier programs in Prolog/KR (as in fig.11) from this diagram. Each node of fig.12 corresponds to each Prolog predicate, and each arc corresponds to the relation between calling and called predicates. The loop check program is added to the predicate containing a loop path. Thus, the forward reasoning programs can be automatically synthesized from temporal logic assertions. The automatic synthesis of backward reasoning programs is much harder, but is possible.

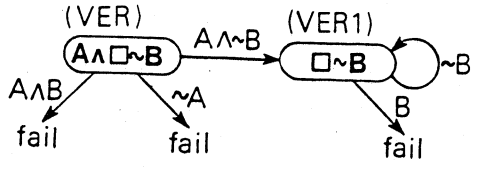


fig.12 state diagram for $\sim(A \supset \nabla B)$

6. Example

In this chapter a gate-level verification example, whose assertion is one of the upper-level temporal logic specifications, is presented. It is to check whether the circuit of module-Paul (fig.7) satisfies one of the Paul's temporal logic specifications: □(Call ⊃ ▽Hear) (shown in fig.13). Since negation of the assertion is (Call∧□~Hear) we begin with Call=1. Every state is expressed with the list of the values for all variables at that time. As condition A in section5.1 of this case is Call, the predicate LOGICA is th check whether the element for variable Call of the current state list is '1' or not, and as the condition B is ~Hear, the predicate LOGIC_B is to check whether the element for variable Hear is '0' or not. As seen in figure, there are counter examples for this assertion. Therefore CY=0 (CN=1) is initially required to satisfy this assertion.

```
(ASSERT (STTRAN *P *N) (PAULC *P *N 1))          ; Verifying module-Paul
(ASSERT (LOGICA (*MESSAGE 1 *HEAR *INFIN *CY *CN))   ; Call=1
(ASSERT (LOGIC B (*MESSAGE *CALL 0 *INFIN *CY *CN))  ; Hear=0
                        Infin=0
:(ver (*message *call *hear 0 *cy *cn))          ; Start of Verification
(((0 1 0 0 1 0)) TYPE<>N (*@M_3894 *@CALL_3894 0 0 1 0))  ; Counter
(((1 1 0 0 1 0)) TYPE<>N (*@M_3894 *@CALL_3894 0 0 1 0))  ;   examples
NIL
```

fig.13 verification example of gate-level : □(Call⊃▽Hear)

The execution time of verification for the data transfer system in various levels is about 30~300ms in the HITACHI M-200H (10 MIPS). If the number of external variables is fixed, the execution time increases in proportion to the number of gates or the number of states in DDL. Though the execution time increases exponential with the number of necessary state transitions needed for verification, the number of necessary state transitions is usually less than 10 in a rather large system from experiences. Consequently, our approach is very practical.

7. Conclusion

Verification techniques of gate- and DDL-level designs are introduced using the handshaking example. Gate- or DDL- designs are expressed with the table between the present and next state in Prolog/KR. Assertions (specifications) are made by temporal logic and this makes it possible to express state sequences (e.g. timing charts).

Prolog has very powerful pattern matching and automatic backtracking capabilities enabling easy to write programs which answer temporal logic assertions using forward and backward reasonings.

Verifier programs for forward reasoning are easily acquired from temporal logic assertions using the temporal logic decision procedure.

Our conclusion is that a total verification system handling various design level - specification to gates - can be constructed using temporal logic and Prolog.

Acknowledgement

The authors would like to thank N. Kawato, T. Saito, and H. Katoh of Fujitsu Lab. for their helpful suggestions and encouragemants. The authors would also like to thank T. Uehara for his meaningful advices.

References

[1] T. Uehara, F. Maruyama, T. Saito, N. Kawato: 'DDL Verifier', 5th Computer Hardware Description Language and their Applications, September 1981.

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

[3] J. R. Duley, D. L. Dietmeyer: 'A Digital System Design Language (DDL)', IEEE Trans. Computer, Vol.C-17, pp 850-861 1968.

[4] T. Uehara, F. Maruyama, T. Saito, N. Kawato: 'An Interactive Logic Synthesis System Based Upon AI Technique', 19th Design Automation Conference, June 1982.

[5] A. Pnueli, S. Shelah, J. Stavi: 'On the Temporal Analysis of Fairness', 7th Annual Symposium on Principle of Programming Languages.

[6] H. Nakashima: 'Prolog/KR User's Manual', Faculty of Engineering, Univ. of Tokyo, Tech. Rep. METR82-4, March 1982.

[7] T. Chikayama: 'UTILISP Manual', Faculty of Engineering, Univ. of Tokyo, Tech. Rep. METR81-6, September 1981.

[8] W. F. Clocksin, C. S. Mellish: 'Programming in Prolog', Springer-Verlag, 1981.

[9] N. Kawato, T. Saito, F. Maruyama, T. Uehara: 'Design and Verification of Large-Scale Computers by using DDL', 16th Design Automation Conference, June 1979.

[10] B. T. Hailpern: 'Verifying Concurrent Processes Using Temporal Logic', Dep. Computer Science, Stanford Univ., August 1980.

[11] G. V. Bochmanm: 'Hardware Specification with Temporal Logic: An Example', IEEE Trans. on Computer, Vol.C-31, No.3, pp 223-231 1982.

[12] G. A. Blaauw: 'Digital System Implementation', Prentice-Hall 1978.

[13] T. Uehara, T. Saito, F. Maruyama, N. Kawato: 'DDL Verifier and Temporal Logic', 6th Computer Hardware Description Languages and their Applications, May 1983.