Computer Hardware Description Languages
and their Applications, C.J. Koomen and T. Moto-oka (eds.)
Elsevier Science Publishers B.V. (North-Holland)

Logic Design Assistance with Temporal Logic

Masahiro FUJITA*, Hidehiko TANAKA, Tohru MOTO-OKA

Department of Electrical Engineering
the University of Tokyo
Tokyo, Japan

Abstract

We have already proposed new verification and synthesis methods for hardware logic design with temporal logic and Prolog, which supports hierarchical design consistently [1,2,3]. Specifications are written in temporal logic, which is an extension to traditional logic and can easily describe timing relations among variables. The verifier for gate and state diagram designs is implemented using Prolog [1,3]. An efficient method for synthesizing state diagrams was also presented in [2]. However, the verification method in [1] is rather basic idea and is not applicable to large real hardwares. In this paper, specification, verification and synthesis systems for hardware logic design, implemented in C-Prolog [7] and applicable to very large hardware designs, are presented [3]. Specifications are described in the conjunction of simple temporal logic expressions, which drastically reduces the required time for verification and synthesis. And the efficient techniques for verification and their implementations in Prolog as experimental systems are shown.

1. Introduction

In recent years, computer systems become larger and more complicated, and they have become a very essential part in our lives. This means hardware designs must have much reliability and one can not make logic designs without computer assistance. 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 new verification and synthesis methods for hardware logic design with temporal logic and Prolog, which supports hierarchical design consistently [1,2,3]. Specifications are written in temporal logic, which is an extension to traditional logic and can easily describe timing relations among variables. The verifier for gate and state diagram designs is implemented using Prolog [1,3]. Also an efficient method for synthesizing state diagrams was presented in [2]. However, the verification method in [1] is a rather basic idea and can not be applied to large real hardwares. In this paper, a complete specification, verification and synthesis system for hardware logic design, which is applicable to fairly large hardwares, is presented [3].

We divide hardware systems into two parts: function parts, which actually execute logic and arithmetic functions, and synchronization parts, which control timings for data transfer among function units. A synchronization part is specified in propositional temporal logic and design of it is automatically verified or synthesized.

In this paper we concentrate on assistance for the design of a synchronization part. First the techniques for specifying hardwares, which reduce the required time for verification and synthesis drastically, are shown. Second the methods for automatic verification of synchronization parts are reviewed. We also present several methods for increasing the efficiency of verification, which enables us to handle large and complex system designs. And then an efficient synthesis method from temporal logic specifications is referenced. Finally the unified logic design assistant systems using those techniques in C-Prolog are presented.

2. Specification in Temporal Logic

2.1. Synchronization Parts and Function Parts

As a rule, a system can be divided into two parts: a function part, which actually executes logical and arithmetic operations, and a synchronization part, which controls timings for data transfers among function units. For example, consider the data transfer system by handshaking sequences (fig.1) [1], where the data path from the register Infout to the register Infin is a function part, and the circuit generating the control signals (Call and Hear) for handshaking sequences is a synchronization part.

In designing a function part, the most important problems are whether calculations of functions are terminated within the time specified using given inputs. As compared with this, designs of synchronization parts have the problems whether the specified data are sent on the specified time. In designing a synchronization part, a designer must manage the whole circuit working in parallel and may easily make a mistake. Therefore, a synchronization part should be automatically verified or synthesized. This paper presents efficient verification and synthesis methods for synchronization parts.

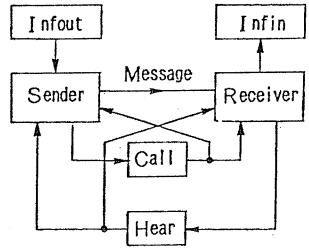


Fig.1 A Data-Transfer System by Handshaking Sequences

2.2. Specification of Synchronization Parts in Temporal Logic

In this section we first briefly introduce temporal logic (we use here Linear Time Temporal Logic [4]), and then show techniques for specifying hardware in it, by which we can efficiently verify or synthesize designs. The detailed discussion about temporal logic can be found in [4,5].

Linear Time Temporal Logic (LTTL) is an extension of the 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 (with no temporal operators): P is true at present
@P: P is true on the next time (in sequential circuits, next clock),
[]: P is true on the present and all the future times,
<>: P is true at least on a time in the present and future,
P U Q: P is true on all the times until the first time where Q is true.

Temporal logic can describe temporal sequences among variables and therefore can express timing relations shown usually in timing diagrams.

'If the signal P is active, then the signal Q is active on the next time' is described as

```
    [](P -> @ Q)    (1)
```

(->:IMPLY ~:NOT ,:AND /\:AND ',' and '/\' both means logical AND)

(1) guarantees 'If P is active, then Q is active', but P may be active otherwise. If it is desired 'P becomes active if and only if on the next time when Q is ac-

[](~Q -> ((@~Q) U P))) (2)

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

For example, the relationship shown in fig.2: '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

[](S -> ((P -> @Q) U E)) (3)

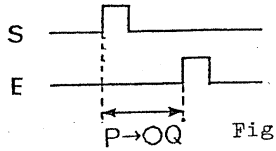


Fig.2

As seen from (3), a complex specification requires temporal operators nested many times. However, if an interval I is defined as 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, then the specification (3) is described with the logical AND of following simple expressions.

[](~I -> ((~I) U S)),
[](S -> I),
[](I -> (I U E)), (4)
[](E -> (~I)),
[]((I /\ P) -> @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 (4), complex specifications can be described with the conjunction of simple temporal logic expressions. Moreover, it is easy to specify sequentiality in L/TTL using interval signals. Not only declarative but also procedural descriptions must be necessary to smoothly specify behaviors of hardwares. Procedural descriptions have two aspects: parallel and sequential. It is easy to describe parallelism in temporal logic. Parallelism is described by the conjunction of each action. For example, 'the two actions: P -> @Q and R -> @S are working in parallel' is described as

P -> @Q,
R -> @S. (5)

On the other hand, it is tedious and not easy to describe sequentialities, e.g., 'first execute P and then execute Q'. However, if we use some signals expressing whether P and Q are 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. Let Ip and Iq be interval signals respectively expressing whether P and Q are executed or not, that is,

[](Ip <-> P), (A<->B = A->B/\B->A)
[](Iq <-> Q). (6)

Then, the relationship 'first execute P and then execute Q' is described in the conjunction of simple expressions:

Ip,
~Iq,
[](~Ip -> Iq), (7)
[](~Iq -> ~Iq U ~Ip),

Thee first two lines of (7) mean that initially Ip is active and Iq is inactive, and the next two lines mean that if Ip becomes inactive, then Iq becomes active. Note that the basic ideas of Ip and Iq are the same as in Interval Temporal Logic by Moszkowski [6]. However, we describe them only in Linear Time Temporal Logic for ease mechanical handlings.

Interval signals are additional variables and indicate internal states of the hardware specified, and therefore, they are not necessarily observable from 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 the conjunction of simple temporal logic expressions, which drastical-

3. Hardware Descriptions in Prolog

In this section we explain how to describe hardware in Prolog The syntax of Prolog used here is that of C-Prolog [7]. C-Prolog is developed at EdCAD of Edinburgh University.

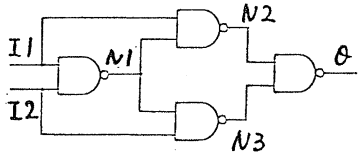
The primitive combinatorial gates, such as, NAND, AND, NOT, etc., are described in Prolog as facts corresponding to the truth tables. For example, a NAND gate is described as below:

```
fnand([1,1,0]).
fnand([0,X,1]).
fnand([0,0,1]).
```

(We treat two values (0 and 1), though it is easily extended to handle more values. We also assume combinational circuits have no time delay.). The last element in the list of each line means the output value of that gate, and the other elements mean the input values. 'X' means 'don't care'.

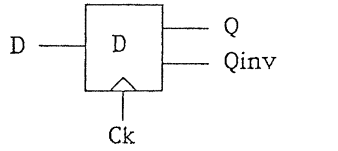
Networks are described by using the same variable on the terminals of the same net. As the results of the pattern matching mechanism of Prolog, the terminals of the same net have the same values. For example, the circuit (EXCLUSIVE-OR gate) is shown below:

```
feor([I1,I2,O]):- fnand([I1,I2,N1]),
                  fnand([I1,N1,N2]),
                  fnand([N1,I2,N3]),
                  fnand([N2,N3,O]).
```

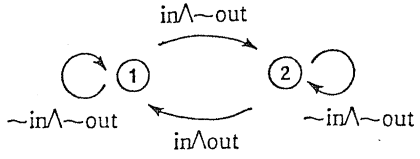


As for the sequential circuits, besides the input and output terminals, the present internal state and the next internal state are added to the arguments of the facts. For example, D-flip-flop is shown below:

```
dff([D,0,Q,Qinv],Q,Q):- fnot([Q,Qinv]).
dff([D,1,Q,Qinv],Q,D):- fnot([Q,Qinv]).
fnot([0,1]).
fnot([1,0]).
```



The first argument of 'fdff' contains the D-input, the clock signal, the output (Q), and its negation (Qinv). The last two arguments are the present internal state and the next internal state. Also, fnot([Q,Qinv]) means that Q is the negation of Qinv. Therefore, the first line means that if the present clock signal is 0, the next internal state is the same as the present internal state. Any flip-flops can be described in the same way.



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

(b) Prolog Description for (a)

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

(c) Global State Diagram in Prolog

Fig.3 State Diagram Description in Prolog

Any synchronous circuits are described in the same way as above. Moreover,

state, and the next state, and so, they are described in Prolog with the table of those values like flip-flops or sequential circuits. For example, a state diagram, shown in fig.3(a), is described in Prolog as in (b). 'logic' is a predicate expressing the conditions for state transitions. The first argument of 'logic' (i.e., a, b, c, d) are names of state transitions and the second is a variable containing a list of each variable's value (i.e., in, out). Therefore, 'logic(a,[0,0])' means that the condition for the state transition 'a' is ~in/\~out and corresponds to the state transition from state '1' to the state '1'.

The global state diagram of the given state Diagrams is easily acquired with Prolog. For example, the global state diagram for the two state diagrams, 'state_diagram1' and 'state_diagram2', is described in Prolog as shown in fig.3(c).

Finally we show the translation method of temporal logic expressions to Prolog. The method is base upon the temporal logic decision procedure [3,5]. Using the temporal operator expansion rules, any temporal logic expressions can be expanded to state diagram having so called 'eventuality' (explained later).

The expansion process 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 remaining 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.

<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})

The expansion rules for each temporal logic operator are shown above. This is acquired from the temporal logic axioms or the temporal operators' definitions and, <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 the symbol {P} following ~P/\@<>P of <2> indicates that.

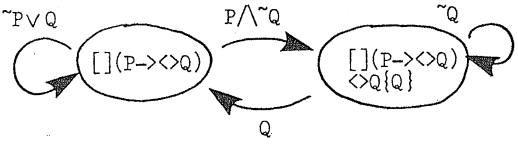
Using this table, any temporal logic expressions can be expanded into the conditions at present and the ones in the next time. Differing from state diagrams, these conditions may have eventualities as described above.

The expansion process of [](P -> <>Q) is shown in fig.4 as an example. First, [](P -> <>Q) is expanded as follows, referring to <1> and <2> of the above table. As [](P -> <>Q)/\<>Q{Q} appears as the conditions for the next time, it is expanded next in a similar way. All the conditions required for the next time are already treated. So we get the state diagram with eventuality as shown in fig.4(b). State diagrams with eventualities are expressed in Prolog with five arguments: present variable list, present state, next state, present eventualities and next eventualities. Each eventuality is assigned the value 0 or 1 indicating whether the eventuality is satisfied or not. So, the state diagram for [](P -> <>Q) with eventuality is described in Prolog as shown in fig.4(c).

```
[](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})

[](P -> <>Q)/\<>Q{Q}
 =(Q/\@[](P -> <>Q))
\/(~Q/\@([](P -> <>Q)/\<>Q{Q}))
```

(a) Expansion Process



(b) State Diagram with Eventuality for [](P-><>Q)

```
sample(1,1,[0],[0],P):- logic(1,P).
sample(1,2,[0],[1],P):- logic(2,P).
sample(2,1,[1],[0],P):- logic(3,P).
sample(2,2,[1],[1],P):- logic(4,P).
```

```
%          P Q
logic(1,[0,0]).
logic(1,[P,1]).
logic(2,[1,0]).
logic(3,[P,1]).
logic(4,[P,0]).
```

(c) Prolog Description for (b)

Fig.4 Expansion of [](P-><>Q)

As seen above, any descriptions in gates, state diagrams or temporal logic can be described in Prolog. We can ask various questions to those Prolog descriptions about the relationships between the arguments of those, that is, the ones between the present and the next state, etc. For example, if we want to know the two input values of the EXCLUSIVE-OR gate (see above) provided that the output value is 1, we ask:

```
?- feor([I1,I2,1]).
```

The system answers I1=1 and I2=0 and the execution successfully halts with a single answer. If we want to know all the answers for that condition, we ask:

```
?- feor([I1,I2,1]), print([I1,I2]), fail.
```

In the above, 'fail' is a system predicate causing compulsorily backtrackings. Using the Prolog's automatic backtracking mechanism, we can easily investigate all the cases for given conditions, which is very useful for verification and synthesis.

4. Verification of Synchronization Parts

4.1. The Verification Method

We verify that the hardware descriptions in Prolog really satisfy specifications. Let S be the temporal logic expression for a specification and D be the temporal logic expression for hardware designed (any designs can be expressed in temporal logic, because they are described with the relations between the present and the next as seen above). So we must investigate the next formula:

```
[](D -> S).      (8)
```

We verify (8) by showing that the negation of

```
D -> S           (9)
```

gives us a contradiction for all cases. The negation of (9) is

```
D/\~S.           (10)
```

(10) shows us that it is enough that in order to verify (9), the cases satisfying

the negation of S must be checked not to satisfy D.

As seen in the previous section, any temporal logic expressions can be described in state diagrams (with eventuality) and also in Prolog. So, in order to check that (10) is unsatisfiable, we have only to do the following.

(1) make state diagrams (with eventuality) for S and D, and also make Prolog descriptions for the global state diagram for them (that is, a state diagram expressing D/\~S).
(2) check whether there is any infinite state transition path for that global state diagram by state-transitioning on it. If it exists (that is, D/\~S is satisfied), it is printed out as a counter example (one cycle of the loop is printed out), and if not (that is, D/\~S is not satisfied), the design is correct with respect to the specification S.

Note that since specifications are expressed in the conjunction of simple temporal logic expressions, it is enough to check those expressions one by one.

The translation method for is already explained in section 3. We can execute (2) by two ways: forward or backward in temporal sequences. The forward reasoning checks the global state diagram on the direction of state transitions, and the backward reasoning checks it on the reverse direction of state transitions. The forward reasoning procedure in Prolog is as follows.

(i) Start with the initial state, get the next state of the global state diagram (the state diagram expressing D/\~S)), and repeat getting the next state of the global state diagram until the same state (the state of the state diagram of the negation of the specification and the states of the design description) as the past one is appeared and the state transition path falls into a loop.
(ii) When there is a loop, check whether the loop satisfies all eventualities (if any). If eventualities are not satisfied, make compulsory backtrack to examine another path. And if satisfied, this path is a counter example and printed out.

In (i) and (ii) above, if a state transition path of the design descriptions does not satisfy the negation of the specification, another path is taken by the Prolog's automatic backtracking mechanism. The verifier in Prolog for the above procedure is shown in fig.5. 'H' in the predicate 'verify_forward' contains the history of the state transition path currently examined by the program. Note that the global state diagram (D/\~S) is automatically acquired from the design and the negation of specifications (see section 6).

```
verify_forward([[Pstate_design,Pstate_spec],[Peve_design,Peve_spec]],H):-
    negation_spec(Pter,Pstate_spec,Nstate_spec,Peve_spec,Neve_spec),
    design(Pter,Pstate_design,Nstate_design,Peve_design,Neve_design),
    (member([[Pstate_design,Pstate_spec],[Peve_design,Peve_spec]],H)
     -> (check_eventuality([[Pstate_design,Pstate_spec],
                            [Peve_design,Peve_spec]],H)
         -> (print('error'),
             print([[[Pstate_design,Pstate_spec],[Peve_design,Peve_spec]]|H]))
         ; fail )
     verify_forward([[Nstate_design,Nstate_spec],[Neve_design,Neve_spec]],
                    [[[Pstate_design,Pstate_spec],[Peve_design,Peve_spec]]|H])).
```

Fig.5 Forward Verification Program in Prolog

On the other hand, the backward reasoning procedure traces back the procedure above.

(i) Start with any state, repeat getting the last state of the global state diagram until the same state as the future one is appeared and the state transition path falls into a loop.
When there is a loop, check whether the loop satisfies all eventualities (if any). If eventualities are not satisfied, make compulsory backtrack to examine another path and go to (i). And if satisfied, go to (ii).
(ii) Again, repeat getting the last state of the global state diagram, and check whether the state transition path can reach the initial state of the state diagram of the negation of the specification. If it can not, backtrack to examine another path. And if it can, this path is a counter example and is printed out.

The backward reasoning verifier is easily programmed in Prolog. The details are found in [3].

4.2. Increasing the Efficiency of Verification

The complexity of calculations of the method in section 4.1 is considered to be NP-complete, and the total time needed for verification grows exponentially with the scale of circuits. Therefore, some methods for increasing the efficiency of verification are required to keep the verification time of practical size circuits small enough. In [3], thinking of verifications of gate level designs, three methods for increasing the efficiency are given. They are:

- filtering of a design description and getting the part of it truly needed for the verification,
- memorizing state transitions already treated in order not to treat the same state transition twice,
- utilizing specifications for external environments in order to decrease the number of cases for verification.

We here explain the most important one, filtering of a design description, in the following.

As seen in section 2, a specification can be expressed in the conjunction of simple temporal logic expressions. The verifier checks these simple expressions one by one, and each expression has only a few external output terminals. Starting with these external output terminals and tracing back from outputs to inputs along the networks to be verified, we can extract the part of the logic networks truly influencing the specification, that is, truly needed for the verification. We here name this procedure 'filtering of a design description'.

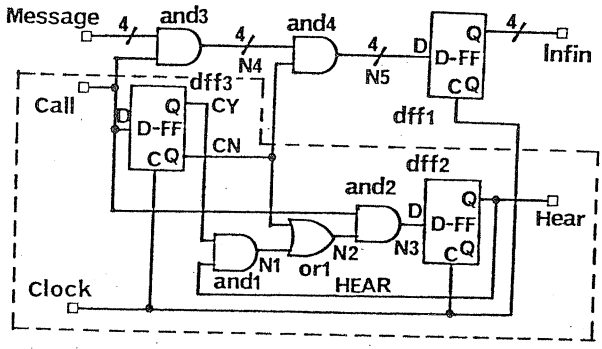


Fig.6 Receiver of the Date-Transfer System of fig.1

For example, the entire circuit of the receiver by handshaking [1] is shown in fig.6. The result of filtering by the specification:

[](Call -> <> Hear) (11)

is the interior surrounded by dotted lines. (Hear is the only external output terminal and so the filtering procedure starts with Hear.) Filtering algorithm in Prolog is shown in [3].

As mentioned above, a system can be divided into the synchronization part and the function part. The verifier treats only synchronization parts, and the part of the design descriptions truly needed for the verification of a specification is a part of the synchronization part and much smaller than the whole descriptions. From our experiences the size of filtered circuits is less than 100 modules. Therefore, the filtering of a design description decreases the verification time very much and keeps it manageably small.

5. Synthesis of State Diagram

We have already developed an efficient method for synthesizing state diagrams from temporal logic specifications [2,3]. The method regards already synthesized state diagrams as knowledge, and the state diagram for a specification is incrementally synthesized using that knowledge. We omit to explain it for space limit. Please see [2,3] for the details.

6. Logic Design Assistance Systems

So far we have shown specification, verification and synthesis techniques using temporal logic and Prolog. Those are implemented in C-Prolog as a logic design verification system and an automatic synthesis system.

The verification system in Prolog is shown in fig.7. Designs at gate level are described in structural hardware description language HSL [9] (almost the same as SDL [10]), and designs in state diagrams are described in DDL-S [3] (subset of DDL [8]), and those are transformed into Prolog descriptions by HSL-translator and DDL-S-translator respectively. Specifications in temporal logic, which are written in the form of lists, are expanded into the state diagrams in Prolog descriptions by the TL-translator. The filtering program extracts the truly needed part from the whole design data base and then transforms it into the descriptions for the verifier.

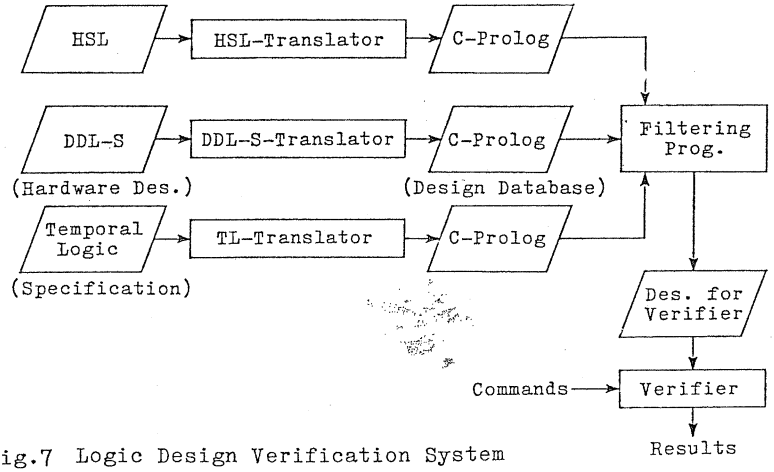


Fig.7 Logic Design Verification System

The verifier program accepts design descriptions to be verified and the state

diagram of the negation of the specification in Prolog, and state transitions both of them together (that is, the global state diagram) according to the methods shown in section 4.1. In verifying a design, if some counter examples are found, the design is incorrect, and so the state transition path are printed out as counter examples. If not, the design is proved to satisfy the specification. Currently, the analysis of counter examples is done manually. However, the mechanical assistance in the analysis is possible, and we are now studying on it. The program size is about 2000 steps.

The details of the automatic synthesis system are found in [2,3]. The program size is also about 2000 steps.

7. Conclusions

Specification, verification and synthesis techniques of synchronization parts with temporal logic are presented. These method are implemented in C-Prolog and are applied to real hardwares. The results are satisfactory in execution time and quality.

Introducing interval signals, not only parallelism but also sequentiality are smoothly described in the conjunction of simple temporal logic expressions, which drastically reduces the time for verification and synthesis.

The time needed for verifying synchronization parts of hardwares is considered to grow exponentially with the scale of it in the worst case. However, 'filtering of a design description' keeps the size of the circuit to be really examined manageably small even with a large hardware. As specifications are described in the conjunction of simple expressions, the size of the circuit filtered by that simple expression is considered to be less than 100 gates from experiences [3]. And from our verification results [3], circuits under 100 gates (modules) can be verified within a few minutes on a main frame computer. Therefore, it is concluded that systems of well hierarchical designs, even if they are very large ones, can be verified within several minutes.

References

[1] M. Fujita, H. Tanaka and T. Moto-oka, "Verification with Prolog and Temporal Logic", IFIP 6th Computer Hardware Description Languages and their Applications, May 1983.
[2] M. Fujita, H. Tanaka and T. Moto-oka, "Specifying Hardware in Temporal Logic & Efficient Synthesis of State-Diagrams Using Prolog", Proc. of FGCS '84, Tokyo Japan, November 1984.
[3] M. Fujita, "Logic Design Assistance with Temporal Logic", Doctoral Dieertation, Information Engineering, University of Tokyo, 1984.
[4] Z. Manna and A. Pnueli, "Verification of Concurrent Programs, Part 1: The Temporal Framework", Dept. of Computer Science, Stanford Univ. Report STAN-CS-81-836, June 1981.
[5] P. Wolper, "Temporal Logic Can Be More Expressive", 22nd Annual Symposium on Foundation of Computer Science, October 1981.
[6] B. Moszkowski, "A Temporal Logic for Multi-Level Reasoning about Hardware", IFIP 6th Computer Hardware Description Languages and their Applications, May 1983.
[7] F. Pereira, "C-Prolog Users Manual Version 1.2a"
[8] J.R. Juley and D.L. Dietmeyer, "A Digital System Design Language (DDL)", IEEE Trans. on Computer, Vol.C-17, No.9, pp850-861, September 1968.
[9] Y. Sugiyama, O. Karatsu and T. Sudo, "VLSL Design System", Monograph of Thechnical Group on Design Technology of Electronics Equipment of IPSJ, 7-2, December 1980 (Japanese).
[10] W.M. VanCleemput, "A Hierarchical Language for the Structural Description of Digital Systems", ACM IEEE 14th DA Conference, June 1977.