VLSI '87, C.H. Séquin (editor)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1988

Temporal Logic Based Fast Verification System Using Cover Expressions

Hiroshi Nakamura **Masahiro Fujita*** **Shinji Kono**
Hidehiko Tanaka

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

* FUJITSU LABORATORIES LTD.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

We have developed a verification and synthesis method for hardware logic designs specified by temporal logic using Prolog, but this system was not satisfactory from the view point of speed and memory. Hence, we have implemented another verification system using the C language, where the combinational circuit part is handled in sum-of-product form (cover expressions).

While the time required for the verification in both systems are nearly equal in the cases of small designs, the larger the scale of design is, the more it takes to verify in the Prolog-version system (increases almost exponentially). The C-version system can handle much larger designs in comparison, and it has successfully verified a DMA controller about 1000 times faster than the Prolog-version system.

1 Introduction

In the past several years there has been an increasing interest in verifying, as distinct from simply testing, the proposed logic designs [2,6].

We have developed a verification and synthesis method for hardware logic designs specified by temporal logic with Prolog [4], but this system was unsatisfactory due to its deficiency in speed and memory. Hence, we have implemented another verification system using the C language. This system verifies the synchronous circuits of the synchronization part in the digital systems. In this system, the combinational circuit part is handled in sum-of-product form. Since the undefined values of the variables are directly expressed in this form, the stage of backtracking is reduced and the number of times required to trace the combinational circuit is also reduced.

In this system, we used **Tokio**[9] as a specification description language, which is based on temporal logic, and which enables us to describe a specification in any levels[7].

In this paper, we present the method of verification of hardware logic circuits using temporal logic. We will show the efficiency of this system implemented with the C language, and will compare the results with those implemented with Prolog.

1.1 contents

In the following sections, we discuss the following topics:

Section 2 The structure of the system.

Section 3 Verification based on temporal logic.

Section 4 Verification method using cover expressions.

Section 5 Application and evaluation of the system.

Section 6 Verification method using terminal variables.

Section 7 Conclusions.

2 The Structure of the Verification System

The structure of the verification system is as shown in Figure 1. This system verifies the synchronous circuits of the synchronization part in the digital systems. The synchronization part is generally small enough to be treated in sum-of-product form. The parts of translating **Tokio** into Linear Time Temporal Logic (**LTTL**; see section 3) and **LTTL** into state diagrams are implemented with Prolog [4]. In this paper, we will not discuss the method of translating **Tokio** into **LTTL**. The basic idea of the verification in this system (see section 3) is generally the same as that in the Prolog-version system. HSL is a hardware description language which only describes the networks among the gates.

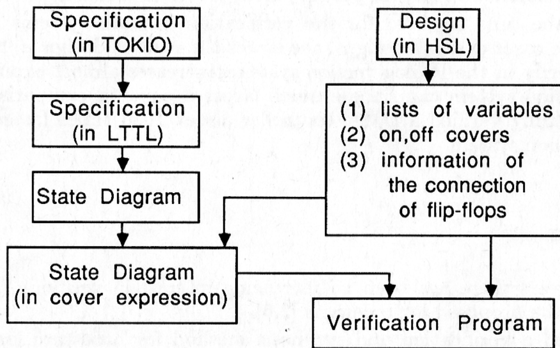

Figure 1: Structure of the Verification System

3 Temporal Logic and State Diagram

In this section, we first briefly introduce temporal logic, then describe the method of translating it into state diagrams, and finally explain the verification method using state diagrams.

3.1 Specifications in Temporal Logic

There are several kinds of temporal logic; here we use Linear Time Temporal Logic (**LTTL**)[11], which is an extension of the traditional logic with four temporal operators added, that is, $\circ$ (next), $\Box$ (always), $\Diamond$ (sometimes), and U (until). **LTTL** is defined not on continuous states but on discrete states. The first three operators are unary and the last is a binary operator. The meaning of each temporal operator is as follows.

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

LTTL can express a wide variety of properties of sequences, which make it easy to describe the specifications of the hardware. For instance,

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

is described as

$\square(P \rightarrow \circ\, Q)$.

(From now on, "$A \rightarrow B$" means that "if A holds true then B must be true".)

3.2 Translation into State Diagrams

Next, we describe the technique to translate **LTTL** formulas into state diagrams. The basic idea of this technique is that **LTTL** formula can be decomposed into sets containing formulas which are either atomic (that is, without temporal operators) or that have $\circ$ as their main operator. The atomic sets are transition conditions and the rest excluding the outermost $\circ$ operator are conditions in the next state (details are described in [10]). The decompositions are repeated until every condition in the next states produced during the decompositions are the same as those conditions already treated.

The decomposition rules are as follows.

- $\square F = F \wedge \circ\square F$
- $\diamond F = F \vee (\sim F \wedge \circ\diamond F\{F\})$
- $F1\ U\ F2 = F2 \vee (F1 \wedge \sim F2 \wedge \circ(F1\ U\ F2))$

(From now on, "$\sim$" represents negative.)

For example, let P, Q and R are atomic and we want to translate

(A) $\square P$

(B) $\sim((P \wedge \circ\square Q) \rightarrow \circ\square R)$

into state diagrams using above rules. It goes as follows;

(A) $\square P = P \wedge \circ\underline{\square P}$

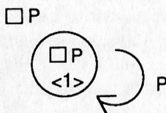

Figure 2: State Diagram (1)

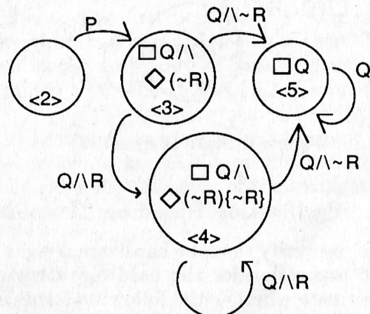

Figure 3: State Diagram (2)

Since the condition in the next state '$\Box P$' (underlined) is the same as the condition at the current state, the decomposition is completed and the corresponding state diagram is obtained as shown in Figure 2.

The translation of (B) goes;

$$\textbf{(B)} \sim ((P \wedge \circ\Box Q) \rightarrow \circ\Box R)$$
$$= (P \wedge \circ\Box Q \wedge \sim (\circ\Box R))$$
$$= (P \wedge \circ\Box Q \wedge \circ\Diamond(\sim R))$$
$$= (P \wedge \circ\underline{(\Box Q \wedge \Diamond(\sim R))}$$

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

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

Therefore, the corresponding state diagram is as shown in Figure 3.

Satisfiability A **LTTL** formula is satisfiable iff it has at least one infinite sequence of state transitions when it is translated into a state diagram. The logic formula (A) is satisfiable because it has an infinite sequence of state transitions $< 1 >, < 1 >, \cdots$. Similarly, the logic formula (B) is satisfiable because it has an infinite sequence of state transitions $< 5 >, < 5 >, \cdots$. Obviously, an infinite sequence is nothing but a loop. Here, the sequence $< 4 >, < 4 >, \cdots$ is not infinite because it does not satisfy the eventuality $\{\sim R\}$. The eventuality $\{P\}$ means that P must eventually be true in all sequences of future states which follow the state $\{P\}$. Since '$\sim R$' is never true in the sequence $< 4 >, < 4 >, \cdots$, this sequence cannot be infinite.

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

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

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

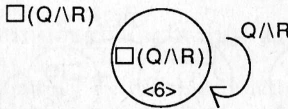

Figure 4: State Diagram (3)

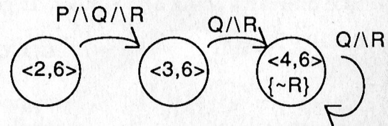

Figure 5: State Diagram (4)

3.3 Verification Based on Temporal Logic

Here, we verify that the hardware designs satisfy the specifications. Let D be the temporal logic expression for the hardware design and S be that for the specification. We must investigate whether the following formula;

$$\mathrm{D} \rightarrow \mathrm{S}$$

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

$$D \wedge \sim S$$

is unsatisfiable. In order to check this, we only need to do the following operations;

- Make state diagrams for $\sim$ S and D.
- Check whether there is any loop for both state diagrams $\sim$ S and D.

If there exists an infinite sequence, the design does not satisfy the specification (contradiction), and if it does not exist, the design is correct with respect to the specification S.

3.4 Implementation on Prolog

Here, we describe the method of obtaining the state diagram for the design on the Prolog-version system. This is where the difference between the Prolog-version system and C-version system is most obvious.

Since states for the design are nothing but the conditions of the flip-flops, in order to acquire the state diagram for the design, it is only necessary to trace all the gates and decide the next condition of the flip-flops provided that the current condition of them is given.

At first, the description concerning networks between gates is translated into Prolog. For example, AND-gate like in Figure 6 is described as

and2($[I1, I2, O1]$).

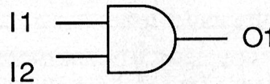

Figure 6: AND gate

There also exists a data base for the functional gates such as

and2([1,1,1]).
and2([A,0,0]).
⋮

and so on. Therefore, if the values of $I1$ and $I2$ are both 1, the value of $O1$ is unified to 1 by unification. The next condition of the flip-flops is only acquired when this operation is executed throughout all the gates.

Here, we must consider the case that the current input values are not fixed such as external inputs. For example, let us consider that $I1 = 1$, and the value of $I2$ is not fixed in Figure 6. In this case, the value of $O1$ is unified to 1 at first because the value of $I2$ is unified to 1 using a data base, and tracing the gates is continued. Even if there does not exist a loop in this case, since there still remains the possibility of contradiction, the verification backtracks and the value of $O1$ is unified to 0 (the value of $I2$ is unified to 0), and the verification continues.

Therefore, in this system, all the gates should be traced every time in obtaining the next condition of the flip-flops and there also exist many backtrackings, which lead to the degradation in the efficiency of the verification. To raise the efficiency of the verification, the number of backtrackings and tracing gates should be reduced. Thus, we suggest two approaches for the efficiency;

1. Use triple-valued logic and handle undefined value.
2. Trace the combinational part of the design only once.

In the next section, we show the more efficient verification system in which these two approaches are implemented.

4 Logic Design Verification using Cover Expressions

The synchronous circuits are divided into the combinational part and flip-flop part as in Figure 7.

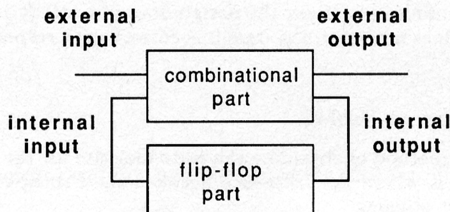

Figure 7: Structure of Synchronous Circuits

Here, we show the verification method by handling the combinational part as cover expressions [3].

At first, we briefly explain the cube and cover.

4.1 Cube and Cover

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

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

For example, let us consider a logic function with 4 inputs and 2 outputs. For $f_1 = x_1 x_2 \overline{x_4}$, we have $c = [01\ 01\ 11\ 10\ 1\ 0]$.

The input part of c is the subvector of c containing the first n entries of c. The output part of c is the subvector of c containing the last m entries of c. A variable corresponding to 11 in the input part is referred to as an input don't care, and 00 never appears in the input part.

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

$$f_1 = x_1 x_2 + \overline{x_2} x_3 + x_1 x_3;$$

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

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

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

Example.
$$\begin{array}{ll} f_1 = x_1x_2 & [01\ 01\ 11\ 1] \\ f_1 = x_2x_3 & [11\ 01\ 01\ 1] \\ \Downarrow & \Downarrow \\ f_1 = x_1x_2x_3 & [01\ 01\ 01\ 1] \end{array}$$

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

4.2 Verification Method using Cover Expressions

The verification flowchart using cover expressions is shown in Figure 8. (From now on, the state diagram corresponding to the Negation of the Specification is called **NS**.) We explain Figure 8 by verifying an example. The example is the control part of a receiver by handshaking [5]. The design is shown in Figure 9 and the specification to be verified is

$\Box$(Call $\rightarrow \Diamond$ Hear)

with the condition that flip-flops are reset at the initial state.

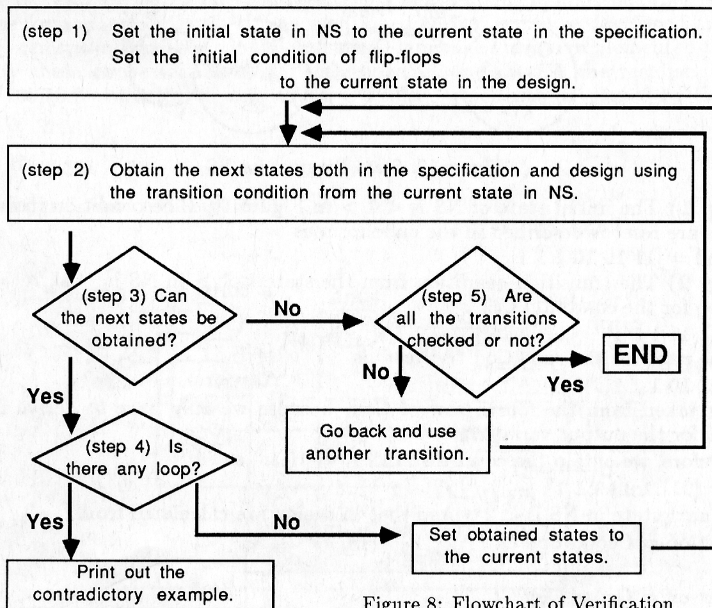

Figure 8: Flowchart of Verification

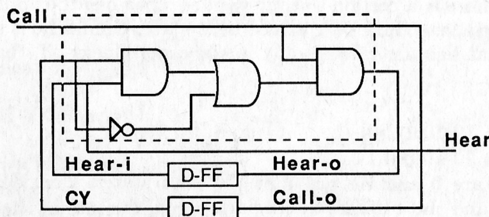

Figure 9: The Control Part of Receiver by Handshaking

(preparation) We describe each cube in the form of [Call, CY, Hear-i, Call-o, Hear-o, Hear] (input variables are Call, CY, and Hear-i; output variables are Call-o, Hear-o, and Hear). Then on-cover and off-cover of the combinational part of this circuit are as follows.

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

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

For example, the second and third rows of Con show that Hear-o is on in two cases, that is, either Call is on and CY is off or Call, CY, and Hear-i are all off.

The connections between flip-flops and the combinational part are described as $\circ$Hear-i = Hear-o and $\circ$CY = Call-o.

The negation of the specification "$\sim$ $\Box$(Call $\rightarrow$ $\Diamond$Hear)" is translated into a state diagram such as in Figure 10. This state diagram is nothing but **NS**.

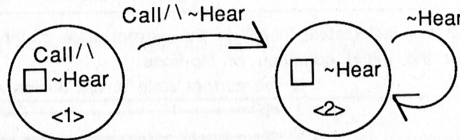

Figure 10: State Diagram for NS

(step 1) The initial state of NS is $< 1 >$ in Figure 10. The condition in which the flip-flops are reset is described in the cube form as

Ccond = [11 10 10 1 1 1].

(step 2) The transition condition from the state $< 1 >$ in NS is "Call $\wedge$ $\sim$ Hear". The cube for the condition Call is

[01 11 11 1 1 1].

Then the cube for the condition "$\sim$ Hear" is

[11 11 10 1 1 1],

which is taken from the fourth row of Coff, because we only have to derive from the off-cover for the output variable Hear.

Therefore, we obtain the cover for "Call $\wedge$ $\sim$ Hear" as

Ct = [01 11 10 1 1 1].

The next state in NS is $< 2 >$ and that in design are calculated from

Cnext-on = Ccond$\cdot$Con$\cdot$Ct

and

Cnext-off = Ccond$\cdot$Coff$\cdot$Ct.

(step 3) The next state in the design is obtained from Cnext-on and Cnext-off in the following way. If there is a certain output variable that has the value 1 only in the cover Cnext-on, that variable should be 1 at the next state. Similarly, if there exists a certain output variable that has the value 1 only in the cover Cnext-off, that variable should be 0 at the next state.

Here, since

Cnext-on = [01 10 10 1 1 0]

Cnext-off = [01 10 10 0 0 1],

Call-o and Hear-o are 0 and Hear is 1 at the next state. Considering the connections between flip-flops and the combinational part, the next state in the design is as follows.

Cnext = [11 01 01 1 1 1].

If there are some variables that have the value 1 both in Cnext-on and Cnext-off, the values for those variables at the next state can not be decided.

If there exists a variable that has the value 1 neither in Cnext-on nor Cnext-off, the next state in the design can not be obtained. In this case, verification flow goes to the (step 5) as shown in Figure 10.

(step 4) Check whether there is any transition loop both in NS and in the design. If there exists a loop and the loop satisfies eventuality, that is a contradictory example.

In this case, since there does not exist any loop, set $< 2 >$ to the current state in NS and set Cnext to the current state in the design (in other words, set Cnext to *new* Ccond), and go to the (step 2).

(step 2) The condition of the state transition in NS is
Ct = [11 11 10 1 1 1].

In this case, since Ct·Ccond = **nil**, both Cnext-on and Cnext-off are **nil**. and the next state in the design can not be obtained. Then go to the (step 5).

(step 5) Since there remains no state transitions in NS, verification has been finished.

5 Evaluation of Verification System

Here, we use two examples; one is a receiver by handshaking[5] and the other is a DMA controller for a mini computer[1]. As mentioned earlier, we have developed two verification systems; the Prolog-version[4] and the C-language version implemented this time. We verify these two examples using both systems, and discuss the results. We used VAX11/730 (0.2 ~ 0.3 MIPS) and UNIX C-Prolog, which was developed by Edinburgh University [8].

(1) Receiver

The design is shown in Figure 11.

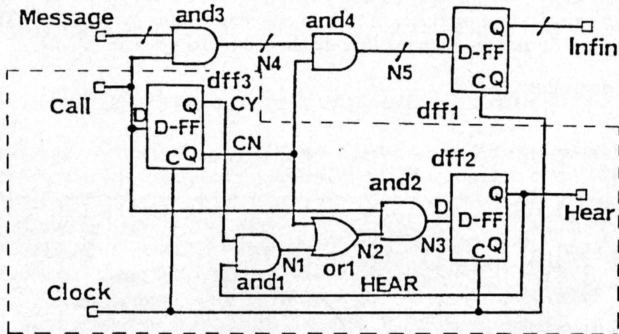

Figure 11: Receiver by Handshaking

We verify the specification

$\Box(\text{Reset} \rightarrow \Box(\text{Call} \rightarrow \Diamond\text{Hear}))$.

The results are shown in Table 1.

CPU time [sec]		Prolog version: verification part, memorizing states — without	Prolog version: verification part, memorizing states — with	C language version: HSL → cover	C language version: state diagram → cover	C language version: verification part without memorizing states
not filtered, bit width of data path	1bit	9.88	4.28	2.3 (2.1)	0.7	0.6
	2bits	204.80	28.25	⋮	⋮	⋮
	3bits	5645.80	253.3	⋮	(not measured)	(not measured)
	4bits	⋮	⋮	5.2 (3.3)	1.4	1.4
	8bits	(Keep on increasing)		10.2 (4.7)	2.9	3.1
	16bits	⋮	⋮	26.8 (8.4)	9.1	10.2
filtered		0.87	0.93	*	0.4	0.4

* In C version, the design is filtered in making covers, and so the time depends on the bit width of data path. The time for each width is shown above in (parentheses).

(VAX11/730 0.2~0.3 MIPS)

Table 1: CPU time for Verifying Receiver

Filtering Design Description One technique for restraining the increase of verification time is *filtering design description* which is to extract the necessary part for the verification. For example, as for this specification, there is only one output variable Hear and therefore, we only need to verify the filtered part about Hear (it is surrounded by the broken lines in Figure 11). This technique is implemented on both systems.

Memorization of states In the verification flowchart as in Figure 8, when we get a state that has appeared before, we need not check this state again provided that every state that has appeared is remembered. This is called *memorization of states.*

The basic operation for obtaining the next state is the bit-calculation in this C-version system whereas it is unification in the Prolog-version system. Hence, it takes much more time to obtain the next state in the Prolog-version system than in the C-version. This means that the memorization of states is not so efficient in this system as in the Prolog-version, therefore we do not implement this method on this system.

(2) DMA controller

Figure 12: DMA controller

The design is shown in Figure 12. We verify these two specifications.

(1) $\Box((\text{Reset} \wedge \circ \Box \sim \text{Reset} \wedge \Box \sim \text{Acdt}) \rightarrow \circ\Box(\text{Rqdma} \rightarrow \circ\ \text{Rqdt}))$

(2) $\Box((\text{Reset} \wedge \circ \Box \sim \text{Reset} \wedge \circ \Box \sim \text{Rqdma}) \rightarrow \circ\Box(\text{Acdt} \rightarrow \circ \sim \text{Rqdt}))$

Filtered part is dotted in Figure 12, and the results are shown in Table 2.

CPU time [sec]		Prolog version verification part memorizing states		C language version		
		without	with	HSL → cover	state diagram → cover	verification part without memorizing states
not filtered	(1)	> 60,000	> 60,000	22.5	4.5	7.9
	(2)	> 60,000	> 60,000	22.5	4.5	7.8
filtered	(1)	> 60,000	2672.05	18.6	2.1	2.8
	(2)	> 60,000	1923.35	18.6	2.1	2.7

("> 60,000" means over 60,000 seconds) (VAX11/730 0.2~0.3 MIPS)

Table 2: CPU time for Verifying DMA Controller

Evaluation What differs most between the two systems is the way the combinational part is handled. In the Prolog-version system, all gates are traced every time in obtaining the next state in the design, whereas those are traced once in making covers Con and Coff in the C-version system. Moreover, because the undefined values cannot be handled in Prolog, there exists in the Prolog-version system a lot of needless backtrackings which do not exist in the C-version system.

Therefore, while the time required for the verification in both systems are nearly equal in the cases of small the designs, the larger designs are, the longer it takes to verify in the Prolog-version system (increases almost exponentially). The C-version system can handle much larger designs in comparison, and it verified DMA controller about 1000 times faster than the Prolog-version system. Also, it takes little time to make covers.

6 Verification Method using Terminal Variables

In case the designs become larger and the number of input variables increases, it is not easy to translate the combinational part of the designs into cover expressions. The number of cubes is 2^{n-1} in the worst case where n be the number of input variables. The more complicated the logic of the combinational part becomes, the more it becomes the worst case. Although that part of the synchronization part is not usually so large and complicated as that of the function part, covers may explode.

In this section, we present a verification method using terminal variables which prevents covers from exploding. Introducing terminal variables makes it easy to translate the design into cover expressions, since the logic for output variables becomes simple. This method, however, is not implemented yet.

In using terminal variables, the structure of the synchronous circuits is as shown in Figure 13. Terminal variables should be derived from the original designs on condition that the terminal inputs and outputs have one-to-one correspondence.

The flowchart of the verification is very similar to what is mentioned in section 3. Different points happen where on-cover and off-cover of the combinational part are used. That is,

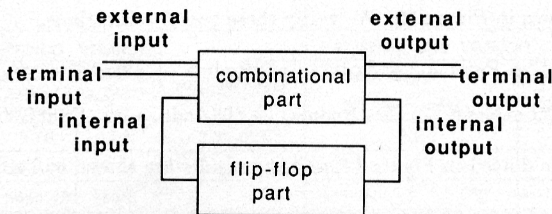

Figure 13: Structure of Synchronous Circuits using Terminal Variables

(1) translating a transition condition of the specification into cover expressions

(2) calculating the next state in the design.

(preparation) First, we get on-cover and off-cover of the combinational part of the circuit. The form of each cube is

[I,FI,TI,O,FO,TO].

Here, each column means as follows.

I : external input variables

FI : internal input variables

TI : terminal input variables

O : external output variables

FO : internal output variables

TO : terminal output variables

Input part consists of I, FI, and TI, and output part consists of O, FO, and TO.

(1) To obtain the cover Ct, which is the transition condition of NS. Ct has the form

[I,FI,TI,O,FO,TO],

and values of all the output variables are 1. Here, we remove terminal variables from the input part of Ct.

removal of terminal variables In case that a terminal input variable TI_i is 01 (10), the input part of Ct is replaced by that part of the intersection between Ct and some rows of on-cover (off-cover) which corresponds to TO_i — TI_i and TO_i correspond to each other one by one —, and then the value of TI_i is changed into 11. This operation should be repeated until all the values of TI are changed into 11.

If a certain input variable, that is I, FI, and TI, is changed into 00 during this operation, that means "the cube is **nil**" and this cube should be deleted. If the cover Ct becomes **nil** during this operation, this means that this transition condition cannot be satisfied and another transition should be found.

(2) We remove the terminal variables in the same way from the input part of Cnext-on and Cnext-off, which are obtained as described in section 3.

The rest of the verification flowchart are the same as mentioned earlier.

7 Conclusions

We have presented the verification method using cover expressions. The verification system where this method is implemented can verify larger designs and it has verified a DMA controller about 1000 times faster than the Prolog-version system. This is due to handling the combinational part in cover expressions.

We have also presented the verification method using terminal variables. We intend to implement this method and to show the efficiency of this method in the case of much larger designs.

References

[1] *User Device Design Manual for PANAFACOM U-series.*

[2] H.G. Barrow. Verify: a program for proving correctness of digital hardware designs. *Artif,Intel.*, 24:437–492, 1984.

[3] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovannni-Vincentelli. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers.

[4] M. Fujita. Logic Design Assistance with Temporal Logic. In *CHDL '85*, pages 129–138, IFIP, 1985.

[5] M. Fujita. Verification with Prolog and Temporal Logic. In *CHDL '83*, IFIP, 1983.

[6] M.J.C. Gordon. *Why higher-order logic is a good formalism for specifying and verifying hardware.* Technical Report 77, Cambridge University, 1985.

[7] M. Fujita and S. Kono and H. Tanaka. Aid to hierarchical and structured logic design using temporal logic and Prolog. In *Prodeedings.Pt.E*, pages 283–294, IEE, 1986.

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

[9] S. Kono and T. Aoyagi and M. Fujita and H. Tanaka. Implementation of temporal logic programming language Tokio. In *Logic Programming Conference '85*, pages 138–147, ICOT, 1985.

[10] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* Technical Report STAN-CS-82-925, Stanford University, 1982.

[11] Z. Manna and A. Pnueli. *Verification of Concurrent Programs Part1. The Temporal Framework.* Technical Report STAN-CS-81-836, Stanford University, 1981.