# Hardware Verification at Functional Design Stage

FUMIHIRO MARUYAMA*

With the increasing use of LSIs in computers, there has been a greater demand for high reliability in logic design. In order to respond to this demand, we have introduced formal design description into the functional design stage by means of the hardware description language DDL, and have developed a simulator and a translator, which extracts and arranges the information for circuit design from the DDL functional descriptions. Here the research findings on the verification of logic design are reported. Although there are several publications on research results in this field, they deal only with applications of the proof of theorems or the testing method for programs.

In this paper, a new method of verification is presented that fully exploits the state transition representation used conventionally in hardware design. The method described here detects the inconsistencies in the hardware functional descriptions and verifies that the hardware meets the given specifications, using the information extracted and arranged by the translator from the DDL functional descriptions in the state transition representation. This method is aimed basically at verifying the logic design of large-scale computers and provides an effective verification algorithm for checking the interfaces between units, in which problems of erroneous design are more frequent. Although this method is currently being evaluated using a conversational-mode test system, it is considered to be extremely effective for detecting design errors that cannot be detected by ordinary simulations, and for finding the causes of conditions that the designer never expected.

## 1. Introduction

With the advancement of computer circuits from LSI to VLSI, changes in technology are becoming increasingly complex, and the demand for high reliability in logic design is becoming even more stringent.

Previously, after the specifications were determined, a logic designer would prepare the functional design, using block diagrams, state diagrams, time charts, etc., and then would design the gate levels based on the functional design. However, because these methods do not describe the system completely, there was always the problem of communication between individual designers. Another drawback was that the design could not be checked until gate level simulation was completed.

Because of these problems, the authors adopted hardware description language, DDL (Digital System Design Language) [1] [2], at the functional design stage. By designing the gate levels only after the functional design has been checked thoroughly, design errors can be detected early, subsequently increasing the reliability of logic design. We also developed a simulator that checks the operation, and a translator that automatically converts the functional design into gate levels, thereby performing the software supporting functions for DDL[3].

Although reliability of the logic design will now be dependent solely upon reliability of the functional design (because the use of the translator makes gate level design highly reliable), it is impossible to check all possible situations during the simulations. It is also difficult to check interfacing between the units during the simulations; these interfaces pose the greatest problems in a large-scale system. Thus, to solve all problems, it will be necessary to verify the designs at the functional design stage using a method different from the usual simulations.

Although several recently-published research papers deal with hardware verification [4][5], they merely discuss the application of theorem-proving methods or program-testing methods. Unlike a program, which is in a procedural form, it is natural to describe hardware in a non-procedural manner. Therefore, even the application of program-verification methods will be limited.

Here a new method of verification is proposed which fully utilizes the conventional hardware design technique of state transition representation. The following pages depict this new method of detecting inconsistencies in hardware functional descriptions and of checking if the given specifications are met by the design, using the information extracted and arranged by the translator mentioned above.

This verification method is aimed mainly at verifying the logic design of large-scale computers, and is being investigated further to determine its practicability. In particular, the application of this method to checking the design of interfaces between units is being studied because these are the most problematic portions in a large-scale system design.

Section 2 discusses an example of inconsistency in the functional design stage, and describes the basic philosophy used in this work for checking it; that is, reducing these and further steps to merely checking whether a logical inequality is satisfied identically as

*Information Processing Laboratory, Fujitsu Laboratories Ltd.

follows.

$$(\text{Logical expression}) \equiv 0 \qquad (1.1)$$

Section 3 presents an algorithm for retracing the time for checking the inequality (1.1), and Section 4 discusses how this algorithm can be used to check if the given specifications are satisfied. Finally, Section 5 presents a brief discussion of the application of this method in our test system.

## 2. Functional Design Description and Inconsistencies Contained Therein

DDL is a language at the register transfer level for describing the functions in a hardware design. Inconsistencies often exist in a hardware design description. Verification that such inconsistencies do not exist will be covered here.

### 2.1 DDL

The hardware description language, DDL, using the state transition representation, will be described here to the extent necessary for further discussions in this paper. Figures 1 and 2 show the block diagram and the functional design descriptions using DDL for a simple example computer (SAMPLE 1). Table 1 lists the symbols used in DDL.

The signal lines and terminals are simply called Terminals. The operation of transferring the information to the terminals from the information source (called merely the Sources) is called the Connection. A connection is taken to correspond to an ideal circuit in which there is no time delay between the input and its output (Fig. 3). Here, the expression $|C1|T=A\wedge B$ means that $A\wedge B$ is connected to terminal $T$ if $C1$ is 1. The value of a terminal will be assumed to be 0 if the value connected to it is not specified.

Devices with memory function, such as flip-flops or their combinations, are called either Registers or Storage. The operation of loading the information from the data

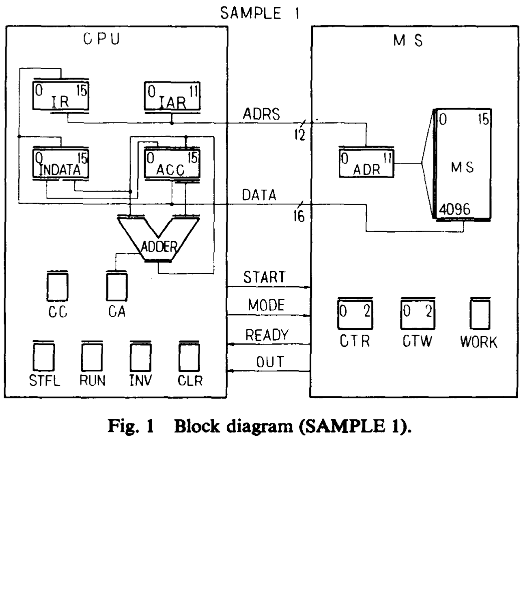

Fig. 1 Block diagram (SAMPLE 1).

```
<SYSTEM> SAMPLE1:
  <TIME> CLK(10).  /* CLOCK */
    <TERMINAL> START,MODE,ADRS(12),READY,OUT,DATA(16).
    <AUTOMATON> MS: CLK: /* AUTOMATON MS IS CONTROLLED BY CLOCK CLK */
      <STORAGE>  MS(4096,16).  /* 16 BITS 4096 WORDS */
      <REGISTER> ADR(12),WORK,CTR(3),CTW(3).
      <STATES>
        IDLE:        /* STATE IDLE */
              | START |  ADR←ADRS,WORK←1,→RCY
              ;          →IDLE..
        RCY:         /* STATE RCY */
              CTR←CTR+1,
              | CTR:=1 | →READ
              ;          →RCY..
        READ:        /* STATE READ */
              OUT=1,CTR←0,
              | ¬MODE |  DATA=MS(ADR),WORK←0,→IDLE
              ;          →WRITE..
        WRITE:       /* STATE WRITE */
              MS(ADR)←DATA,→WCY.
        WCY:         /* STATE WCY */
              CTW←CTW+1,
              | CTW:=1 | WORK←0,CTW←0,→IDLE
              ;          →WCY..
      <END>.
    <END> MS.
    <AUTOMATON> CPU: CLK: /* AUTOMATON CPU IS CONTROLLED BY CLOCK CLK */
      <REGISTER> IR(16),IAR(12),INDATA(16),ACC(16),
                 CA,CC(2),RUN,CLR,INV,STFL.
      <STATES>
        IF0:         /* STATE IF0 */
              | RUN | START=1,ADRS=IAR,→IF1
              ;       →IF0..
        IF1:         /* STATE IF1 */
              | OUT | IR←DATA,IAR←IAR+1,→IF2
              ;       →IF1..
        IF2:         /* STATE IF2 */
              | IR(0:3):=1 | START=1,ADRS=ADR,→LOAD0..
              | IR(0:3):=2 | START=1,ADRS=ADR,STFL←1,→STORE0..
                   .
                   .
                   .
        STORE0:        /* STATE STORE0 */
               | OUT | →STORE1
               ;       →STORE0..
        STORE1:        /* STATE STORE1 */
               DATA=ACC,→STORE2.
        STORE2:        /* STATE STORE2 */
               | READY | STFL←0,→IF0
               ;         →STORE2..
                   .
                   .
                   .
    <END> CPU.
<END> SAMPLE1.
```

Fig. 2 Functional design in DDL (SAMPLE 1).

Table 1 Symbols in DDL.

| Symbol | Meaning |
|---|---|
| ¬ | Negation |
| ∧ | Logical Product |
| ∨ | Logical Sum |
| := | Equality |
| = | Terminal Connection |
| ← | Register Transfer |
| → | State Transition |
| \| \| | If-Clause |
| ; | Else |
| /* */ | Comment |
| $ | Qualifier for Automaton Names |

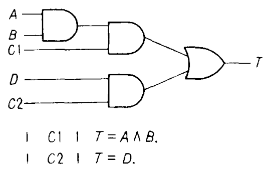

Fig. 3 Terminal and connection operation.

source (the source) into the registers (or the storage) is called a Transfer. These transfers correspond to the operation of a sequential circuit in which the input at some clock period determines the output at the next clock period, as shown in Fig. 4 (the clock periods can be thought of as discrete instants of time). For register GLFF in the figure, L is the input data, G is the input condition, and P is the clock.

Any section of the hardware that contains a set of

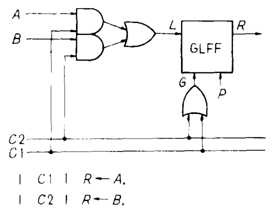

Fig. 4 Register and transfer operation.

control functions is called Automaton. Every automaton usually stays in one of its States, and the operations described for that particular state will be executed in that state. The operation of specifying the state into which the automaton should go during the next clock period is called a Transition.

The local conditions for the operations can be specified by means of IF statements:

$$|\mathrm{Be}|\ \mathrm{OP}_t.$$

or,

$$|\mathrm{Be}|\ \mathrm{OP}_t;\ \mathrm{OP}_f.$$

The set of operations, $\mathrm{OP}_t$, will be executed when the value of logical expression Be is 1 (true), and the set of operations, $\mathrm{OP}_f$, will be executed if the value of Be is 0 (false).

At this point, the DDL entries will be briefly described. The computer SAMPLE1 comprises two automatons, namely, MS and CPU, which are controlled by clock, CLK. The terminals, storages, and registers are first declared, and then the states of automaton MS and the operations executed under those states are described. The operations executed in the IDLE state are shown in lines 8 to 10. These operations are as follows: if the value of the START terminal is 1, then the value of the ADRS terminal is transferred into the ADR register, the value 1 is transferred into the WORK register, and the state will then change to RCY. If the value of START is 0, then automaton will remain in the IDLE state.

The DDL entries that express the circuit behavior are converted into information pertaining to the circuit configuration in an operation called Translation. The information so obtained is used for the gate level design. During translation, all the operations for the terminals, registers, and states will be grouped and edited according to the conditions under which they are executed. This translation, designed to assist the gate level design, also has a major role in the method of verification to be described here, because the translation list, not the original DDL entry, is referred to during the verification. Figure 5 shows a part of the result of translating the entries shown in Fig. 2 for the SAMPLE1 computer. The operations transferring a value in register CTR are the two lines underlined in Fig. 2. The operation of line 12 (executed in the RCY state), and that of line 16 (executed in the READ state), are grouped together during editing and will appear in the translation result, as shown by No. 8–1 and 8–2 in Fig. 5. The operations in Fig. 5, with no entries in the Condition column are executed unconditionally.

```
----------------------------------------------------------
|   1    ADRS     (  0: 11)     12 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE|    SOURCE     |    CONNECTION CONDITION   |
----------------------------------------------------------
| 1- 1|(0:11)| CPU$IAR(0:11)| CPU$IF0 ∧ CPU$RUN          |
| 1- 2|      | CPU$IR(4:15) | ((CPU$IR(0:3):=1) ∨ (CPU$IR|
|     |      |              | (0:3):=2) ∨ (CPU$IR(0:3):=3|
|     |      |              | ) ∨ (CPU$IR(0:3):=4) ∨ (CPU|
|     |      |              | $IR(0:3):=5)) ∧ CPU$IF2    |
----------------------------------------------------------

----------------------------------------------------------
|   2    DATA     (  0: 15)     16 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE |        SOURCE        | CONNECTION CONDITION|
----------------------------------------------------------
| 2- 1| (0:15)| CPU$ACC(0:15)        | CPU$STORE1          |
| 2- 2|       | MS$MS(MS$ADR(0:11))| ¬MODE ∧ MS$READ       |
----------------------------------------------------------

----------------------------------------------------------
|   3    MODE     (  0:  0)      1 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE |        SOURCE        | CONNECTION CONDITION|
----------------------------------------------------------
| 3- 1| (0)   | CPU$STFL             |                     |
----------------------------------------------------------

----------------------------------------------------------
|   4    OUT      (  0:  0)      1 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE |        SOURCE        | CONNECTION CONDITION|
----------------------------------------------------------
| 4- 1| (0)   | 1                    | MS$READ             |
----------------------------------------------------------

----------------------------------------------------------
|   5    READY    (  0:  0)      1 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE |        SOURCE        | CONNECTION CONDITION|
----------------------------------------------------------
| 5- 1| (0)   | ¬MS$WORK             |                     |
----------------------------------------------------------

----------------------------------------------------------
|   6    START    (  0:  0)      1 BIT(S) TERMINAL       |
----------------------------------------------------------
| NO. | RANGE|    SOURCE     |    CONNECTION CONDITION   |
----------------------------------------------------------
| 6- 1|(0)   | 1             | CPU$IF0 ∧ CPU$RUN          |
| 6- 2|      | 1             | ((CPU$IR(0:3):=1) ∨ (CPU$IR|
|     |      |               | (0:3):=2) ∨ (CPU$IR(0:3):=3|
|     |      |               | ) ∨ (CPU$IR(0:3):=4) ∨ (CPU|
|     |      |               | $IR(0:3):=5)) ∧ CPU$IF2    |
----------------------------------------------------------

----------------------------------------------------------
|   7    ADR      (  0: 11)     12 BIT(S) REGISTER       |
----------------------------------------------------------
| NO. | RANGE |       SOURCE        | TRANSFER CONDITION |
----------------------------------------------------------
| 7- 1| (0:11)| ADRS(0:11)          | MS$IDLE ∧ START    |
----------------------------------------------------------

----------------------------------------------------------
|   8    CTR      (  0:  2)      3 BIT(S) REGISTER       |
----------------------------------------------------------
| NO. | RANGE |    SOURCE    |     TRANSFER CONDITION    |
----------------------------------------------------------
| 8- 1| (0:2) | CTR(0:2) + 1| MS$RCY                     |
| 8- 2|       | 0           | MS$READ                    |
----------------------------------------------------------

----------------------------------------------------------
| NO. |   NEXT STATE    |   LAST STATE    |   CONDITION  |
----------------------------------------------------------
|12- 1| MS$IDLE         | MS$IDLE         | ¬START       |
|12- 2|                 | MS$READ         | ¬MODE        |
|12- 3|                 | MS$WCY          | CTW(0:2):= 1 |
----------------------------------------------------------
|13- 1| MS$RCY          | MS$IDLE         | START        |
|13- 2|                 | MS$RCY          | ¬(CTR(0:2):= 1)|
----------------------------------------------------------
|14- 1| MS$READ         | MS$RCY          | CTR(0:2):= 1 |
----------------------------------------------------------
|15- 1| MS$WCY          | MS$WCY          | ¬(CTW(0:2):= 1)|
|15- 2|                 | MS$WRITE        |              |
----------------------------------------------------------
|16- 1| MS$WRITE        | MS$READ         | MODE         |
----------------------------------------------------------

----------------------------------------------------------
|  26    STFL     (  0:  0)      1 BIT(S) REGISTER       |
----------------------------------------------------------
| NO. | RANGE | SOURCE |       TRANSFER CONDITION        |
----------------------------------------------------------
|26- 1| (0)   | 0      | CPU$STORE2 ∧ READY              |
|26- 2|       | 1      | (CPU$IR(0:3):= 2) ∧ CPU$IF2     |
----------------------------------------------------------

----------------------------------------------------------
| NO. |   NEXT STATE    |   LAST STATE    |   CONDITION  |
----------------------------------------------------------
|40- 1| CPU$STORE1      | CPU$STORE0      | OUT          |
----------------------------------------------------------
```

Fig. 5 Translation list (SAMPLE 1).

## 2.2 Inconsistencies Included during Functional Design

The following types of inconsistencies can be present in the descriptions of a functional design.

(i) The conditions for connecting (or transferring) different sources to the same terminal (or register) are not mutually exclusive.

(ii) The conditions for changing from one state to another are not mutually exclusive.

(iii) The logical sum of the transition conditions in any one state is not equal to 1.

In (i), two or more values can be connected (transferred) to the terminal (register) simultaneously. In (ii) and (iii), the state at the next instant of time may be either indeterminate, or two or more states. Those kinds of inconsistencies are unlikely to be detected as inconsistencies in the gate level designs. For example, the functional design using DDL in Fig. 3 contains an inconsistency of type (i) when $C1$ and $C2$ are not exclusive, but this inconsistency will not be detected in the circuit diagram, because this circuit design is the same for the logically-consistent functional design.

$$|C1 \wedge \neg C2| \quad T = A \wedge B.$$
$$|\neg C1 \wedge C2| \quad T = D.$$
$$|C1 \wedge C2| \quad T = (A \wedge B) \vee D.$$

This is one of the reasons for necessity of design verification at the functional design stage.

The following points should be checked to verify that no logical inconsistencies are in the functional design of computer SAMPLE1 (see the translation list in Fig. 5). The condition for no type (i) inconsistency to be present in the entries for the terminal ADRS is:

Conditions No. 1–1 and 1–2 should be mutually exclusive, (2.1)

where No. 1–1 and 1–2 indicate the operations in the translation list. (This notation will be followed throughout this paper.)

The conditions for a type (ii) or type (iii) inconsistency to be absent in automaton MS in the IDLE state are:

Conditions No. 12–1 and 13–1 should be mutually exclusive. (2.2)

The logical sum of conditions No. 12–1 and 13–1 should be 1. (2.3)

The checking of conditions (2.1) through (2.3) can be easily transfomed into merely testing whether or not a logical expression is identically equal to zero, as follows:

$$(\text{Logical expression}) \equiv 0. \tag{2.4}$$

That is, we can rewrite (2.1) as

$$\begin{aligned}&(\text{CPU\$IF0} \wedge \text{CPU\$RUN}) \wedge (((\text{CPU\$IR(0: 3)}:=1)\\ &\quad \vee (\text{CPU\$IR(0: 3)}:=2) \vee (\text{CPU\$IR(0: 3)}:=3)\\ &\quad \vee (\text{CPU\$IR(0: 3)}:=4) \vee (\text{CPU\$IR(0: 3)}:=5))\\ &\quad \wedge \text{CPU\$IF2}) \equiv 0,\end{aligned} \tag{2.5}$$

and both (2.2) and (2.3) as

$$\text{START} \wedge \neg\text{START} \equiv 0. \tag{2.6}$$

The aim of this paper is to develop a generalized verification algorithm that converts the verification of the absence of internal inconsistencies (as well as the verifications to be described later) into verifying only expression (2.4).

The verification of (2.4) is carried out using the various principles of Boolean algebra, such as $A \wedge \neg A \equiv 0$, and the following information:

Each of the automaton is always in one, and only one state, (2.7)

$$(A := v_1) \wedge (A := v_2) \equiv 0, \tag{2.8}$$

where $A$ is any terminal (register), and $v_1$ and $v_2$ are different values. Condition (2.5) is verified using the information provided by (2.7) and the fact that the two states of automaton CPU are CPU\$IFO and CPU\$IF2. Condition (2.6) is self-explanatory. Most of the problems within a single automaton can be verified in the above manner.

Next, the verification that there are no type (i) inconsistencies in the terminal DATA can be simplified into verifying the logical expression:

$$\text{CPU\$STORE1} \wedge \neg\text{MODE} \wedge \text{MS\$READ} \equiv 0, \tag{2.9}$$

where CPU\$STORE1 is the state of automaton CPU, and MS\$READ is the state of automaton MS. However, this expression cannot be verified using the information provided by (2.7) and (2.8), because this problem extends over two automatons. Very often different automatons are designed by different people and, hence, design errors are very likely to be present in the interface between two automatons. Thus, it is very important to verify the designs of the interfaces between the automatons, and the methods of verification presented here is particularly effective for this purpose. The algorithm used here will be described in the next section, in which it is assumed that all the automatons are synchronized to the same clock signals.

## 3. The Retracing Algorithm

When the verification extends over many automatons, it may not be possible to verify expression (2.4) using only the information provided by (2.7) and (2.8). In such situations, we start with the assumption that the left side of logical expression (2.4) is equal to 1 at some point of time, and try to extract the inconsistency by retracing the history backward in time. If the inconsistency is found, then the verification of (2.4) will be completed according to the methods of reductio ad absurdum.

However, it is not advisable to retrace the time for the entire history. For example, the necessary and sufficient condition for one-bit register $R$ to be set (the value in that register should be 1) at a particular instant of time is given as the following condition at the previous instant of time,

$$\begin{aligned}&(\text{Condition for } R \text{ to be set})\\ &\quad \vee ((R:=1) \wedge \neg(\text{Condition for } R \text{ to be reset}))\end{aligned}$$

since a register is capable of memorizing data. But if we retrace the previous necessary and sufficient conditions in time as above, the logical expression to be handled grows very quickly with the time retraced and, consequently, the verification may not be possible. Besides, most of this data will not be pertinent to the problem at hand. Hence, in this method, not all the necessary and sufficient conditions are retraced, but only the appropriate ones are altered into necessary conditions (this is called reducing to necessary conditions, and does not mean that an error is verified as normal). Because the state representations are considered to be very important in the problem of verification, the necessary and sufficient conditions for the states (given in the translation list) are retraced. We propose two such algorithms here. In addition, we also discuss some fundamental theorems of the processing of loops.

In the following, we consider the verification of two automatons, $A_1$ and $A_2$. $V_1$ and $V_2$ denote the sets of all the states of $A_1$ and $A_2$, respectively, and st $1 \wedge$ st 2 (st $1 \in V_1$, st $2 \in V_2$) is also taken to be an element (st 1, st 2) of the direct product set $V_1 \times V_2$. The algorithm indicates whether or not a logical expression of the type

$$v \wedge C(v \in V_1 \times V_2, C \text{ is the logical expression}), \qquad (3.1)$$

is false, as follows:

$$v \wedge C \equiv 0. \qquad (3.2)$$

## 3.1 Replacement

The operation of retracing the operational history in time is realized by replacing each of the conditions included in any logical expression by other appropriate logical expressions. The conditions that are replaced can be broadly classified into three types: terminal conditions, register conditions, and state conditions.

Terminal conditions are the conditions that include the terminals (excepting the external terminals that are outside the system). Terminal replacements are the operations of replacing the terminal conditions with equivalent logical expressions (the necessary and sufficient conditions for the terminal conditions at that particular instant of time) by replacing the terminals in the terminal conditions with the expressions corresponding to the source of that terminal. For example, the terminal replacement for terminal $T$ in Fig. 3 consists of replacing terminal condition $T$ (which can have one of the values–'0' and '1') with the logical expression $(A \wedge B \wedge C1) \vee (D \wedge C2)$. Although time is not retraced in the result by terminal replacement, the terminal conditions must be excluded from the logical expression to retrace the history, because the logical expression should not contain terminals that have no memory functions. (It is assumed that all storage devices have been clearly declared as such, and that there are no terminal loops.)

The register conditions are the conditions that consist only of registers. Register replacements consist of replacing the registers that appear in the register conditions with the logical expressions for the sources of these specified registers, thereby obtaining the logical expressions equivalent to the register conditions (corresponding to the necessary and sufficient conditions at the previous instant of time for those register conditions). For example, register replacement for register $R$ in Fig. 4 consists of extracting the expression $(A \wedge C1) \vee (B \wedge C2) \vee (R \wedge \neg(C1 \vee C2))$ from register condition $R$ (which can assume one of the values 0 and 1).

State replacements are the operations of replacing the states with the necessary and sufficient conditions for the automaton in question to enter that state. By carrying out register and state replacements the logical expression will be retraced in time by one interval.

## 3.2 Algorithm I

First we shall describe the algorithm that retraces the time for only the states (Step 0–Step 3).

Step 0 . . . Assume that logical expression (3.1) in the verification is true. Next, proceed to Step 1.

Step 1 . . . Eliminate the terminal conditions by carrying out terminal replacements repeatedly. The verification will be complete if the logical expression so obtained is equal to 0. Otherwise proceed to Step 2.

Step 2 . . . Force all conditions, other than state conditions, to 1 (true). The result is a logical expression of the form (which we shall call 'the standard form'):

$$v_1 \vee v_2 \vee \cdots \vee v_m(v_1, v_2, \cdots, v_m \in V_1 \times V_2)$$

Next, proceed to Step 3.

Step 3 . . . Carry out state replacement for all the states appearing in the logical expression. The verification will be complete if the logical expression so obtained is equal to 0. Otherwise return to Step 1.

In Algorithm I above, the change to necessary conditions is made in Step 2, and the time is retraced by one interval in Step 3.

The logical expression obtained after the operations of Step 1 to Step 3 on logical expression $C_1$ will be denoted by np($C_1$) (necessary pre-condition). np($C_1$) is the necessary condition at the previous interval of time for condition $C_1$ to be satisfied at a particular instant of time. Further, if $\mathrm{np}^i(i=0, 1, 2, 3, \cdots)$ is defined as follows:

$$\mathrm{np}^i(C_1) = \mathrm{np}(\mathrm{np}^{i-1}(C_1)) \quad (i \geq 1), \qquad (3.3)$$

$$\mathrm{np}^0(C_1) = C_1, \qquad (3.4)$$

then, $\mathrm{np}^i(C_1)$ will be the necessary condition at $i$ intervals of time before, for the condition $C_1$ to be satisfied at a particular instant of time. Once $\mathrm{np}^i(C_1) \equiv 0$, then the assumption that $C_1$ is satisfied at a certain instant of time will be abandoned. Therefore, Algorithm I can be used for verifying (3.2).

The execution of Algorithm I results in the formation of a graph (tree) with the value of the summit point being the elements of $V_1 \times V_2$ and 0. If $v(\in V_1 \times V_2)$ is the root in Step 0, and if np($v$) has the standard form $v_1 \vee v_2 \vee \cdots \vee v_m$, then the $m$ points $v_1, v_2, \cdots, v_m$, with the respective contents are taken as the starting points for subsequent operations. Next, for this new summit point (leaf) $\tilde{v}(\neq 0)$, the contents of the elements of $V_1 \times V_2$ in the standard form of np($\tilde{v}$) (these contents are taken as 0 if np($\bar{v}$)$\equiv$0) are taken as the new summit points, and so on.

Since Algorithm I stops only when the verification is successful, it is necessary to formulate the rules for stopping the operations using the tree described above.

Termination Rule 1 . . The verification will be completed (successful) if the contents of all the leaves is 0.

Termination Rule 2 . . . The verification will be stopped (failed) if an element of $V_1 \times V_2$ appears twice in the same path, because Algorithm I cannot be used in such cases. Consider that an element of $V_1 \times V_2$, for example $v$, appeared twice in the same path. In Algorithm I it is necessary to show $v \equiv 0$, but in this case, the problem has returned to the starting point itself. Hence, this verification cannot be made by Algorithm I, which is the reason for Termination Rule 2. However, even when the algorithm stops because of Termination Rule 2, it is possible to provide the designer with useful information regarding the counter example (such as when condition (3.2) is not likely to be satisfied). Based on this information, the designer can check for inconsistencies in the design.

The following theorem can be proved regarding the size of Algorithm I.

Theorem: Algorithm I stops, at the most, after $|V_1| \cdot |V_2|$ (the product of the number of states of the two automatons) intervals of time have been retraced.

Proof: If there is a path that has a value other than 0, even after retracing $|V_1| \cdot |V_2| (= |V_1 \times V_2|)$ intervals of time, then there must be an element that has appeared two or more times; otherwise, there will have to be a $(|V_1 \times V_2| + 1)$th element of $V_1 \times V_2$. Hence, the algorithm stops because of Termination Rule 2. Q.E.D.

As an example of applying Algorithm I, we consider the verification of (2.9). The translation result of Fig. 5 will be referred to during this verification.

Step 0 . . . CPU \$ STORE1 $\wedge$ $\neg$MODE $\wedge$ MS \$ READ, (3.5)

Step 1 . . . From No. 3–1 in Fig. 5,

CPU \$ STORE1 $\wedge$ $\neg$CPU \$ STFL $\wedge$ MS \$ READ, (3.6)

Step 2 . . . Making $\neg$CPU \$ STFL equal to 1,

CPU \$ STORE1 $\wedge$ MS \$ READ, (3.7)

Step 3 . . . From No. 14–1, 40–1,

CPU \$ STORE0 $\wedge$ OUT $\wedge$ MS \$ RCY
$\wedge$ (CTR(0: 2): = 1), (3.8)

Step 1 . . . From No. 4–1,

CPU \$ STORE0 $\wedge$ MS \$ READ $\wedge$ MS \$ RCY
$\wedge$ (CTR(0: 2): = 1). (3.9)

Using the information from (2.7) that MS \$ READ $\wedge$ MS \$ RCY$\equiv$0, it is seen that (3.9)$\equiv$0 and, hence, the verification of (2.9) is completed. There is no type (i) inconsistency in terminal DATA (bidirectional bus) between CPU and MS.

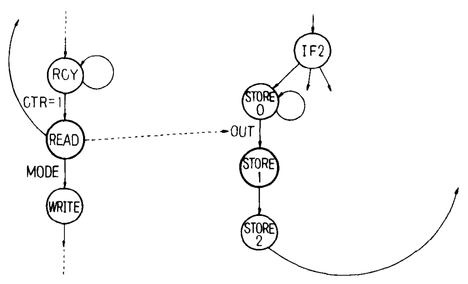

Fig. 6 State diagram (SAMPLE 1).

The above verification using Algorithm I was quite easy, because (2.9) was based on the mutually-exclusive nature of the STORE1 state of automaton CPU and the READ state of automaton MS, and also because the two automatons were synchronized via terminal OUT (Fig. 6). It is considered that Algorithm I will be effective in many cases, because the automatons generally are synchronized via the terminal signals.

## 3.3 Algorithm II

In some situations, the verification may require time retracing for registers also, because the synchronization between automatons is obtained through registers, etc. We now describe an algorithm that also retraces the time for registers. However, the time retracing will be done only for those registers that are specified by the designer, because the designer can very often identify the registers that are related to the verification problem at hand. This algorithm (Step 0 to Step 3) is a more accurate version of Algorithm I.

Step 0 and Step 1 . . . Same as Algorithm I.

Step 2 . . . All conditions, other than the state conditions and the register conditions that consist of only the specified registers, are set as 1 unconditionally.

Step 3 . . . Carry out state and register replacements for all the state and register conditions, with only the specified registers appearing in the logical expression. The verification will be completed if the resulting logical expression is equal to 0. Otherwise, return to Step 1.

(End of Algorithm II)

As in Algorithm I, if the logical expression obtained after steps 1 to 3 from logical expression $C_1$ is denoted by np($C_1$) (generally, this np($C_1$) is under stronger restrictions than the corresponding np($C_1$) in Algorithm I), and if np$^i$($C_1$) is defined as in (3.3) and (3.4), then for $C_1$ to be satisfied at some instant of time, np$^i$($C_1$) is the necessary condition that should be satisfied at $i$ intervals earlier. Therefore, (3.2) also can be verified by this algorithm for the same reason as for Algorithm I.

## 3.4 Processing of Loops

If the logical expression obtained by time retracing from logical expression $C(\neq 0)$ is such that

$$\text{np}(C)=C\vee C_1,\quad (C_1 \text{ is a logical expression})$$

then

$$\text{np}^i(C)=C\vee C_i,$$
$$(i=1,2,3,\cdots;\ C_i \text{ is a logical expression})\quad (3.10)$$

and no value of $i$ can be found for which $\text{np}^i(C)$ is equal to 0; hence, the verification appears impossible to complete.

The processing often falls into such loops when time is being retraced. The following is a theorem related to the procesing of loops.

Theorem: If

$$\text{np}(C_1)=(C_1\wedge C_2)\vee C_3,$$
$$(C_2 \text{ and } C_3 \text{ are logical expressions})$$

and the designer can explicitly state that a logical con-condition $C_a$ was satisfied for at least one earlier interval, where $C_a$ is such that

$$C_a\wedge C_1\wedge C_2\equiv 0,\quad \text{then } C_1\equiv 0 \text{ if } C_3\equiv 0.$$

Proof: Since $C_3\equiv 0$, $\text{np}(C_1)=C_1\wedge C_2$, and $\text{np}^2(C_1)=\text{np}(C_1\wedge C_2)$. Since $\text{np}(C_1\wedge C_2)\supset\text{np}(C_1)\wedge\text{np}(C_2)$ from the definition of np, we have

$$\text{np}^2(C_1)\supset\text{np}(C_1)\wedge\text{np}(C_2)=C_1\wedge C_2\wedge\text{np}(C_2).$$

Further,

$$\text{np}^i(C_1)\supset C_1\wedge C_2\wedge\text{np}(C_2)\wedge\cdots\wedge\text{np}^{i-1}(C_2)$$
$$(i=1,2,3,\cdots).$$

From $C_a\wedge C_1\wedge C_2\equiv 0$,

$$\text{np}^i(C_1)\supset\neg C_a\quad (i=1,2,3,\cdots).$$

Therefore, if it is assumed that $C_1$ is satisfied, the statement given by the designer will be contradicted.

$$\therefore\ C_1\equiv 0\qquad\text{Q.E.D.}$$

If the conditions of the theorem are satisfied, then the verification of $C_1\equiv 0$ will be reduced to the verification of only $C_3\equiv 0$.

In this section the algorithms which carry out the verification by retracing time were described. These algorithms can be used to verify the design of interfaces, which are usually the most important parts of the verification in large-scale systems.

Because these algorithms do not retrace time by following the necessary and sufficient conditions completely, the verification may sometimes fail because of the information that was discarded. Therefore, the selection of information is very important (of course, an error will never be verified as normal, irrespective of this selection of information), and this selection requires considerable knowledge of the design. On the other hand, because some information will be provided about how the subsequent verification is to be made even when the verification failed, it always will be possible to repeat the verification accordingly.

## 4. Verifying If the Basic Specifications are Met

In this section verification of the functional design whether or not it satisfies the basic specifications is discussed. An example is taken to show that even these types of verification can be reduced to those handled by Algorithm I or Algorithm II in the form of a logical expression. First, the basic specifications are given for the computer that is being designed in this example, and the design model for that computer is given in terms of DDL entries, then its verification is reduced to that of the type for (2.4). The verification of (2.4) is carried out by Algorithm II, thereby verifying whether or not the basic specifications have been satisfied.

### 4.1 Basic Specifications and Functional Design

The computer designed and verified in this example is a store-through type. The CPUs (Central Processing Units) in most modern large-scale computers contain buffer storage (there are small-capacity, high-speed, storage devices for adjusting the speed difference between the main storage and the CPU). In the store-through method, fresh data from the channel (the unit for exchanging the data between the I/O units and the storage) is stored only in the main storage (the corresponding old data in the buffer storage will not be updated). If the CPU tries to fetch the old data in the buffer storage, data will be moved into the buffer storage from the main storage. In such store-through systems, the CPU must be designed so that it always uses the most recent data.

The block diagram, the DDL functional design, and the translation list are shown in Figs. 7, 8, and 9, respectively, for the model computer design SAMPLE2, for which a part of the specification is that the CPU must always use the most recent data. For the sake of simplicity it is assumed that both main storage, M, and buffer storage, BS, have lengths of 1 word each, and the only processings by CPU that are shown here are the

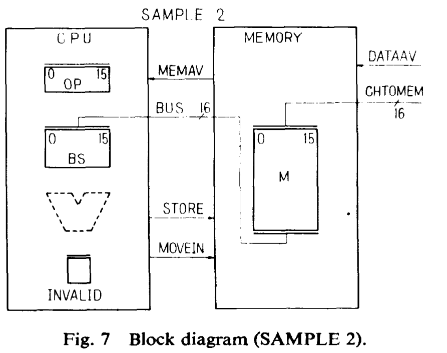

Fig. 7 Block diagram (SAMPLE 2).

```
<SYSTEM> SAMPLE2:
 <TIME> P(100).   /* CLOCK */
   <ENTRANCE> DATAAV,CHTOMEM(16). /* TERMINALS FROM OUTSIDE */
   <REGISTER> OP(16).
   <TERMINAL> MEMAV,BUS(16),STORE,MOVEIN.
   <AUTOMATON> CPU: P: /* AUTOMATON CPU IS CONTROLLED BY CLOCK P */
     <REGISTER> BS(16),INVALID.
     <STATES>
       A:          /* STATE A */
           | MEMAV |
             | OP(0:1):=0 | /* CHANNEL STORE */
               INVALID←1,STORE=1,→A..
             | OP(0:1):=1 | /* FETCH */
               | INVALID | /* MOVE-IN */
                 MOVEIN=1,→B
               ; →A...    /* FETCH OPERATION IS NOT DESCRIBED */
             | ¬(OP(0:1):=0) ∧ ¬(OP(0:1):=1) |
               /* OPERATIONS ARE NOT DESCRIBED EXCEPT STORE AND FETCH */
               →A.
           ; →A..
       B:          /* STATE B */
           INVALID←0,BS←BUS,→A.
     <END>.
   <END> CPU.
   <AUTOMATON> MEMORY: P: /* AUTOMATON MEMORY IS CONTROLLED BY CLOCK P */
     <REGISTER> M(16).
     <STATES>
       D:          /* STATE D */
           MEMAV=1,
           | STORE |  →E..
           | MOVEIN | →F..
           | ¬STORE ∧ ¬MOVEIN | →D..
       E:          /* STATE E */
           | DATAAV |  /* CHANNEL STORE */
             M←CHTOMEM,→D
           ; →E..
       F:          /* STATE F */
           BUS=M,→D.  /* MOVE-IN */
     <END>.
   <END> MEMORY.
<END> SAMPLE2.
```

Fig. 8 Functional design in DDL (SAMPLE 2).

channel-store (data transfer from the channel to the main storage) and the data-fetch from the buffer (the instructions are assumed to be transferred from an external source into register OP). When a channel-store instruction is given, 1-bit register INVALID will be set, thereby invalidating the contents of the buffer. If a fetch operation is made while the INVALID bit is on, the data is moved in, and the INVALID bit will be reset. Because the CPU executes these operations successively, the design of Fig. 8 can be considered to satisfy the specifications. It is verified in the following.

The fetch operations are made when the CPU state is $A$, terminal signal MEMAV is on, and register INVALID is off. Consequently, the condition BS = M should be satisfied if the expression $A \wedge \mathrm{MEMAV} \wedge \neg \mathrm{INVALID}$ is true. Here, the specification can be expressed as;

$$(A \wedge \mathrm{MEMAV} \wedge \neg \mathrm{INVALID}) \supset (\mathrm{BS} = \mathrm{M}). \tag{4.1}$$

Therefore, the verification of whether or not the design of Fig. 8 satisfies the specifications is reduced to verifying the expression (4.1). In fact, we shall verify whether or not the converse of this expression is identically false; that is, check if

$$A \wedge \mathrm{MEMAV} \wedge \neg \mathrm{INVALID} \wedge (\mathrm{BS} \neq \mathrm{M}) \equiv 0. \tag{4.2}$$

Thus, the verification of whether or not the basic specifications are satisfied is reduced to verifying (4.2), which is of the same form as (2.4).

### 4.2 Verification by Algorithm II

Verification of (4.2) will be executed by applying Algorithm II. Among all registers, only INVALID, BS, and M will be specified, and the algorithm will retrace the time for them also. The translation result of Fig. 9 will be referred to during the verification.

Step 0 . . .

$$A \wedge \mathrm{MEMAV} \wedge \neg \mathrm{INVALID} \wedge (\mathrm{BS} \neq \mathrm{M}). \tag{4.3}$$

```
 -----------------------------------------------------------
|   1    BUS      (  0: 15)    16 BIT(S) TERMINAL          |
 -----------------------------------------------------------
| NO. | RANGE |          SOURCE         | CONNECTION CONDITION|
 -----------------------------------------------------------
| 1- 1| (0:15)| MEMORY$M(0:15)          | MEMORY$F            |
 -----------------------------------------------------------

 -----------------------------------------------------------
|   2    MEMAV    (  0:  0)     1 BIT(S) TERMINAL          |
 -----------------------------------------------------------
| NO. | RANGE |          SOURCE         | CONNECTION CONDITION|
 -----------------------------------------------------------
| 2- 1| (0)   | 1                       | MEMORY$D            |
 -----------------------------------------------------------

 -----------------------------------------------------------
|   3    MOVEIN   (  0:  0)     1 BIT(S) TERMINAL          |
 -----------------------------------------------------------
| NO. | RANGE |SOURCE|        CONNECTION CONDITION          |
 -----------------------------------------------------------
| 3- 1| (0)   | 1    | CPU$INVALID ∧ (OP(0:1):=1) ∧ CPU$A|
|     |       |      | ∧ MEMAV                            |
 -----------------------------------------------------------

 -----------------------------------------------------------
|   4    STORE    (  0:  0)     1 BIT(S) TERMINAL          |
 -----------------------------------------------------------
| NO. | RANGE |SOURCE|        CONNECTION CONDITION          |
 -----------------------------------------------------------
| 4- 1| (0)   | 1    | (OP(0:1):= 0) ∧ CPU$A ∧ MEMAV      |
 -----------------------------------------------------------

 -----------------------------------------------------------
| NO. | NEXT STATE | LAST STATE |        CONDITION          |
 -----------------------------------------------------------
| 8- 1| D          | D          | ¬STORE ∧ ¬MOVEIN          |
| 8- 2|            | E          | DATAAV                    |
| 8- 3|            | F          |                           |
 -----------------------------------------------------------
| 9- 1| E          | D          | STORE                     |
| 9- 2|            | E          | ¬DATAAV                   |
 -----------------------------------------------------------
|10- 1| F          | D          | MOVEIN                    |
 -----------------------------------------------------------

 -----------------------------------------------------------
|  11    M        (  0: 15)    16 BIT(S) REGISTER          |
 -----------------------------------------------------------
| NO. | RANGE |          SOURCE         |  TRANSFER CONDITION |
 -----------------------------------------------------------
|11- 1| (0:15)| CHTOMEM(0:15)           | MEMORY$E ∧ DATAAV   |
 -----------------------------------------------------------

 -----------------------------------------------------------
| NO. | NEXT STATE | LAST STATE |        CONDITION          |
 -----------------------------------------------------------
|12- 1| A          | A          | (OP(0:1):=0) ∧ MEMAV      |
|12- 2|            | A          | ¬MEMAV                    |
|12- 3|            | A          | ¬((OP(0:1):=0) ∨ (OP(0:|
|     |            |            | 1):=1)) ∧ MEMAV           |
|12- 4|            | A          | ¬CPU$INVALID ∧ (OP(0:1)|
|     |            |            | :=1) ∧ MEMAV              |
|12- 5|            | B          |                           |
 -----------------------------------------------------------
|13- 1| B          | A          | CPU$INVALID ∧ (OP(0:1):|
|     |            |            | =1) ∧ MEMAV               |
 -----------------------------------------------------------

 -----------------------------------------------------------
|  14    BS       (  0: 15)    16 BIT(S) REGISTER          |
 -----------------------------------------------------------
| NO. | RANGE |          SOURCE         |  TRANSFER CONDITION |
 -----------------------------------------------------------
|14- 1| (0:15)| BUS(0:15)               | CPU$B               |
 -----------------------------------------------------------

 -----------------------------------------------------------
|  15    INVALID (  0:  0)     1 BIT(S) REGISTER           |
 -----------------------------------------------------------
| NO. | RANGE |SOURCE|         TRANSFER CONDITION           |
 -----------------------------------------------------------
|15- 1| (0)   | 0    | CPU$B                              |
|15- 2|       | 1    | (OP(0:1):=0) ∧ CPU$A ∧ MEMAV       |
 -----------------------------------------------------------
```

Fig. 9 Translation list (SAMPLE 2).

Step 1 . . . From No. 2–1 in Fig. 9,

$$A \wedge D \wedge \neg \mathrm{INVALID} \wedge (\mathrm{BS} \neq \mathrm{M}). \tag{4.4}$$

Step 3 . . .

$$① \vee ② \vee ③ \vee ④ \vee ⑤ \vee ⑥. \tag{4.5}$$

Expressions ① to ⑥ are the following:

$① \equiv A \wedge D \wedge (\mathrm{OP} \neq 0) \wedge \neg \mathrm{INVALID} \wedge (\mathrm{BS} \neq \mathrm{M})$,
$② \equiv A \wedge E \wedge \mathrm{DATAAV} \wedge \neg \mathrm{INVALID}$
$\wedge (\mathrm{BS} \neq \mathrm{CHTOMEM})$,
$③ \equiv A \wedge F \wedge \neg \mathrm{INVALID} \wedge (\mathrm{BS} \neq \mathrm{M})$,
$④ \equiv B \wedge D \wedge (\mathrm{BUS} \neq \mathrm{M})$,
$⑤ \equiv B \wedge E \wedge \mathrm{DATAAV} \wedge (\mathrm{BUS} \neq \mathrm{CHTOMEM})$,
$⑥ \equiv B \wedge F \wedge (\mathrm{BUS} \neq \mathrm{M})$.

On the other hand, $A \wedge F$, $B \wedge D$, and $B \wedge E$ are shown to be equal to 0 by applying Algorithm I, and ③, ④, ⑤ ≡ 0. Therefore, the algorithm will be applied to ① ∨ ② ∨ ⑥ from now.

Step 1 . . . From No. 1–1,

$$\textcircled{1}\vee\textcircled{2}\vee(B\wedge F\wedge(\mathrm{M}\neq\mathrm{M}))\equiv\textcircled{1}\vee\textcircled{2}. \qquad (4.6)$$

Now, considering (4.4), it turns out to be possible to apply the theorem in the preceding section to this case, with:

$$C_1\equiv A\wedge D\wedge\neg\mathrm{INVALID}\wedge(\mathrm{BS}\neq\mathrm{M}),$$
$$C_2\equiv(\mathrm{OP}\neq 0),$$
$$C_3\equiv A\wedge E\wedge\mathrm{DATAAV}\wedge\neg\mathrm{INVALID}\wedge(\mathrm{BS}\neq\mathrm{CHTOMEM}),$$
$$C_a\equiv\mathrm{INVALID}.$$

This is because, at the starting time of the computer, the buffer storage is empty, and register INVALID is loaded with 1 (for simplicity this manipulation is not given in the original DDL description in Fig. 8). Therefore, verification of (4.2) is reduced to that of:

$$A\wedge E\wedge\mathrm{DATAAV}\wedge\neg\mathrm{INVALID}\wedge(\mathrm{BS}\neq\mathrm{CHTOMEM})\equiv 0. \qquad (4.7)$$

Step 2 . . . applied to

$$A\wedge E\wedge\mathrm{DATAAV}\wedge\neg\mathrm{INVALID}\wedge(\mathrm{BS}\neq\mathrm{CHTOMEM}),$$
$$A\wedge E\wedge\neg\mathrm{INVALID}. \qquad (4.8)$$

Step 3 . . .

$$(A\wedge E\wedge\neg\mathrm{DATAAV}\wedge\neg\mathrm{INVALID})\vee(B\wedge E\wedge\neg\mathrm{DATAAV}). \qquad (4.9)$$

The theorem can be applied to this case again (with $C_a\equiv\mathrm{INVALID}$, again). Considering $B\wedge E\equiv 0$ (by algorithm I),

$$A\wedge E\wedge\neg\mathrm{INVALID}\equiv 0,$$

is obtained. Hence, the verification of (4.2) is completed.

$$\therefore\ (A\wedge\mathrm{MEMAV}\wedge\neg\mathrm{INVALID})\supset(\mathrm{BS}=\mathrm{M}).$$

This example shows that if the basic specifications can be given in the form of logic expressions, the verification of whether or not they are satisfied by the design can be carried out by applying the algorithms in the preceding section after reducing it to the verification of (2.4).

## 5. Test System and Practical Systems

A conversation-type test system was built using a minicomputer to verify (2.4) with the information of (2.7) and (2.8). The software is structured to reference the information provided by (2.7) and (2.8) and is centered on an algorithm that determines whether or not the given logical expression is a tautology according to propositional logic. The functional designs using DDL of large-scale computers were checked using the above system. The CPU functional design has about 30 K gates, the DDL descriptions require about 10 K steps, and there are four automatons, each having approximately eight states. This DDL program was translated by the DDL translator in a large-scale computer (M-180II). Using this translation result, it was verified whether or not there is any type (i) inconsistency (described in Sec. 2) among the 228 condition paris in which two different values are coupled (transferred) to the same terminal (register). The following results of this classification are shown in Fig. 10.

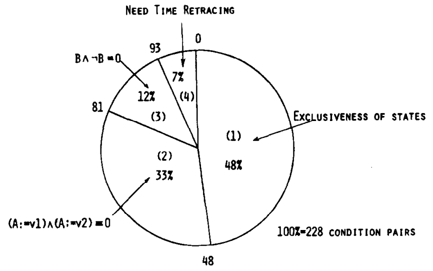

Fig. 10 Classification of exclusiveness

(1) Those verified by (2.7)
(2) Those verified by (2.8)
(3) Those verified by $B\wedge\neg B\equiv 0$ ($B$ is a logical expression)
(4) Those for which the verification requires time retracing, because the conditions extend over many automatons.

Although verifications of type (4) are few, they are very important, as well as very difficult. About half the verifications of type (4) have been successful in the test system, but some uncertain conditions still remain. These are the locations where the design was based on the different durations of time required for the different processing carried out in parallel. Although the verification of portions related to such parallel processing, or pipeline processing, can be considered to be within the scope of the methods described up to now, there are problems in our small-scale test system, such as storage capacity. When building a practical system, it is essential that a reasonably large-capacity storage be used for the list processing related to the analysis of the logical expressions, and that the list processing be carried out properly. In addition, the storage required is likely to be reduced by processing the tree described in Sec. 3 with the priority based on the depth, rather than by processing the entire logical expression at the same time. Furthermore, it may be essential to build a data base to efficiently store the translation result from which the logical expression to be substituted can be obtained at high speed during condition replacement.

## 6. Conclusion

In this paper a hardware verification method based on the state transition representation has been proposed. This method uses the information gathered by the DDL translator (which has already been developed), and it has been shown that this method is suitable for the verification of large-scale computer designs. Also, two algorithms have been presented that are very useful in verifying the interfaces between units, such interfaces being most likely to have design errors.

The methods presented here are currently being evaluated using a conversational-mode system, and it is proposed to make the system extremely effective for detecting design errors that are not detected by simulations, and for determining the causes when some conditions occur that were not foreseen by the designer. We plan to develop a practical system based on these results and to study further the methods of verification.

Finally, the author wishes to express his deep sense of gratitude to Mr. Miyakawa, Manager of his Information Processing Laboratory, for his kind advice, and to Mr. Uehara, Mr. Kawato, Mr. Saito, and many others in his laboratory for their valuable encouragement and comments. The author also wishes to express his heartfelt thanks to Mr. Tsuchimoto, Mr. Tokura, and other designers for their practical advice and help.

**References**

1. Duley, J. R. and Dietmeyer, D. L.: A Digital System Design Language (DDL), IEEE Trans. on Comp., Vol. C-17, No. 9, pp. 850–861 (1968).
2. Dietmeyer, D. L.: Logic Design of Digital Systems, p. 800, Allyn and Bacon (1977).
3. Kawato, N., Saito, T., Maruyama, F. and Uehara, T.: Design and Verification of Large Scale Computer by using DDL, Proc. 16th Design Automation Conference, pp. 360–366 (June 1979).
4. Wagner, T. J.: Hardware Verification, Ph. D. dissertation, Comp. Sci. Dept., Stanford Univ. (Sept. 1977).
5. Darringer, J. A.: The Application of Program Verification Techniques to Hardware Verification, Proc. 16th Design Automation Conference, pp. 375–381 (June 1979).