CHAPTER 1

Formal Verification Techniques for Digital Systems

MASAHIRO FUJITA, SATOSHI KOMATSU, and HIROSHI SAITO
University of Tokyo, Japan

1.1 INTRODUCTION

In deep submicron technology, a large and complex system that has a wide variety of functionalities has been integrated on a single chip. However, it is getting too harder and harder to identify all design bugs in such a large and complex system. If design bugs caused by the initial specification are identified at lower level of abstraction, we are required redesign of the system from the initial specification to fix the bugs. As a result, the productivity of the system will be decreased. In current system designs, the verification time to check whether a design is correct or not takes 80% of the overall time. Therefore, the development of verification techniques in each level of abstraction is indispensable.

Logic simulation is a widely used technique for the verification of a design. It simulates the output value for given input patterns. However, because the quality of verification deeply depends on given input patterns, there is a possibility that design bugs exist that cannot be identified during logic simulation. Because the number of required input patterns is exponentially increased when the size of a design is increased, it is impossible to verify the overall design by logic simulation. To solve this problem, the development of formal verification techniques is indispensable.

In formal verification, specification and design are translated into mathematical models. Formal verification techniques verify a design by proving the correctness mathematically. Therefore, formal verification techniques can verify the overall design exhaustively. Formal verification techniques have been widely used for the verification of software designs. These techniques are then extended for the verification of hardware designs. In particular, after the development of binary decision diagram (BDD), the ability of formal verification techniques is significantly

Dependable Computing Systems. Edited by Hassan B. Diab and Albert Y. Zomaya

improved because BDD can represent large and complex logic functions efficiently. As a result, until now, many formal verification tools for hardware designs have been developed.

In the rest of Chapter 1, we describe formal verification techniques for hardware designs. In Section 1.2, we describe basic techniques for formal verification. In particular, the overview of BDD is described. In Section 1.3, verification techniques for combinational circuits are described. In Section 1.4, verification techniques for sequential circuits are described. This section is further divided into two subproblems. One is for equivalence checking of two sequential circuits and the other is for model checking of a sequential circuit. Finally, in Section 1.5, we conclude this chapter.

1.2 BASIC TECHNIQUES FOR FORMAL VERIFICATION

1.2.1 About Formal Verification

In this subsection, we describe the overview of formal verification. Figure 1.1 shows the overview of formal verification. In formal verification, both specification and design are translated into mathematical models via front-end tools. Since this is a mathematical model, finite state machine, temporal logic, and so on are used. For a representation of these models, BDD is used. Because the ability of

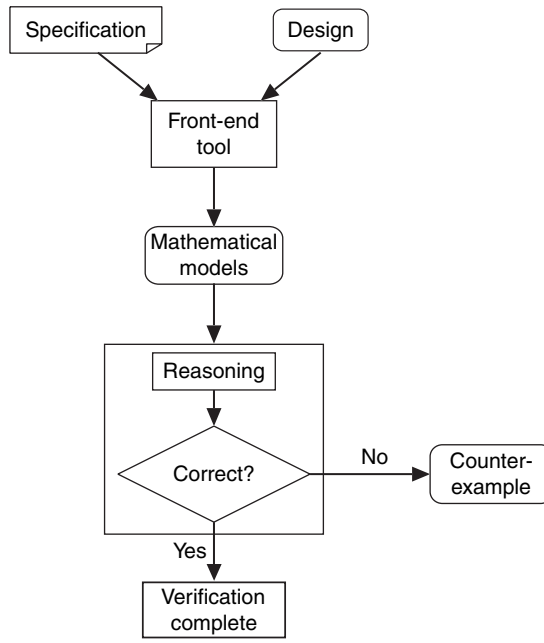

Figure 1.1

front-end tools affects formal verification, the selection of mathematical models and front-end tools is very important. After mathematical models are obtained, the correctness of designs is verified by mathematical reasoning. It is equivalent to simulate all the cases in logic simulation. If there exists a design bug, formal verification techniques produce a counterexample to support debugging. Below, we describe definitions of two formal verification techniques, which are described in Sections 1.3 and 1.4.

Equivalence Checking Equivalence checking verifies whether two given designs are equivalent or not [Fig. 1.2(a)]. In Section 1.3, we describe equivalence checking techniques of two combinational circuits. On the other hand, in Section 1.4, we describe an equivalence checking technique of two sequential circuits.

Model Checking Model checking verifies whether a design satisfies its specification or not. In Section 1.4, we describe a model checking technique by using a temporal logic.

1.2.2 BDD

Since Bryant's [5] development of efficient BDD manipulation methods, BDD has been widely used to represent logic functions. Figure 1.3(a), (b), and (c) represent BDDs for function $f = x_1 + \overline{x_2} \cdot x_3$. In BDDs, nonterminal nodes (surrounded by circles) are labeled by a variable. The variable of a nonterminal node v is denoted as var(v). Terminal nodes (surrounded by rectangles) are labeled by a logic value (i.e., 0 or 1). The logic value of a terminal node v is denoted as *value*(v). Each nonterminal node has two edges. The left side edge is traversed when the value of corresponding variable is 0. On the other hand, the right side edge is traversed when the value is 1. The following node traversed from the left edge is denoted as *low*(v), whereas the node traversed from the right edge is denoted as *high*(v).

In BDD, each node represents a logic function. The logic function f_v for a node v is defined as follows:

$$f_v = value(v) \quad \text{if } v \text{ is a terminal node}$$
$$f_v = \overline{\text{var}(v)} \cdot f_{low(v)} + \text{var}(v) \cdot f_{high(v)} \quad \text{if } v \text{ is a non-terminal node}$$

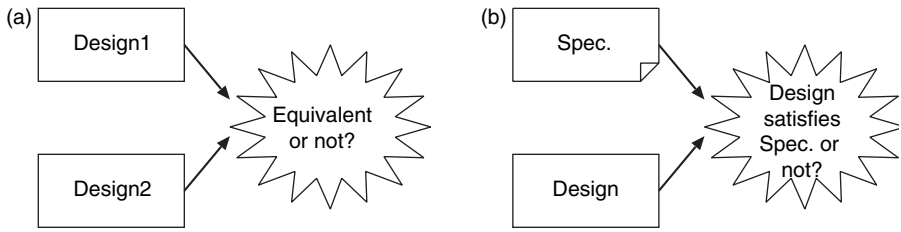

Figure 1.2

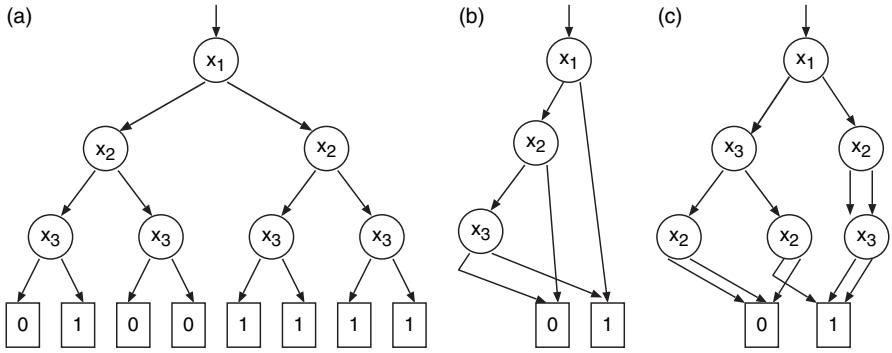

Figure 1.3

From the definition, BDD can be considered as a graph representation of a Shannon expansion. Suppose that v is a nonterminal node, $f|_{v=0}, f|_{v=0}$ are functions that are derived by assigning 0 and 1 to the function f. Then,

$$f_{low(v)} = f|_{\mathrm{var}(v)=0}$$
$$f_{high(v)} = f|_{\mathrm{var}(v)=1}$$

As shown in Fig. 1.3, several BDDs exist that represent the same logic function. In these BDDs, the reduced ordered BDD (ROBDD) is important.

Let us consider the variable order $x_1 < x_2 < x_3$ from the root of BDDs in Fig. 1.3. In BDDs of Fig. 1.3(a) and (b), this order is preserved in all paths from the root. These BDDs are called ordered BDDs (OBDDs). On the other hand, there is a path in BDD of Fig. 1.3(c) in which the order is not satisfied.

Then, we consider the reduction of BDD. If $low(v) = high(v)$ is satisfied in a node v, the node v is called redundant. The redundant node is removed, as in Fig. 1.4(a). On the other hand, if $low(u) = low(v)$ and $high(u) = high(v)$ are satisfied for a pair

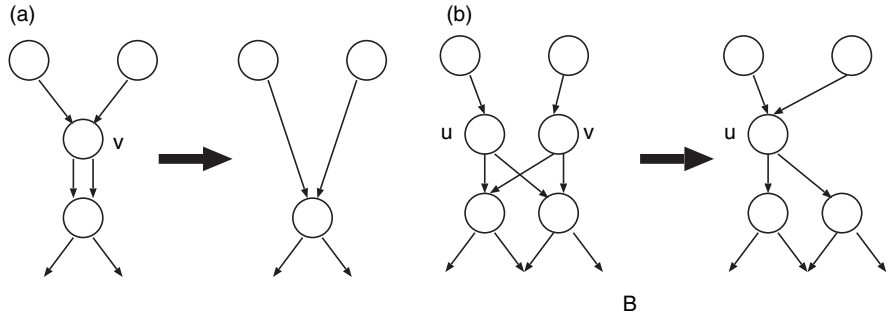

Figure 1.4

of nodes u and v, nodes u and v are said to be equivalent. One of the equivalent nodes is removed, as in Fig. 1.4(b). These operations are applied to OBDD until there is no redundant node and no equivalent node. The obtained BDD after these operations corresponds to ROBDD. Note that the size of BDD depends on the variable order. Therefore, if the size is large, dynamic variable ordering is applied to obtain smaller BDD.

There are two important characteristics in BDD. (1) ROBDD is canonical when the variable order is fixed. It means that, if several BDDs for a logic function are reduced with the same variable order, the same ROBDD is obtained. (2) The time required to a manipulation of logic functions is proportional with respect to the size of BDD. These characteristics are used for verification techniques described in Sections 1.3 and 1.4. For the detail of these characteristics, please refer to Bryant [5].

1.3 VERIFICATION TECHNIQUES FOR COMBINATIONAL CIRCUIT EQUIVALENCE

Checking functional equivalence between two logic designs is one of the key techniques in system LSI design flow. There are many situations in which we should compare two logic designs, such as to verify the correctness of logic synthesis tool, to verify manual circuit modification because of engineering changes or debugging, and so on. In the practical LSI design flow, the modifications of designed circuits occur very frequently to eliminate the design constraints violation in terms of area, delay time, power consumption, and so on. In such a situation, the equivalence verification between original circuit and modified circuit is indispensable.

Unfortunately, the equivalence checking between two logic designs is known to be a co-NP complete problem. This implies that it is difficult to efficiently verify the equivalence in all cases. To overcome the nature of this problem, the practical equivalence checking methods exploit functional similarities or structural similarities of target logic designs from the observation that most of the practical equivalence checking problem is between two similar designs. By using this property of combinational equivalence checking problem, many verification tools are presently used in the commercial system LSI designs.

The rest of this section is organized as follows. In Subsection 1.3.1, we describe basic approaches for combinational equivalence checking. Next, we show more practical verification methods based on internal equivalence points in Subsection 1.3.2. Then "false negative" problem is described in Subsection 1.3.3. Finally, we summarize combinational equivalence checking methods in Subsection 1.3.4.

1.3.1 Basic Approaches

There are a few basic approaches for checking the equivalence between two combinational circuits.

The first approach is the logic simulation method. Figure 1.5 shows the strategy of the equivalence checking by using logic simulation. The same input signals (test bench) are given to inputs of both circuits, and the outputs of each circuit are compared. If the two outputs are not identical throughout the logic simulation, the designs are not equivalent. On the other hand, the equivalence between two designs cannot be proved even if the outputs of each design are identical unless the exhaustive logic simulation is carried out. To prove the equivalence between two designs having n-inputs, the outputs are checked for 2^n input patterns, which is not practical for large combinational circuits in terms of simulation time. Note that the simulation runtime grows exponentially for the large combinational circuits.

The second approach is the functional method. For this purpose, Reduced Ordered BDD (ROBDD) [5] has been widely used for combinational equivalence checking problem. As described in the previous section, ROBDD is a canonical representation of a logic function. In other words, two logic functions have the same ROBDD representation if they are functionally equivalent. Therefore, the equivalence of two logic designs can be checked by building BDDs of both designs and then checking the equivalence between the BDDs. Because BDD can represent most practical logic functions in compact graphs, many combinational equivalence checking methods based on BDDs have been proposed [13]. However, many circuits (such as combinational multiplier) exist having BDD representations that are not compact because of their logic structures. It means that BDD-based methods are not almighty for all kinds of logic functions because performance of these methods is restricted mainly by the number of BDD nodes.

The third approach is the structural method. The structural method does not exploit the logic function of the circuits to be verified; rather, it exploits the structure and the topology of the circuits. In many cases, the structural method has a linear runtime to verify the combinational equivalence, but it is not robust compared with functional methods because it ignores the logic functions.

Although these methods are practical for the equivalence checking of small combinational circuits, many equivalence checking methods that exploit the combination of above methods have been proposed. In particular, many methods to exploit structural similarity between target two logic designs are based on internal

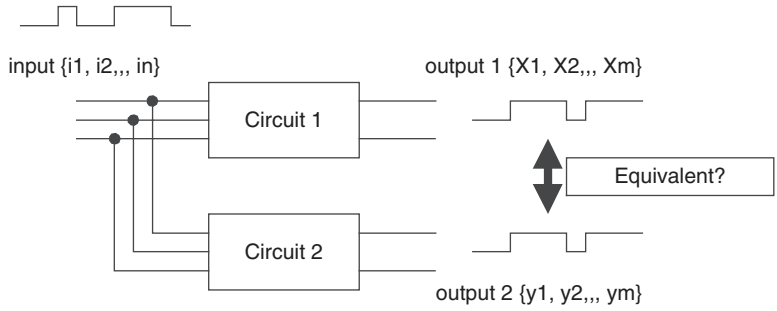

Figure 1.5

equivalence points. Internal equivalence points are used to partition the equivalence checking problem into smaller problems [1]. In actual cases where equivalence checking is carried out, two designs are considerably similar and they have a great deal of functionally or structurally equivalent points in those designs. From such an observation, many equivalence checking methods exploit internal equivalence points to reduce the size of problem and achieve practical reduced runtime.

In these equivalence checking methods, a "miter" [3] is used. Figure 1.6 shows the concept of "miter." A miter is composed of a two-input XOR gate whose inputs are connected to each output of circuits to be verified. If the output of the XOR gate is always 0 for all input patterns, the functions of two designs are equivalent. In other words, the equivalence checking problem can be translated into ATPG problem to check whether the output of the XOR gate is testable. Generally, ATPG-base equivalence checking methods are suitable for finding out inequivalence between two logic designs, whereas BDD-base equivalence checking methods are suitable for finding out equivalence of two circuits. In addition, BDD-base methods exhaust large memories to represent logic functions, whereas ATPG-base methods require long run-time to search possible input patterns.

In the following subsection, we present the combinational equivalence checking methods that exploit internal equivalence points.

1.3.2 Combinational Equivalence Checking by Using Internal Equivalence Points

In the practical case of combinational equivalence checking, two designs have much similarity in terms of their structures or functionalities. In cases of circuit modifications to eliminate constraints violations, the amount of circuit modification would be very small and most of circuit structure would be kept after the modification. In other cases, trivial circuit modifications such like buffer insertion, gate sizing, clock tree synthesis, scan insertion, and so on are frequently carried out in the LSI design flow. Equivalence checking is indispensable during these design flow. But it is important that many internal equivalence points exist in two target designs during these LSI design procedures.

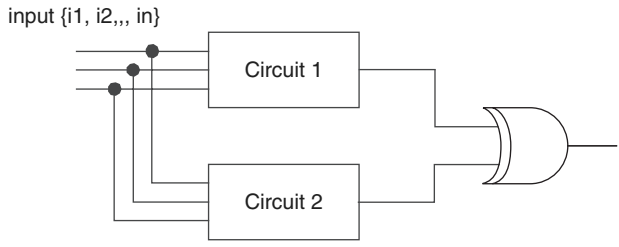

Figure 1.6

Although the combinational equivalence checking problem is a co-NP complete problem, exploiting internal equivalent points can reduce the complexity of problem (Fig. 1.7). Berman et al. [1] proposed an equivalence verification framework which exploits internal equivalence points and partitions the problem into smaller ones. This approach is very efficient, and almost all equivalence checking tools are based on this technique. However, this method has essential property that "false negative" occurs frequently. We will describe "false negative" problem later.

Figure 1.8 shows the flowchart of the verification method which uses internal equivalence points. At first, the list of "candidate of internal equivalent pairs (CIEP)" is generated. Next, a vertices pair ($v1$, $v2$) among CIEP list is selected in breadth first order from primary inputs. Then the equivalence between $v1$ and $v2$ is verified. If $v1$ is equal to $v2$, the vertices pair is added to the list of "verified internal equivalent pairs (VIEP)," and circuit substitution is carried out as shown in Fig. 1.9 [3]. Finally, the equivalence between both designs is verified. If all primary outputs are in the VIEP list after all candidates are verified, both circuits are equivalent.

Extraction of CIEPs To extract CIEPs, random pattern simulation is frequently used. Thousands of cycles of test patterns are given to primary inputs of both circuits. (Of course, the inequivalence between two circuits may be found in this logic simulation. In such a case, the verification is terminated and the counter example is acquired.) After the logic simulation, the pair of vertices that are equal throughout the simulation is listed and is added to CIEP list.

Another CIEP extraction method is the structural method. Kuehlmann et al. used circuit graph hashing by using AND/INVERTER graph [14] for this purpose [15]. The function graphs for both circuits are constructed from inputs to outputs. If a

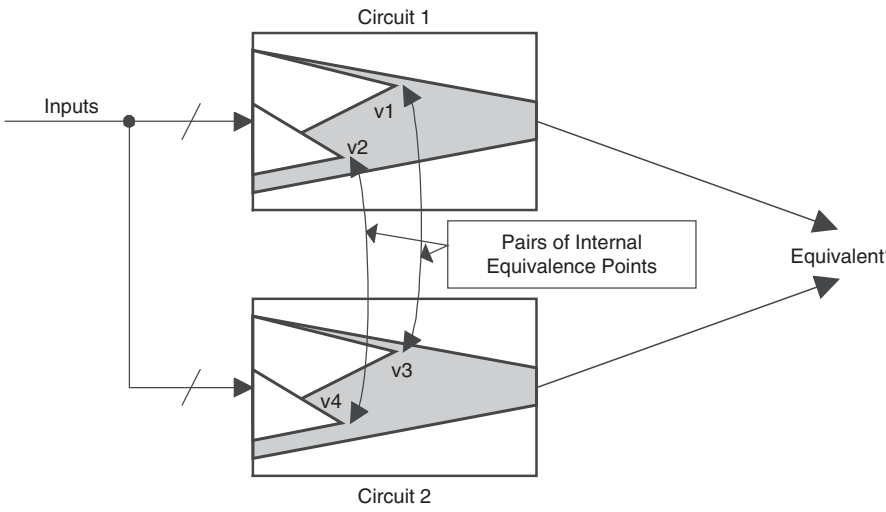

Figure 1.7

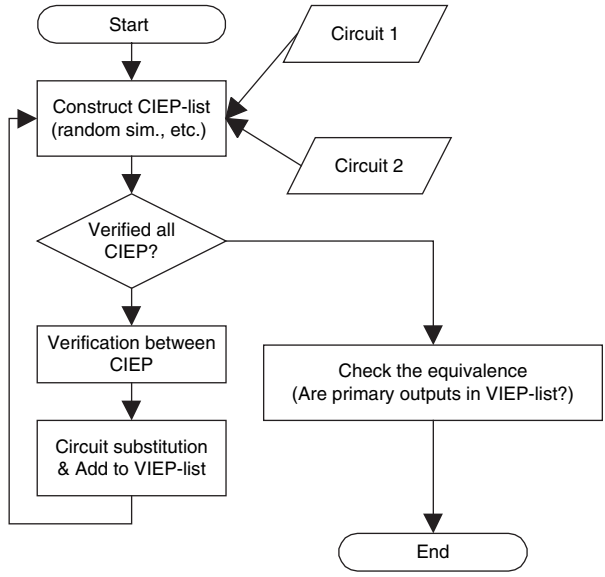

Figure 1.8

structurally identical vertex exists in the preconstructed graph, the new AND vertex is merged to the vertex (Fig. 1.10). In Fig. 1.10, a circle indicates AND node and an edge indicates a connection between two nodes. In addition, an edge with filled circle indicates the connection with INVERTER function. Because vertex $v1$ of each circuit is structurally identical, the vertices are merged. This method is based on only the structural method; the runtime to construct AND/INVERTER graph is linear to the circuit size. After the internal equivalence points are extracted, the remaining problems are forwarded to other BDD- and ATPG-based equivalence checking engines in their work.

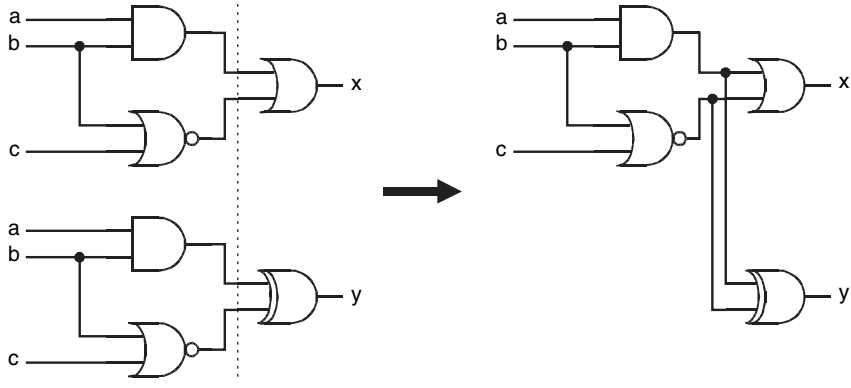

Figure 1.9

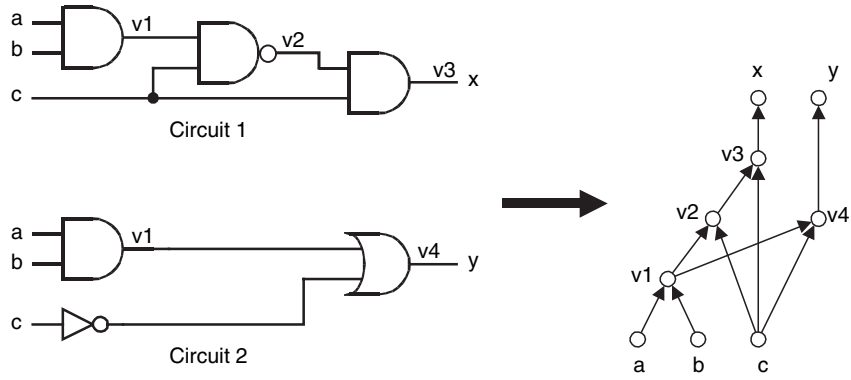

Figure 1.10

Verification Between CIEP The reduced equivalence checking problem is verified by using basic methods, such as exhaustive simulation, BDD-based-method, ATPG-based method, structural method, and so on. Because of the co-NP complete nature of the problem, a single core technique cannot verify wide variety of circuits. Therefore, many combined core techniques have been proposed to improve the robustness of the verification tools.

Even if the equivalence checking between CIEP fails, inequivalence between both internal partitioned circuits is not proved. It is because of "false negative" problem. We will describe it in the following Subsection.

1.3.3 False Negatives

As described in the previous subsection, "false negative" problems occur in the equivalence checking method that exploits internal equivalence points. Figure 1.11 shows the example of the false negative. By using the CIEP extraction methods, ($v1$, $v1'$) and ($v2$, $v2'$) are extracted as CIEP. Apparently, the equivalence of both pairs can be verified by using a basic equivalence checking method and two gates of circuit 2 are substituted by the gates of circuit 1. This procedure corresponds to Fig. 1.9. After the circuit substitution, the equivalence between $x(v1, v2) = v1 + v2$ and $y(v1', v2') = v1 \oplus v2$ is checked. However, it cannot be verified directly in this case. Actually, the case that the inputs of the partitioned circuit (the inputs pair ($v1$, $v2$)) is (1, 1) does not occur because $v1 = a \cdot b$, $v2 = \overline{b + c}$.

This situation can be solved by checking whether such input patterns can occur or not [16]. First, XOR operation between x and y is carried out.

$$\begin{aligned} x \oplus y &= (v1 + v2) \oplus (v1 \oplus v2) \\ &= (v1 + v2) \oplus (v1 \cdot \overline{v2} + \overline{v1} \cdot v2) \\ &= (v1 + v2) \cdot (v1 \cdot v2 + \overline{v1} \cdot \overline{v2}) + \overline{v1} \cdot \overline{v2} \cdot (v1 \cdot \overline{v2} + \overline{v1} \cdot v2) \\ &= v1 \cdot v2 \end{aligned}$$

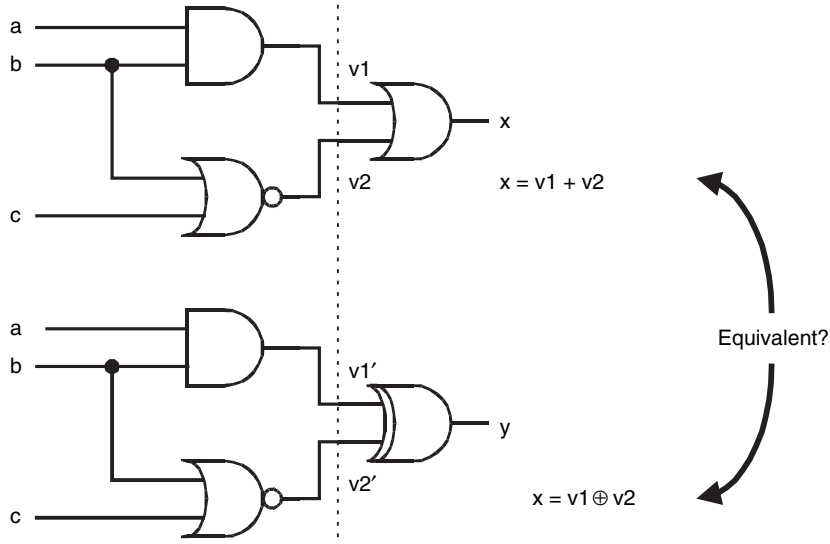

Figure 1.11

From this result, we can observe that x and y are not equal when $(v1, v2) = (1, 1)$. If $v1 \cdot v2 = 0$ is proved, the value of $x \oplus y$ is always 0. It means that the equivalence between x and y is proved. To check it, $v1 = a \cdot b$ and $v2 = \overline{b + c}$ are substituted for $v1$ and $v2$ of above equation.

$$\begin{aligned} v1 \cdot v2 &= (a \cdot b) \cdot \overline{b + c} \\ &= (a \cdot b) \cdot (\overline{b} \cdot \overline{c}) \\ &= 0 \end{aligned}$$

It means that there is no input pattern that makes $v1 \cdot v2 = 1$. In other words, x and y are always equivalent.

Another method for eliminating false negative is to expand the partitioned circuits. In this example, the equivalence can be verified by expanding the partition to primary inputs. Of course, it increases the complexity of the basic equivalence checking problem and verification runtime. In the worst case, the equivalence checker may verify whole circuits. To efficiently repartition the circuits to be verified, some heuristic methods are proposed. For example, the circuit topology is exploited to define the circuit partitioning in Matsunaga [16].

1.3.4 Summary of Combinational Circuit Equivalence Techniques

In this Section, equivalence checking methods of combinational circuits are presented. The combinational equivalence checking methods are based on the logic simulation method, the structural method, and the functional method. Although equivalence between two logic designs can be verified by using BDD-base or ATPG-base method directly, the verification runtime is very critical for the verification of large circuits.

Therefore, most of the verification techniques use internal equivalence points to partition the target circuits and reduce the size of problem. On the other hand, "false negative" problems occur inescapably because of the nature of circuit partitioning. It is one of the most important key features in the combinational equivalence checking problem.

Future research topics in combinational equivalence checking should focus on the improvement of robustness. It relies on the improvement of basic equivalence checking techniques and tuning of the combining existing equivalence techniques.

1.4 VERIFICATION TECHNIQUES FOR SEQUENTIAL CIRCUITS

1.4.1 Introduction

In this subsection, formal verification techniques of sequential circuits are described. Formal verification techniques of sequential circuits are generally categorized into two groups: equivalence checking and model (property) checking. Equivalence checking verifies whether two sequential circuits are equivalent, whereas model checking verifies whether a circuit satisfies its specification.

In formal verification of sequential circuits, a specification (property) and a circuit are translated into mathematical models. By reasoning those models, verification is carried out. In equivalence checking of two sequential circuits, each circuit is translated into a finite state machine (FSM). Given the same input sequence, equivalence checking verifies whether two FSMs have the same behavior (i.e., output) or not [11, 12]. On the other hand, in model checking of a sequential circuit, a property is translated into a temporal logic, whereas the circuit is translated into a labeled state transition graph called Kripke structure [2, 4, 6, 7, 9]. In a given state of the state transition graph, model checking verifies whether the logic holds.

The abilities of both equivalence checking and model checking depend on how to represent reachable states on a given circuit. For the representation of reachable states, explicit and implicit methods exist. In explicit methods [4, 9], reachable states are enumerated from initial states by assigning input sequences. The enumerated states are represented in the form of state transition graph. However, the explicit method is not suited for representing reachable states of a large circuit. Therefore, most verification techniques and/or tools use an implicit method for the representation of reachable states. In implicit methods [2, 6, 7], characteristic functions of a set of states and a set of next-state functions are calculated. By using these characteristic functions, reachable states are enumerated. Because these characteristic functions are represented in the form of BDD [5], reachable states of a relatively large circuit can be represented efficiently.

In the rest of this subsection, equivalence checking and model checking based on an implicit method are described.

1.4.2 FSM

A FSM F is represented as a six-tuple $\langle I, S, \delta, S_0, O, \lambda \rangle$, where I represents the set of input signals, S represents the set of states, $\delta : S \times I \rightarrow S$ represents the set of next

state functions, $S_0 (S_0 \subseteq S)$ represents the set of initial states, $O (I \cap O = \varnothing)$ represents the set of output signals, and $\lambda : (S \times I \rightarrow O)$ represents the set of functions of output signals. For example, we consider an FSM that consists of

- $I = \{i_1\}$
- $S = \{s_1, s_2, s_3, s_4\}$
- $O = \{o_1\}$
- $\delta(S \times I) = \{\delta(s_1, 0) = s_1, \delta(s_1, 1) = s_2, \delta(s_2, 0) = s_1, \delta(s_2, 1) = s_3, \delta(s_3, 0) = s_3, \delta(s_3, 1) = s_1, \delta(s_4, 0) = s_3, \delta(s_4, 1) = s_4\}$
- $\lambda(S \times I) = \{\lambda(s_1, 0) = 0, \lambda(s_1, 1) = 1, \lambda(s_2, 0) = 0, \lambda(s_2, 1) = 0, \lambda(s_3, 0) = 0, \lambda(s_3, 1) = 1, \lambda(s_4, 0) = 0, \lambda(s_4, 1) = 1\}$
- $S_0 = \{s_1\}$

Figure 1.12 shows the state transition graph of the FSM.

The reachable states of the FSM are enumerated explicitly on the state transition graph. Starting from the initial states, reachable states are traversed by considering sequences of input signals. For the example of the FSM shown in Figure 1.12, we can identify that states s_2 and s_3 are reachable from the initial state s_1 if the input signal i_1 takes the value 1 in states s_1 and s_2. On the other hand, there is no way to reach state s_4 from the initial state; i.e., s_4 is an unreachable state. In general, no unreachable states are dealt with formal verification.

1.4.3 An Implicit Method for Reachable State Representation

The main drawback of an explicit representation of reachable states is that it cannot represent reachable states of large circuits. For example, it is impossible to make the state transition graph of a sequential circuit that has 100 flip-flops because the circuit may have 2^{100} reachable states. To overcome this problem, most of verification techniques and/or tools use implicit methods. In implicit methods, the set of reachable states is represented as a characteristic function that is represented by using BDD. Therefore, reachable states of large circuits can be represented efficiently.

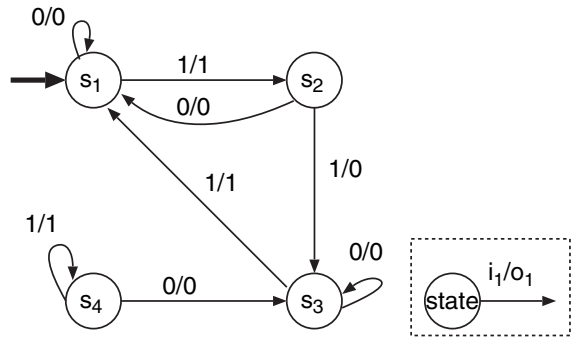

Figure 1.12

Let us consider the characteristic function of a set of states. For a set of states $S(S = \{s_1, \ldots, s_n\})$ and its subset $S'(S' = \{s_{k1}, \ldots, s_{kn}\}(1 \leq ki \leq n))$, a function $\chi S' : S \rightarrow \{0, 1\}$ is defined as follows:

$$\chi S'(s) = 1 \quad (if\ s \in S')$$
$$0 \quad (otherwise)$$

The function $\chi S'(s)$ is called characteristic function of S'. For example, in the FSM of Fig. 1.12, we represent states s_1, s_2, s_3, and s_4 with variables x_1 and x_2, such that $s_1 = \overline{x_1} \cdot \overline{x_2}$, $s_2 = \overline{x_1} \cdot x_2$, $s_3 = x_1 \cdot \overline{x_2}$, and $s_3 = x_1 \cdot x_2$. When the characteristic function of the set of reachable states S' is considered, $\chi S'(s) = 1$ in states s_1, s_2, and s_3, whereas $\chi S'(s) = 0$ in state s_4. Therefore, the logic function of $\chi S'(s)$ will be $\overline{x_1} + \overline{x_2}$. Figure 1.13 shows the BDD of which represents reachable states s_1, s_2, and s_3, respectively.

Similarly, the characteristic function of the set of next state functions $\chi\delta$ is calculated as follows. When states in S are represented by using k variables $x_1, \ldots, x_k$, the set of next state functions, $\delta : S \times I \rightarrow S$, consists of k next state functions such that $\delta_i : \{0, 1\}^k \times I \rightarrow \{0, 1\}$. When we represent next state variables as $x'_1, \ldots, x'_k$, the characteristic function $\chi\delta(x'_i, \delta_i)$ is represented as follows:

$$\chi\delta(x'_i, \delta_i) = \prod_{i=1}^{k} (x'_i \equiv \delta_i)$$

Note that $x'_i \equiv \delta_i$ corresponds to $x'_i \cdot \delta_i + x'_i \cdot \delta_i$.

Let us explain how to calculate the characteristic function for the set of next state functions in the FSM of Fig. 1.12. Suppose that next state functions for next state variables x'_1 and x'_2 are represented as $\delta_1(x_1, x_2, i_1) = i_1 \cdot x_2 + i_1 \cdot x_1$ and $\delta_2(x_1, x_2, i_1) = i_1 \cdot \overline{x_1} \cdot \overline{x_2}$. The characteristic function of the set of next state

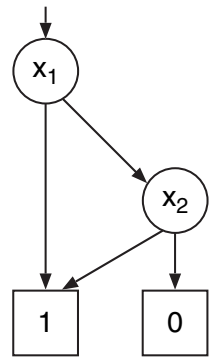

Figure 1.13

functions is as follows:

$$\chi\delta(x_1', x_2', \delta_1, \delta_2) = (x_1' \equiv \delta_1)(x_2' \equiv \delta_2)$$

$$\begin{aligned}\chi\delta(x1', x2', x1, x2, i1) = {} & \overline{x1'} \cdot \overline{x2'} \cdot \overline{x1} \cdot \overline{i1} + \overline{x1'} \cdot x1 \cdot \overline{x2} \cdot i1 + \overline{x1'} \cdot x2' \cdot \overline{x1} \cdot \overline{x2} \cdot i1 \\ & + x1' \cdot x1 \cdot \overline{x2} \cdot \overline{i1} + x1' \cdot \overline{x2'} \cdot x1 \cdot x2 \cdot \overline{i1} + x1' \cdot \overline{x2'} \cdot x2 \cdot i1\end{aligned}$$

Figure 1.14 shows the algorithm that enumerates reachable states when an FSM is given. The inputs of the algorithm are the set of states S, the set of input signals I, the set of next state functions δ, and the set of initial states S_0. In the beginning of the algorithm, the set of reached states *Reached*, the set of states that are the source of state transitions *From*, and the set of states traversed after kth state transitions New^k are initialized by the set of initial states S_0. The following procedures are then carried out while $New^k \neq \varnothing$. (1) The set of states *To* that is traversed by one state transition from the states of *From* is calculated. To calculate *To* is called image computation and represented as the function $Img(\delta, From)$. The detail of the function $Img(\delta, From)$ is described later. (2) New^k is calculated by removing the states in *Reached* from the states in *To*. (3) The obtained New^k is set to *From* for the next state enumeration. (4) Finally, the update of *Reached* by the union of *Reached* and New^k is carried out.

The implementation of the function $Img(\delta, From)$ is different between explicit method and implicit method. In explicit methods, *To* is calculated by enumerating all possible inputs for all states in *From*. On the other hand, in implicit methods, smoothing operation is carried out to calculate *To*. For a logic function f with n variables ($f(x_1, x_2, \ldots, x_n)$), smoothing operation ${}^{\exists}x_i f$ with respect to variable x_i is defined as follows:

$${}^{\exists}x_i f = f_{x_i} + f_{\overline{x_i}}$$

```
BFS_FSM (S, I,  , S0) {
    Reached = From = New^0= S0;
    k = 0;
    do {
        k = k + 1;
        To = Img( , From);                    (1)
        New^k = To - Reached;                 (2)
        From = New^k;                         (3)
        Reached = Reached U New^k;            (4)
    } while (New^k /= ?);
}
```

Figure 1.14

f_{x_i} is derived by assigning 1 for x_i in function f, whereas $f_{\overline{x_i}}$ is derived by assigning 0. Similarly, smoothing operation with respect to variables in a set $X = (x_1, \ldots x_n)$ is defined as follows:

$$^{\exists}Xf = {}^{\exists}x_1({}^{\exists}x_2(\ldots({}^{\exists}x_n f)))$$

As an example, we apply smoothing operation to variable x_1 in a function $f(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 + \overline{x_1} \cdot x_3 + x_2 \cdot x_3 \cdot x_4$.

$$f_{x_1} = x_2$$
$$f_{x_1} = x_3$$

Therefore,

$$^{\exists}x_1 f(x_1, x_2, x_3, x_4) = x_2 + x_3$$

The function $Img(\delta, From)$ with smoothing operation is defined as follows:

$$Img(\delta, From) = {}^{\exists}S{}^{\exists}I(\chi From \cdot \chi\delta).$$

The product of the characteristic functions $\chi From$ and $\chi\delta$ represents the set of states that are traversed from *From* by one state transition. Therefore, after the application of the smoothing operation to the product with respect to variables in S and I, we can obtain the function that is represented by next state variables. For example, in the FSM of Figure 1.12, we calculate the next states traversed from the initial state $\overline{x_1} \cdot \overline{x_2}$ when next state functions $\delta_1(x_1, x_2, i_1) = i_1 \cdot x_2 + i_1 \cdot x_1$ and $\delta_2(x_1, x_2, i_1) = i_1 \cdot \overline{x_1} \cdot \overline{x_2}$ are given. The characteristic function for the set of next state functions $\chi\delta$ is represented as follows:

$$\chi\delta = (x_1' \equiv \delta_1)(x_2' \equiv \delta_2) = (x_1' \equiv i_1 \cdot x_2 + i_1 \cdot x_1)(x_2' \equiv i_1 \cdot x_1 \cdot x_2).$$

The characteristic function for the initial state χS_0 is represented as follows:

$$\chi S_0 = \overline{x_1} \cdot \overline{x_2}.$$

Therefore, the next states traversed from the initial state $\overline{x_1} \cdot \overline{x_2}$ are calculated by the product of $\chi\delta$ and χS_0.

$$\begin{aligned}\chi\delta \cdot \chi S_0 &= (x_1' \equiv i_1 \cdot x_2 + \overline{i_1} \cdot x_1)(x_2' \equiv i_1 \cdot \overline{x_1} \cdot \overline{x_2}) \cdot (\overline{x_1} \cdot \overline{x_2}) \\ &= (x_1' \equiv 0)(x_2' \equiv i_1 \cdot \overline{x_1} \cdot \overline{x_2})\end{aligned}$$

Then, smoothing operation with respect to the variables in sets S and I is applied.

$$^{\exists}I\chi\delta \cdot \chi S_0 = (x_1' \equiv 0)(x_2' \equiv \overline{x_1} \cdot \overline{x_2})$$
$$^{\exists}S^{\exists}I\chi\delta \cdot \chi S_0 = \overline{x_1'} \cdot x_2' + \overline{x_1'} \cdot \overline{x_2'}$$

As a result, we can identify that the next states for s_1 are states $s_1(\overline{x_1'} \cdot \overline{x_2'})$ and $s_2(\overline{x_1'} \cdot x_2')$.

1.4.4 Equivalence Checking of Finite State Machines

Equivalence checking of two sequential circuits verifies whether the behavior of two given FSMs $M1 = \langle I, S, \delta 1, S_0, O, \lambda 1\rangle$ and $M2 = \langle I, T, \delta 2, T_0, O, \lambda 2\rangle$ is equivalent. It is equivalent to check whether an input signal exists that leads to a different output signal when the same input sequence is given from the initial states of $M1$ and $M2$. Note that both $M1$ and $M2$ have the same set of input signals I and the same set of output signals O.

This problem is considered by using the product machine of $M1$ and $M2$, as shown in Fig. 1.15. In the product machine, XNOR of output signals is calculated for all input sequences. The output signals of $M1$ and $M2$ are equivalent when XNOR of each pair of output signals is 1. On the other hand, $M1$ and $M2$ are inequivalent when there exists a pair of output signals such that XNOR is 0. Therefore, we check whether a pair of states in $M1$ and $M2$ such that XNOR is 0 is reachable from the initial states of the product machine.

The product machine of $M1$ and $M2$, $M12 = \langle I, U, \delta 12, U_0, \{0, 1\}, \lambda 12\rangle$ is defined as follows:

- I: *the set of inputs*
- $U = S \times T$
- $\delta 12 : (S \times T) \times I \rightarrow (S \times T) = (\delta 1 : S \times I \rightarrow S, \delta 2 : T \times I \rightarrow T)$
- $U_0 = S_0 \times T_0$

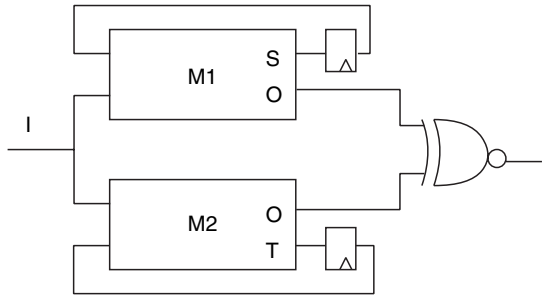

Figure 1.15

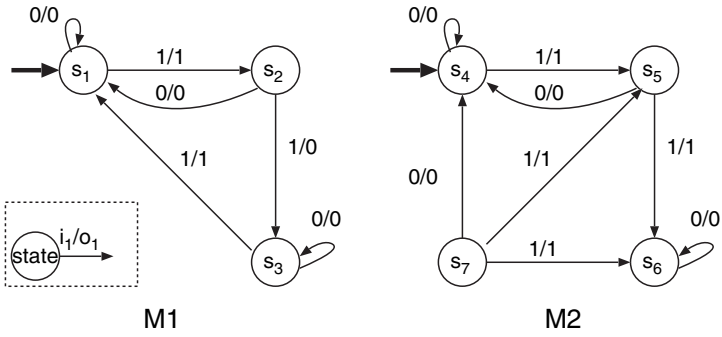

Figure 1.16

- $\{0, 1\}$: *output value of XNOR*
- $\lambda 12 : (S \times T) \times I \rightarrow \{0, 1\}(1\ if\ \lambda 1(S, I) = \lambda 2(T, I), 0\ if\ otherwise$

Let us verify the equivalence of two FSMs shown in Fig. 1.16. Note that the set of reachable states on the product machine is calculated based on the implicit method shown in Section 1.4.3.

States of FSM $M1$ are represented as $s_1 = \overline{x_1} \cdot \overline{x_2}$, $s_2 = \overline{x_1} \cdot x_2$, and $s_3 = x_1 \cdot \overline{x_2}$. When we represent the input signal as i_1, the next state functions δ_1 and $\delta_2(\delta 1 = \{\delta_1, \delta_2\})$ and the output function $\lambda_1(\lambda 1 = \{\lambda_1\})$ are represented as follows:

$$\begin{aligned}\delta_1 &= i_1 \cdot x_2 + \overline{i_1} \cdot x_1 \\ \delta_2 &= i_1 \cdot \overline{x_1} \cdot \overline{x_2} \\ \lambda_1 &= i_1 \cdot \overline{x_2}\end{aligned}$$

On the other hand, states of FSM $M2$ are represented as $s_4 = \overline{x_3} \cdot \overline{x_4}$, $s_5 = \overline{x_3} \cdot x_4$, $s_6 = x_3 \cdot \overline{x_4}$, and $s_7 = x_3 \cdot x_4$. The next state functions δ_3 and $\delta_4(\delta 2 = \{\delta_3, \delta_4\})$ and the output function $\lambda_2(\lambda 2 = \{\lambda_2\})$ are represented as follows:

$$\begin{aligned}\delta_3 &= x_3 \cdot \overline{x_4} + i_1 \cdot \overline{x_3} \cdot x_4 \\ \delta_4 &= i_1 \cdot x_3 + i_1 \cdot \overline{x_4} \\ \lambda_2 &= i_1\end{aligned}$$

The output function of the product machine is calculated as follows:

$$\begin{aligned}\overline{\lambda 1 \oplus \lambda 2} &= (\overline{i_1 \cdot \overline{x_2}})\overline{i_1} + i_1 \cdot \overline{x_2} \\ &= (\overline{i_1} + x_2)\overline{i_1} + i_1 \cdot \overline{x_2} \\ &= \overline{i_1} + \overline{x_2}\end{aligned}$$

The function implies that the value of XNOR is 1 when $i_1 = 0$, regardless of state variables. On the other hand, the value of XNOR depends on the value of $\overline{x_2}$ when $i_1 = 1$. The outputs of $M1$ and $M2$ are different in the states where $x_2 = 1$

because in such states XNOR will be 0. Therefore, we check whether those states are reachable from the initial states of $M12$.

Suppose that the next state variables of $M1$ and $M2$ are represented as x'_1, x'_2, x'_3, and x'_4. The characteristic function of the set of next state functions for $M12$ is represented as follows:

$$\begin{aligned}\chi\delta 12 = \chi\delta 1 \cdot \chi\delta 2 &= (x'_1 \equiv \delta_1)(x'_2 \equiv \delta_2)(x'_3 \equiv \delta_3)(x'_4 \equiv \delta_4)\\ &= (x'_1 \equiv i_1 \cdot x_2 + \overline{i_1} \cdot x_1)(x'_2 \equiv i_1 \cdot \overline{x_1} \cdot \overline{x_2})\\ &\quad (x'_3 \equiv x_3 \cdot \overline{x_4} + i_1 \cdot \overline{x_3} \cdot x_4)(x'_4 \equiv i_1 \cdot x_3 + i_1 \cdot \overline{x_4})\end{aligned}$$

The characteristic function of the initial state $s_1 : s_4$ are represented as follows:

$$\chi U_0 = \chi S_0 \cdot \chi T_0 = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$$

Therefore, the set of the next states traversed from the initial state $s_1 : s_4$ are calculated as follows:

$$\begin{aligned}\chi\delta 12 \cdot \chi U_0 &= (x'_1 \equiv i_1 \cdot x_2 + \overline{i_1} \cdot x_1)(x'_2 \equiv i_1 \cdot \overline{x_1} \cdot \overline{x_2})(x'_3 \equiv x_3 \cdot \overline{x_4} + i_1 \cdot \overline{x_3} \cdot x_4)\\ &\quad (x'_4 \equiv i_1 \cdot x_3 + i_1 \cdot \overline{x_4})(\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4})\\ &= (x'_1 \equiv 0)(x'_2 \equiv i_1 \cdot \overline{x_1} \cdot \overline{x_2})(x'_3 \equiv 0)(x'_4 \equiv i_1 \cdot \overline{x_4})\end{aligned}$$

Then, we apply smoothing operation to the variables in sets U and I.

$$\begin{aligned}{}^{\exists}I\chi\delta 12 \cdot \chi U_0 &= (x'_1 \equiv 0)(x'_2 \equiv \overline{x_1} \cdot \overline{x_2})(x'_3 \equiv 0)(x'_4 \equiv \overline{x_4})\\ {}^{\exists}U{}^{\exists}I\chi\delta 12 \cdot \delta U_0 &= \overline{x'_1} \cdot \overline{x'_2} \cdot \overline{x'_3} \cdot \overline{x'_4} + \overline{x'_1} \cdot x'_2 \cdot \overline{x'_3} \cdot x'_4\end{aligned}$$

As a result, we can identify that the next states of the initial state $s_1 : s_4$ are states $s_1 : s_4$ and $s_2 : s_5$. Because x'_2 is 1 in state $s_2 : s_5$, the product machine produces 0. It means that two FSMs are inequivalent. Note that the state transition graph of the product machine is shown in Fig. 1.17.

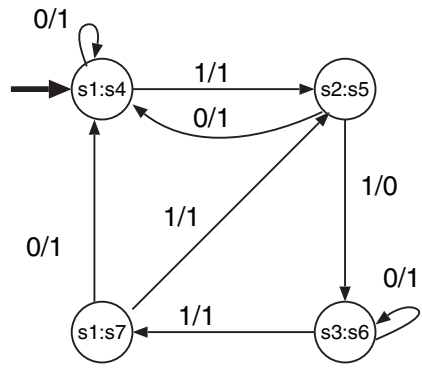

Figure 1.17

1.4.5 Computation Tree Logic and Model Checking

Model checking verifies whether a circuit satisfies its specification (property). Therefore, not only the circuit but also the property is translated into mathematical models. In general, to specify not only static properties but also dynamic properties such that an acknowledgment signal will be true three cycles later when a request signal is true, computation tree logic (CTL), a kind of temporal logic, is used for model checking.

In CTL model checking [2, 4, 6, 7, 9], a circuit is modeled as a labeled state transition graph called Kripke structure. A Kripke structure M is represented as a three-tuple $M = \langle S, R, P \rangle$, where S represents the set of states, R represents the set of state transitions ($R : S \times S \rightarrow \{0, 1\}$), and P represents a mapping function for atomic propositions. The mapping function P, which is denoted as $P : S \rightarrow 2^{|AP|}$ (AP is the set of atomic propositions), represents the set of propositions that are true in each state.

In CTL model checking, $(M, s_0) \models \phi$ means that a CTL formula ϕ holds true in the initial state s_0 of a Kripke structure M. A CTL formula consists of proposition p, NOT operator $^-$, and AND operator $\wedge$. In addition, there are four temporal operators F (Future), G (Global), X (Next), and U (Until) to represent time and two path operators A (Always) and E (Eventually) to represent computation paths from a state. True or false of CTL formulas is defined as follows. Note that ϕ and ψ represent CTL formulas.

- $(M, s_0) \models \phi$ iff $p \in P(s_0)$
- $(M, s_0) \models \overline{\phi}$ iff $(M, s_0)^- \models \phi$
- $(M, s_0) \models \phi \wedge \psi$ iff $(M, s_0) \models \phi$ and $(M, s_0) \models \psi$
- $(M, s_0) \models AX\phi$ iff $^\forall$ state t such that $(s_0, t) \in R$, $(M, t) \models \phi$
- $(M, s_0) \models EX\phi$ iff $^\exists$ state t such that $(s_0, t) \in R$, $(M, t) \models \phi$
- $(M, s_0) \models AF\phi$ iff $^\forall$ path from s_0, there is a state s_i such that $(M, s_i) \models \phi$
- $(M, s_0) \models EF\phi$ iff $^\exists$ path from s_0, there is a state s_i such that $(M, s_i) \models \phi$
- $(M, s_0) \models AG\phi$ iff $^\forall$ path from s_0, $(M, s_i) \models \phi$ is all states s_i
- $(M, s_0) \models EG\phi$ iff $^\exists$ path from s_0, $(M, s_i) \models \phi$ is all states s_i
- $(M, s_0) \models A(\phi U \psi)$ iff $^\forall$ path from s_0, there is a state s_k such that $(M, s_k) \models \psi$ and $^\forall i (0 < i < k)(M, s_i) \models \phi$
- $(M, s_0) \models E(\phi U \psi)$ iff $^\exists$ path from s_0, there is a state s_k such that $(M, s_k) \models \psi$ and $^\forall i (0 < i < k)(M, s_i) \models \phi$

Figure 1.18 shows the image of $AF\phi$, $EF\phi$, $AG\phi$, and $A(\phi U \psi)$.

In an algorithm of CTL model checking, the set of states that holds a property (CTL formula ϕ) is enumerated. The algorithm then checks whether the initial states are included in the set of states. If the initial states are included, it means that the property holds in a given circuit. Otherwise, a counter-example of a violation will be generated. Note that the implicit method described in

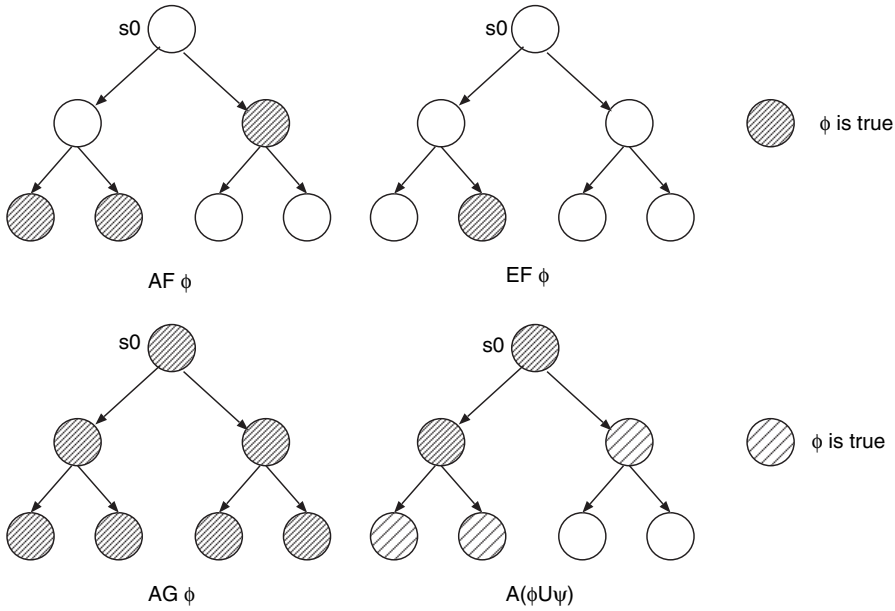

Figure 1.18

Section 1.4.3 is used to calculate the set of states. The algorithm to verify $EG\phi$ is as follows:

1. Calculate the set of states T_0 such that ϕ is true $(T_0 = t|(M, t) \models \phi)$
2. $i = 0$
3. Calculate the set of states $S(s \in S)$ such that $^{\exists}t \in T_i(s, t) \in R$ and $(M, s) \models \phi$
4. $T_{i+1} = T_i \cap S$
5. Terminate if $T_{i+1} = T_i$
6. $i = i + 1$ goto 3

1.4.6 Summary of Sequential Circuit Verification Techniques

In this subsection, equivalence checking and model checking of sequential circuits are described. In both techniques, a specification and a circuit are translated into mathematical models. After reachable state enumeration, verification is carried out for all reachable states. Although an explicit method is intuitive for reachable state enumeration, it is not suited for large circuits. Therefore, most verification techniques and/or tools use an implicit method. Because reachable states are represented as a characteristic function and stored on a BDD, the state space of relatively large circuits can be represented efficiently.

However, as an actual problem, there is a possibility that an implicit method does not work well when reachable states are increased exponentially. To overcome

this problem, verification techniques with abstraction methods have been proposed [8, 10].

1.5 SUMMARY

In Chapter 1, we described formal verification techniques for digital hardware systems. Different from logic simulation, we can expect the complete verification when we use formal verification techniques because in formal verification techniques the correctness of a design is proved mathematically. In Chapter 1, we described equivalence checking and model checking techniques that utilize BDD. These techniques are well established for hardware system verification. Therefore, we can look at many reports for the application of them to real designs. However, there is a big problem in formal verification. If the size of a target design is increased, formal verification techniques take a long verification time. In the worst case, the verification will fail. To overcome this problem, the development of verification techniques that are robust to the size problem are necessary.

REFERENCES

1. C. L. Berman and L. H. Trevillyan. Functional Comparison of Logic Designs for VLSI Circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, Nov. 1989, pp. 456–459, 1989.
2. S. Bose and A. Fisher. Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic. In *Proc. Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 759–764, 1989.
3. D. Brand. Verification of Large Synthesized Designs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, Nov.1993, pp. 534–537, 1993.
4. M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. In *IEEE Transaction on Computers*, vol. C-35, no. 12, pp. 1035–1044, 1986.
5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, vol. 35, pp. 677–691, 1986.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Symbolic model checking: 10^{20} states and beyond. In *Proc. Logic in Computer Science*, 1990.
7. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. Design Automation Conference*, pp. 46–51, 1990.
8. M. Chiodo, T. Shiple, A. Sangiovanni-Vincentelli, and R. Brayton. Automatic Composition and Minimization in CTL Model Checking. In *Proc. International Conference on Computer Aided Design*, pp. 172–178, 1992.
9. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specification. In *ACM Transactions on Programming Languages and Systems*, 8(2): 244–263, 1986.

10. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. In *ACM Transactions on Programming Language and Systems*, vol. 16, no. 5, pp. 1512–1542, 1992.
11. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems* (edited by J. Sifakis), LNCS 407, Springer, 1989.
12. O. Courdert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building Their State Diagrams. In *Proc. Workshop on Computer-Aided Verification*, 1990.
13. M. Fujita, M. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proceedings of the IEEE nternational Conference on Computer Aided Design*, Nov. 1988, pp. 2–5, 1988.
14. S. W. Jeong, B. Plessier, G. Hachtel, and F. Somenze. Extended BDD's: Trading off Canonicity for Structure in Verification Alogrithms. In *Proceedings of the IEEE International Conference on Computer Aided Design*, Nov. 1991, pp. 464–467, 1991.
15. A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 263–268, 1997.
16. Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. In *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 629–634, 1996.