Logic Verification of Incomplete Functions and Design Error Location

Qinhai ZHANG Charles TRULLEMANS

Laboratoire de Microélectronique, Université Catholique de Louvain
B-1348 Louvain-La-Neuve, Belgium

Abstract: At the stage of logic verification, it is necessary not only to detect but also to locate the sources of design errors that may exist in the gate-level circuit. For an incompletely specified function, a method to compute the corresponding 3-terminal BDD that represents the ON-set, OFF-set and DC-set, is described. Two incomplete functions are equivalent if, and only if, their 3-terminal BDDs are isomorphic. If the gate-level circuit is verified to be incorrect, a conditional stuck-at fault model is proposed to represent the circuit with design errors. The incorrect logic values at the design error sites can be considered as conditional stuck-at faults. A design error locating method, based on fault simulation and released pattern generation, is described.

1 Introduction

Verification techniques[1-3] are often used to check the equivalence between the functional-level description and its gate-level implementation. Two complete functions are equivalent if, and only if, their 2-terminal BDDs are isomorphic[3]. For the verification of incomplete functions, the tautology checking[4], and the D-variable BDD[5] were proposed. In this paper, we describe a method to compute the 3-terminal BDDs from the given incomplete functions, and show that the 3-terminal BDDs maintain the canonical form even though complement edges are used. If the common don't care set is used, two incomplete functions are equivalent if, and only if, their 3-terminal BDDs are isomorphic.

If a circuit is verified to be incorrect, the design errors that cause the misbehaviors of the circuit need to be found and corrected. A design error would occur, for example, by accidental substitution of one AND gate for an OR gate. For the location and correction of design errors, several works[6-10] have been reported. In this paper, a conditional stuck-at fault model is proposed. If a circuit is verified to be incorrect, a set of patterns that distinguish the circuit from its original description are generated. The design errors will cause some signal lines in the gate-level circuit to be faulty sources. When applying a distinguishing pattern to the circuit, some of these faulty sources can be considered as conditional stuck-at faults compared with the ideal circuit. If we only consider such design errors that cause single fault source, then the conditional stuck-at fault at the faulty source will be sensitized, and propagated to the primary outputs. A design error locating method, based on the fault simulation and released pattern generation, is described. The experimental results show that our method can locate the design error at one signal line in most cases.

In section 2, we first discuss the construction of a 3-terminal BDD from the 2-terminal BDDs. We then show that the logic verification for two incompletely specified functions requires a simply comparison of their 3-terminal BDDs. In section 3, we propose the conditional stuck-at fault model, then we describe the design error location method, and give some experimental results.

2 Verification of Incomplete Functions

2.1 A Review of BDDs

Basic definitions of 2-terminal BDD and its implementation are given in [3, 11]. With the fixed input variable ordering, a 2-terminal BDD is a canonical form for representation of a complete function. For any complete function, the corresponding 2-terminal BDD can be implemented by the recursive formulation of ITE(F, G, H) = (F * G + ~F * H)[11]. For example, the function F = a ⊕ b can be represented as F = ITE(a, ~b, b), and its 2-terminal BDD is illustrated in Fig. 1a. If the complement edges are used, only the terminal 1 is necessary, and the canonical form can be maintained if all the "1" branches always keep the regular form[11]. In Fig. 1b, the compacted BDD of the function F = a ⊕ b is shown. A dot on an edge indicates a complement edge.

Fig. 1. Flat and compacted BDDs of function F = a ⊕ b

For representation of incomplete function with BDD, two methods are basically used. The first method uses the 3-terminal BDD, and expresses the don't cares explicitly, as shown in Fig. 2a. The second method encodes the incomplete function F into a pair of complete functions $[f_0, f_1]$. The f_0 and f_1 are the minimum and maximum function of F when the don't care value 'x' is regarded as '0' and '1' respectively. This method combines the pair $[f_0, f_1]$ with an additional D-variable[5], and expresses the don't cares implicitly, as shown in Fig. 2b. We use the 3-terminal BDDs in this paper to represent incomplete functions.

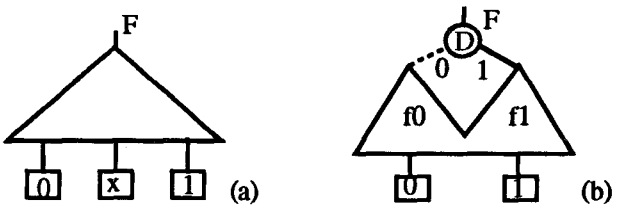


Fig. 2. (a) 3-terminal BDD (b) D-variable BDD

The sizes of BDDs for a given function are very sensitive to the input variable ordering[3]. An efficient method to compute the input variable ordering from a particular gate-level circuit, is described in [12]. The dynamic weight selection and intermediate node determination are used in that method to obtain the input variable ordering hierarchically.

2.2 Implementation of 3-Terminal BDDs

By given the ON-set F_{on} and the don't care set F_{dc} of an incomplete function F, the 2-terminal BDDs of F_{on} and F_{dc} are implemented. Note that F_{on} might be a gate-level implementation of the incomplete function. Therefore, all the paths from the root of BDD F_{on} to the terminal "1" contain the complete ON-set and partial DC-set(DC_{on}), and all the paths from the root of BDD F_{on} to the terminal "0" contain the complete OFF-set and partial DC-set(DC_{off}). Our aim is to extract the pure ON-set, OFF-set and DC-set and present them in one graph. For this purpose, an operator "#" is introduced for the construction of the 3-terminal BDD $F = F_{on} \# F_{dc}$. This is illustrated in Fig. 3.

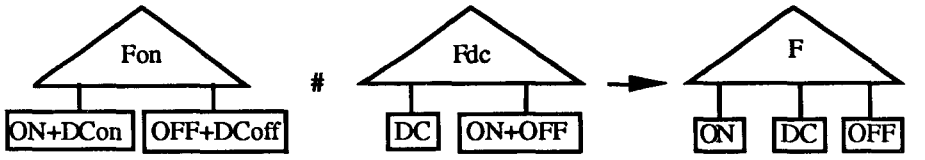


Fig. 3. Construction of the 3-terminal BDD F from 2-terminal BDDs F_{on} and F_{dc}

The basic operation of "#" is defined in Table 1. If we use symbol "x" for the terminal "DC", symbol "1" for the terminal "ON+DC_{on}", and symbol "0" for the terminals "OFF+DC_{off}" and "ON+OFF", as described in Table 2, we can see that the operator "#" performs a partial function of the exclusive-OR operator "⊕".

Table 1: Basic definition of the operator "#"

F_{on} \ F_{dc}	DC	ON + OFF
ON + DC_{on}	DC	ON
OFF + DC_{off}	DC	OFF

Table 2: the operator "#" performs a partial function of the operator "⊕"

F_{on} \ F_{dc}	x	0
1	x	1
0	x	0

In our algorithm, the both 2-terminal BDDs of F_{on} and F_{dc} are first implemented with the same terminal "1" in order to merge sub-functions. For representing an incomplete function, an additional terminal "x" is needed to indicate the don't care cases. By using the ITE(F_{dc}, ~F_{on}, F_{on}) for 3-terminal BDDs, two constraints are added. First, the node v_j of the BDD Fdc is the first element of the triple. Second, a [v_j, p_j] pair is used, where the p_j is the path polarity from the root to the node v_j. The terminal cases are ITE([1, 0], v_i, ~v_i) = x and ITE([1, 1], v_i, ~v_i) = v_i, where v_i is a node of BDD F_{on}. Canonical representation form is a very important feature of BDD. To guarantee the canonical form of a 3-terminal BDD if complement edges are used, two additional standard triples must be added, in addition to the four standard triples described in [8]. These two additional triples are shown in Fig. 4. We always choose

the right member of each equivalent pair. One example of constructing a 3-terminal BDD F from the two 2-terminal BDDs F_{on} and F_{dc} is shown in Fig. 5.

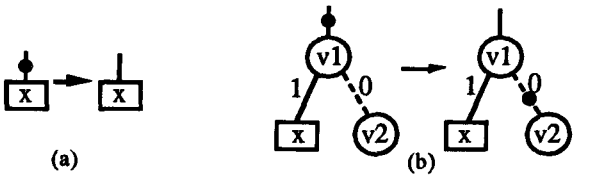


Fig. 4. Two additional standard triples

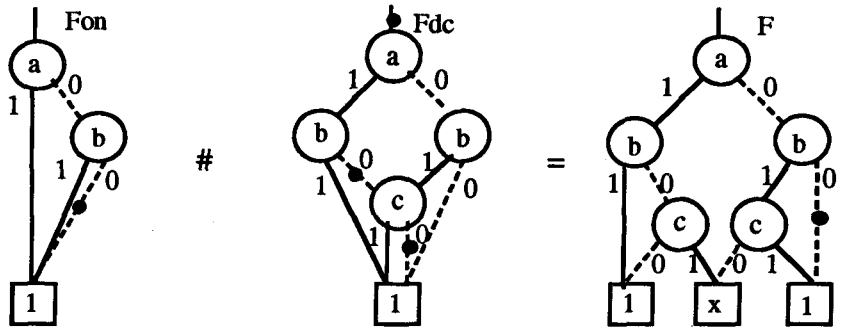


Fig. 5. An example of $F = F_{on} \# F_{dc}$

2.3 Experimental Results: Verification Based on 3-Terminal BDDs

The verification of incomplete functions can be carried out by comparison the 3-terminal BDDs. If the BDDs of function F(G) is computed from the F_{on}(Gon) and the common don't care set, i.e., $F = F_{on} \# (F_{dc} + G_{dc})$, and $G = G_{on} \# (F_{dc} + G_{dc})$, the function F is equivalent to G if, and only if, their corresponding 3-terminal BDDs are isomorphic. Table 3 shows the experimental results on verification of incomplete functions for some circuits[16] compared with the tautology checking[4]. From lack of don't care sets of these circuits, two incomplete functions F and G are constructed for circuit C as: $F_{on}[i] = G_{dc}[i] = O_c[i]$, and $F_{dc}[i] = G_{on}[i] = O_c[i\text{-}1]$. Since $F_{on}[i] \in F_{dc}[i] \cup G_{dc}[i]$ and $G_{on}[i] \in F_{dc}[i] \cup G_{dc}[i]$, the two incomplete functions F and G must be equivalent.

Table 3: Verification based on 3-terminal BDDs & Tautology Checking

		Verification Time		Maximum Memory	
Circuits	BDDs Time	3-T BDDs	Tautology	3-T BDDs	Tautology
C1355	1234.38	0.25	0.29	1453	1453
C1908	341.17	0.07	0.06	243	243
C432	185.78	7.08	8.27	162527	162527
C499	1197.75	37.25	30.23	22996	22995
C5313	46.54	67.32	22.75	9101	9824
C7552	52.56	94.27	197.80	19366	18848
C880	9.78	10.03	14.31	26722	31972

The verification process is applied for each output separately. For a selected output, the 2-terminal BDDs F_{on}, F_{dc}, G_{on}, and G_{dc} are first constructed. The CPU time is given in seconds on a DEC 5000/200 workstation. The maximum memory is a measure of the maximum BDD node requirement, during the verification and BDDs implementation. The results show that our algorithm, for the verification of two incompletely specified functions, is comparable with the tautology checking with respect to the CPU time and memory requirement.

3 Design Error Location

3.1 Distinguishing Pattern Generation

For the purpose of locating design errors, a pre-simulation process is used. Both the functional-level description F and gate-level circuit G are first simulated with 64 random patterns. Once an erroneous output has been found, the pre-simulation process is terminated, and the formal verification for that output begins in order to generate the complete distinguishing pattern set. If no erroneous output has been found, the BDD verification then performs on each output separately.

For a selected erroneous output O_i, the function $Dif = F_i \oplus G_i$ is defined as a difference function. All the paths from the root of Dif to the terminal 1 contribute to the set of distinguishing patterns. If the functional-level description F is not available, the set of simulation patterns given by the designer can be used as a partial functional description, and all the unspecified patterns are treated as don't cares.

3.2 Conditional Stuck-at Fault Model

Design errors may exist in the gate-level design due to the accidental introduction by the designer, or to the undiscovered bugs in the logic synthesis software. In practice, the most common design errors are simple errors, namely, substitution of one simple gate for another, simple extra or missing gate errors, extra or missing wire errors, exchange wire errors, etc.[1]. Design errors can be classified into two classes: (I) the design errors only cause a signal line to be faulty source; (II) the design errors make two or more signal lines to be faulty sources. Most of the simple design errors are class-I design errors. Fig. 6 shows some simple design errors.

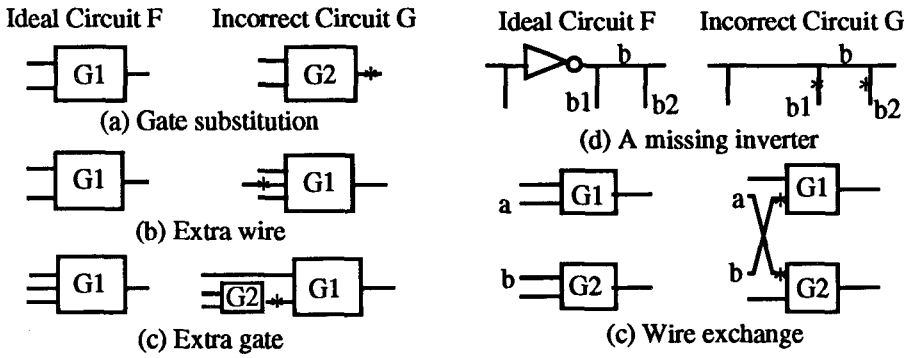


Fig. 6. Some simple design errors

The '*' symbols marked at the incorrect circuits indicate the positions of faulty sources. The complex design errors and multiple design errors usually create multiple faulty sources, and are taken into account in class-II design errors. The location of the sources of class-II design errors is much more difficult than class-I design errors. We assume that only single class-I design error in the circuit is considered.

For analyzing the function of the circuit and locating the sources of design errors, a conditional stuck-at fault model is proposed. Assume that the functions of signal line h in the ideal circuit F and the incorrect circuit G are $h_f(x)$ and $h_g(x)$ respectively. For a given input pattern v, a conditional stuck-at-$h_g(v)$ fault is said to exist at the signal line h if $h_f(v) = \sim h_g(v)$. Notes that the widely used stuck-at fault model assumes that a signal line h is stuck-at "0" or "1" permanently. In contrast, the design errors within a circuit make some signal lines to be temporarily, or conditionally stuck at "0" or "1" under certain patterns. For example, in Fig. 7, a design error exists by substituting the OR gate in the ideal circuit(a) for a NAND gate in the incorrect circuit(b). The two circuits are not equivalent under the patterns [a/1, b/1, c/1] and [a/1, b/0, c/0]. For the first pattern, the signal lines h and o in the incorrect circuit(b) seem to be both stuck-at-0, and for the second pattern, the same signal lines h and o seem to be stuck-at-1.

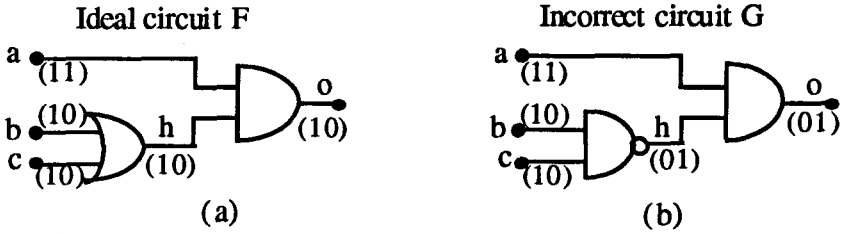


Fig. 7. One example of conditional stuck-at fault

3.3 Design Error Location

Assume that the signal line h is the faulty source, and its associated functions in the circuits F and G are $h_f(x)$ and $h_g(x)$ respectively. The Shannon expansions of functions F and G with respect to the internal variable h are defined as:

$$F(x) = h_f(x) * F_{hf=1}(x) + \sim h_f(x) * F_{hf=0}(x) \quad \ldots (3\text{-}1)$$

and

$$G(x) = h_g(x) * G_{hg=1}(x) + \sim h_g(x) * G_{hg=0}(x) \quad \ldots (3\text{-}2)$$

If considering the internal variable h as an independent variable, the function F*(x,h) must be equivalent to the function G*(x,h), because the difference between functions F(x) and G(x) only results from the difference of $h_f(x)$ and $h_g(x)$. That is, $F_{hf=1}(x) = G_{hg=1}(x)$, and $F_{hf=0}(x) = G_{hg=0}(x)$. Thus, from (3-1) and (3-2), we have

$$Dif = [h_f(x) \oplus h_g(x)] * dG^*(x,h)/dh \quad \ldots (3\text{-}3)$$

Where $dG^*(x,h)/dh = [G_{hg=1}(x) \oplus G_{hg=0}(x)]$ represents the Boolean difference of circuit G with respect to the internal variable h.

Theorem 3.1: A signal line h is said to be the faulty source if, and only if, the following two conditions are satisfied for each distinguishing pattern v: (a) the

conditional stuck-at-$h_g(v)$ fault is detected, i.e., $dG^*(v,h)/dh = 1$; (b) the conditional stuck-at-$h_g(v)$ fault is excited, i.e., $h_f(v) \oplus h_g(v) = 1$.
Proof. The theorem holds directly from the equation (3-3).

By the condition (a) of theorem 3.1, any signal line h that satisfies $dG^*(x,h)/dh \supseteq$ Dif, is a candidate of the faulty source. Given a distinguishing pattern v, the detectability of the conditional stuck-at-$h_g(v)$ fault at the signal line h can be easily determined by fault simulation. However, the signal lines that are on the sensitive paths and propagate the fault effect to the outputs, also satisfy the condition. In order to determine such signal lines, we give the following theorem.

Theorem 3.2: Assume that there are $k(k \geq 1)$ fan-ins of signal line h that have the controlling values for a distinguishing pattern v. If there is a pattern u, $u \in dG^*(x,h)/dh$, but $u \notin$ Dif + DC_i(the don't care set of output O_i), and if there are also k fan-ins that have the controlling values with the pattern u, the signal line h is not the faulty source. Such a pattern u is called a released pattern to the signal line h.
Proof. Assume that the function h_f of signal line h is a symmetric function of its fan-ins, $h_f(x_1, x_2, ...,x_i, x_j, ...,x_m) = h_f(x_1, x_2, ...,x_j, x_i, ...,x_m)$. This assumption is suitable for all the basic gates. Let the signal line h be the faulty source. Since there are the same number of fan-ins that have the controlling values for both patterns v and u, we have $h_f(v) = h_f(u)$, and $h_g(v) = h_g(u)$. Therefore, the pattern u also excites the stuck-at-$h_g(v)$ fault at the signal line h, i.e., $h_g(u) \oplus h_f(u) = 1$. Since $u \in dG^*(x,h)/dh$, the pattern u must be a distinguishing pattern according to the theorem 3.1. This is contradictory to $u \notin$ Dif. ♦

For locating the faulty source that lies somewhere within the circuit, two steps are used. First, the set of signal lines that satisfy $dG^*(x,h)/dh \supseteq$ Dif is computed by fault simulation. The initial set of candidates includes all the signal lines that are the transitive fan-ins of current erroneous output. For each distinguishing pattern v, the circuit is first simulated. Then for each signal line h in the candidate set, a stuck-at-$\tilde{}h_g(v)$ fault is injected, and fault simulation is performed. If the fault is detected, the signal line h remains a candidate, otherwise the signal line h is dropped. After the fault simulation, the candidate set usually contains more than one signal lines. By the theorem 3.2, we can further reduce the candidate set by released pattern generation. If there is a released pattern for the signal line h, it can be dropped from the candidate set. This technique is very efficient for eliminating the candidates that are only on the sensitive paths.

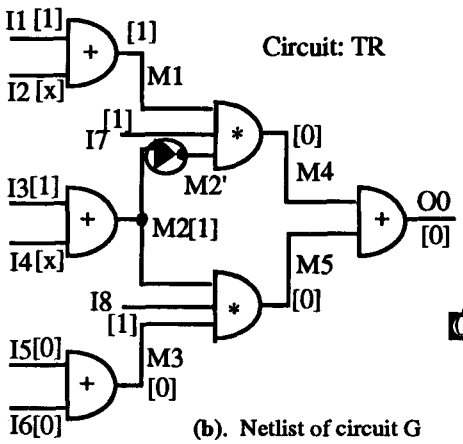
As an explanation of our method, one simple circuit TR is used. The functional-level description F written in Boolean equations is given in Fig. 8a, the net-list G is shown in Fig. 8b, and no don't care exists. The variable ordering is computed as {I3, I4, I7, I1, I2, I8, I5, I6}, and the BDD of Dif is shown in Fig. 8c. 11 distinguishing patterns were generated.

By performing the fault simulation with these distinguishing patterns, only the fan-out stems that are in the current candidate set, are needed to be explicitly simulated. For example, at the pattern labeled at Fig. 8b, a conditional stuck-at-0 fault at the fan-out stem M2 can be detected, M2 remains a candidate, the signal lines {I1, I2, M1, I7, I3, I4, I8}, and the fan-out branch {M2-M5} are dropped from the set of candidates. After the fault simulation, the candidate set contains the signal lines {O0, M4, M2'}

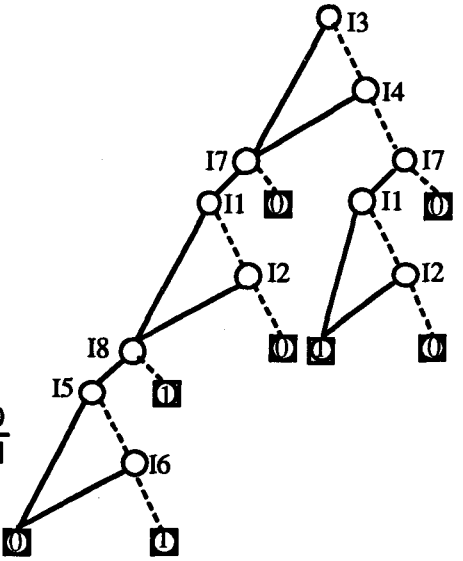
and the fan-out branch {M2-M2'}. By released pattern generation, we can find a released pattern [I1/x, I2/x, I3/1, I4/x, I5/1, I6/x, I7/0, I8/1] for both signal lines O0 and M4. Then, the signal lines O0 and M4 are dropped. The final candidate set contains the signal line {M2'} and the fan-out branch {M2-M2'}. Notes that the signal line M2' and the fan-out branch M2-M2' are error equivalent sites.

```
:NAME-CIRCUIT TR
:INPUT-VARIABLES (I0 I1 I2 I3 I4 I5 I6 I7 I8)
:OUTPUT-VARIABLES (O0)
(
" S1 = I1 + I2 " ON
" S2 = I3 + I4 " ON
" S3 = I5 + I6 " ON
" O0 = S1 * I7 * S2 + S3 * I8 * S2 " ON
)
```

(a). Functional-level description F



(b). Netlist of circuit G



(c). The BDD of function Dif

Fig. 8. An example of design error location

3.4 Hybrid Fault Simulation and Released Pattern Generation

Any combinational circuit can be partitioned into fan-out free regions(FFR)[13]. With FFRs, the expensive fault simulation of combinational circuits can be restricted to the explicit simulation of fan-out stems, and the detectability of other faults are determined locally inside the FFRs[13]. Critical path tracing[14] deals with the faults only implicitly, no fault dropping is necessary. However, the narrowing of faulty candidates is required for the design error location. In our method, the explicit fault simulation is only performed on the fan-out stems that are in the current candidate set.

Since some primary inputs may be unspecified in a distinguishing pattern, the BDDs are used as symbolic representations of signal lines in logic simulation. With BDD technique, a fault injection can be introduced to the objective signal line by

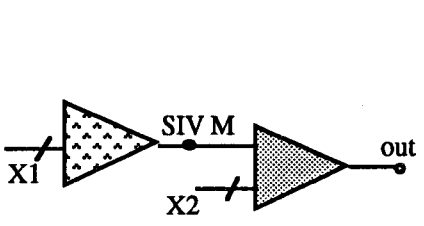
complementing the related BDD. The fault is said to be detected if, and only if, the output BDD of the circuit is also complemented.

After the fault simulation, the final candidate set usually contains several candidates. In fact, most of them are only on the sensitive paths, and propagate the fault effect to the output. For eliminating such candidates, released pattern generation, which is analogous to test pattern generation[15], is introduced. However, there are two main differences. First, the released pattern does not fall into the don't care set and distinguishing pattern set. For this limitation, the BDD of Dif acts as a guidance for the input variable decision making, during the released pattern generation. Second, the released pattern not only detects a conditional stuck-at fault at the signal line h, but also set the same number of fan-ins of signal line h with the controlling values as previously simulated distinguishing patterns.

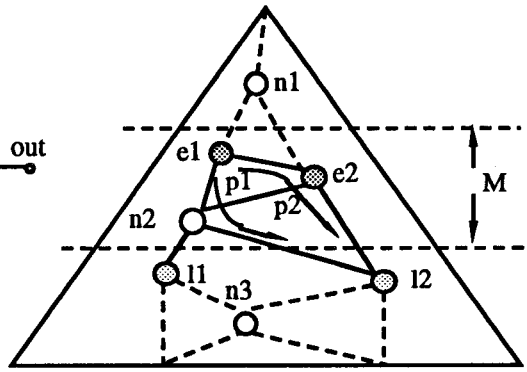
3.5 Speed up Techniques

For a complex circuit, the complete set extracted from the BDD of Dif is very large, and then the fault simulation with these distinguishing patterns either takes a large fraction of computing time, or becomes impossible. In order to speed up the design error locating process, two heuristics are described in this sub-section.

Static Intermediate Variables: If there is a static intermediate variable(SIV)[12] in a circuit, the circuit can be divided by this SIV into two parts as shown in Fig. 9a. Since all the combinations of X1 will set the logic value of the SIV M to 0 or 1, the SIV M can be used as a pseudo primary input to stand for all the inputs X1.



(a). Circuit with a static intermediate variable M

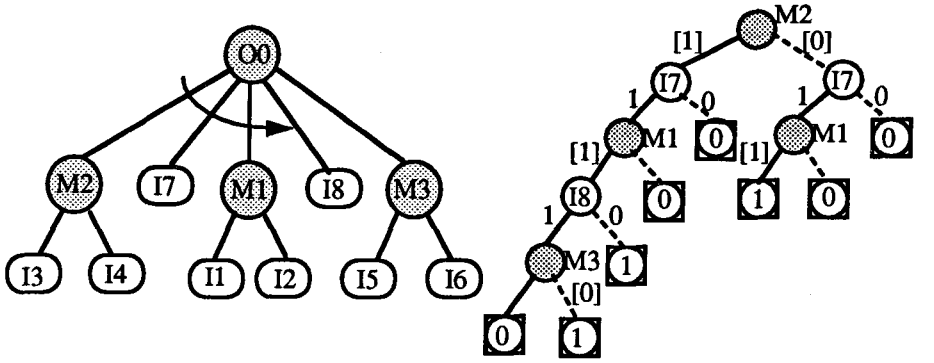


(b). Entering nodes and leaving nodes

Fig. 9. SIV nodes and the corresponding sections in the BDD

According to the strategy of input variable ordering[12], a subset of input variables covered by a SIV M are grouped and corresponding to a section of Dif. For a path from the root of Dif to the terminal 1, an entering node is defined as the first node located in the section, and a leaving node is defined as the first node out of the section. As shown in Fig. 9b, the nodes e1 and e2 are entering nodes of section M, the nodes l1 and l2 are the leaving nodes, and n1, n2 and n3 are neither entering nodes, nor leaving nodes. A triplet $[e_i, l_j, v_{ij}]$ is used to indicates the logic value v_{ij} of a SIV for

all the paths between an entering node e_i and a leaving node l_j. If there are Pa paths from the root of the BDD Dif to the entering node e_i, Pb paths from e_i to the leaving node l_j, and Pc paths from l_j to the terminal 1, then a total of Pa * (Pb - 1) * Pc distinguishing patterns can be reduced by using the triplet $[e_i, l_j, v_{ij}]$. The SIVs and their covered primary inputs can be found in the ordering tree[12]. Fig. 10a shows the ordering tree for the circuit shown in Fig. 8. The nodes M2, M1 and M3 are SIVs. We first select the highest level SIVs {M2, M1, M3} to replace the covered input variables in the corresponding sections. The modified Dif is shown in Fig. 10b. By using the SIVs, seven distinguishing patterns can be saved. However, if a SIV node remains in the final candidate set, the design error location process for the corresponding sub-circuit is then continued.



(a). Variable ordering tree (b). The modifed Dif with [ei, lj, vij] triplets

Fig. 10. Variable ordering tree and modified BDD Dif

Fault Simulation with Partial Distinguishing Patterns: It has been observed that the size of candidate set rapidly decreases in the case of applying a small number of distinguishing patterns, then, it slowly decreases as shown in Fig. 11. Therefore, to instead of simulating all the distinguishing patterns, we choose a number of distinguishing patterns for fault simulation. When the size of candidate set reaches an appropriate value, the fault simulation is terminated, and the released pattern generation is then used for further reducing the candidate set.

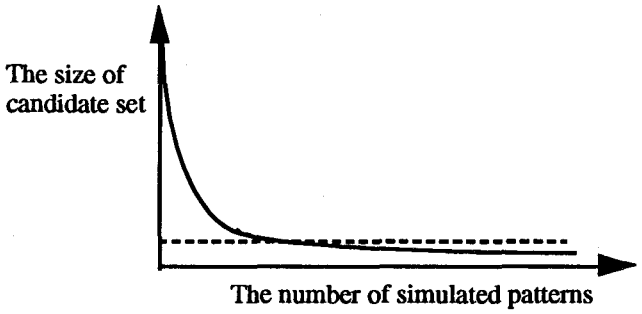


Fig. 11. The size of candidate set vs. the simulated patterns

3.6 Experimental Results: Design Error Location

Dedia, a design error locating tool, based upon the method described above was implemented with C language on a DEC 5000/200 workstation. The inputs of Dedia are the functional-level description(Boolean equations), the don't care set and the net-list representation of the gate-level implementation.

A number of different class-I design errors are artificially introduced into the net-list representations. These different design errors inserted to the circuits are described in Table 4. The first two circuits are real designs in our laboratory, and the others come from the ICCAS85 benchmarks[16]. For each circuit, at least one erroneous output has been found by pre-simulation. Design error location is only applied for the selected erroneous output. Table 5 gives the erroneous output and some statistics for each circuit. All the terms are given with respect to the erroneous output.

For the output O4 of the circuit SIGNET, the difference function Dif = 1. it means that all the combinations of the inputs are the distinguishing patterns. Therefore, the design error must be located at the output O4. Table 6 shows the experimental results for these circuits. The experimental results show that our design error locating method can locate the design error in one signal line for the experimental circuits.

Table 4: The design errors lists

Circuits	The type of design error
LC-CUNIT	A substitution of an OR gate for an AND gate at gate t32
SIGNET	An extra inverter at the output O4
C1355	A missing inverter at the fan-out branch {I20-t644}
C432	A substitution of an AND gate for a NOR gate at gate t151
C880	A missing fan-in t161 at gate t171 with fan-ins t161 and t165
C1908	An extra inverter at the fan-in t108 of gate t65

Table 5: Statistics of benchmark circuits

Circuits	Erroneous output	No. of signal lines	No. of dis. patterns
LC-CUNIT	O14	80	6
SIGNET	O4	198	*
C1355	O1	1154	1048576
C432	O5	269	72579
C880	O9	118	512
C1908	O9	1035	248584

Table 6: Experimental results for design error location

	Fault simulation		Final results	
Circuits	Simulated patterns	Candidates	CPU time(s)	Candidates
LC-CUNIT	6	20	0.40	t32
SIGNET	*	*	0.06	O4
C1355	1087	14	35.82	[I20-t644]
C432	168	5	10.0	t151
C880	85	3	0.04	t165/t171
C1908	212	6	24.2	t108

4 Conclusions

For an incompletely specified function, the method to compute the corresponding 3-terminal BDD has been described. We also showed one application of 3-terminal BDDs for the verification of two incomplete functions. For locating the sources of design errors that exist in a gate-level circuit, a conditional stuck-at fault model was proposed, and a design error location method, based on fault simulation and released pattern generation, has been described. The experimental results showed that our method is efficient to locate the single design error in incorrect circuits.

References

1. M.S. Abadir, J. Ferguson, T.E. Kirkland: Logic Design Verification via Test Generation. IEEE Trans. on CAD Vol. 7, No. 1, Jan. 1988, pp. 138 - 148
2. G.D. Hachtel, R.M. Jacoby: Verification Algorithms for VLSI Synthesis. IEEE Trans. on CAD, Vol. 7, No. 5, MAY, 1988, pp. 616 - 640
3. R.E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Computers Vol C-35, No. 8, Aug. 1986, pp. 677 - 691
4. S. Malik, A.R. Wang, R.K. Brayton, A. Sangiovanni-Vincentelli: Logic Verification Using Binary Decision Diagram in a Logic Synthesis Environment. Proc. ICCAD-88, 1988, pp. 6 - 9
5. S. Minato, N. Ishiura: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. Proc. 27th DAC, 1990, pp. 52 - 57
6. G. Odawara, M. Tomita, O. Okuzawa, T. Ohta, Z. Zhuang: A Logic Verifier Based on Boolean Comparison. Proc. 23rd DAC, 1986, pp. 208 - 214
7. K.A. Tamura: Locating Functional Errors in Logic Circuits. Proc. 26th DAC, 1989 pp. 185-191
8. V. Pitchumani, P. Mayor, N. Radia: A System for Fault Diagnosis and Simulation of VHDL Descriptions. Proc. 28th DAC, 1991, pp. 144 - 150
9. S. Y. Kuo: Locating Logic Design Errors via Test Generation and Don't-Care Propagation. Proc. 1st EURO-DAC, 1992, pp. 466 - 471
10. M. Fujita, T. Kakuda, Y. Matsunage: Redesign and Automatic Error Correction of Combinational Circuits. Proc. IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis, 1990, pp. 253 - 262
11. K.S. Brace, R. L. Rudell, R.E. Bryant: Efficient Implementation of a BDD Package. Proc. 27th DAC, 1990, pp. 40 - 45
12. N. Calazans, Q. Zhang, R. Jacobi, B. Yernaux, A.M. Trullemans: Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams. Proc. EDAC-92, 1992, pp. 452 - 457
13. S. J. Hong: Fault simulation Strategy for Combinational Logic Networks. 8th symp. Fault-tolerant Computing(FTCS-8), 1978, pp. 96 - 99
14. M. Abramovici, et al: Critical Path Tracing: An Alternative to Fault Simulation. IEEE Design & Test of Computers, Vol. 1, No. 1, Feb., 1984, pp. 83 - 93
15.P. Goel: An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. IEEE Trans. on Computer, Vol. C-30, Mar., 1981, pp. 215 - 222
16. F. Brglez, H. Fujiwara: A Neutral Netlist of 10 Combinational Circuits. Proc. ICCAS85