Chapter 2

DEFECTS, FAULTS, FAULT MODELS

Abstract:

Key words:

1. CLASSIFICATION OF FAULTS

1.1 Defects, faults, errors

1.2 Stuck-at fault properties

One of the most important concerns when generating tests or simulating faults in digital systems is the complexity – the huge number of faults we have to work with. To overcome this problem we should reduce the total number of faults to be processed in test generation. To do that we need to know better different properties of faults such as detectability, redundancy, equivalence, dominance, a.o. Each of these properties will help us to select only essential faults and not consider other faults classified as nonessential.

Let $y(x)$ be the logic function of a combinational circuit C, where x is an input vector and $y(x)$ denotes the mapping realized by C. The presence of a fault f transforms C into a new faulty circuit C^f with a function $y^f(x)$. Each input vector t can be regarded as a test, and a sequence of input vectors $T = (t_1, t_2,\ldots, t_n)$ is a test sequence. A test t detects a fault f if $y(t) \neq y^f(t)$.

Fault redundancy. We call a fault f **detectable** if there exists a test t that detects f; otherwise, we call f **undetectable**.

A combinational circuit that contains an undetectable stuck-at-fault (SAF) is said to be **redundant**, since such a circuit can always be simplified by removing at least one gate or gate input.

For example, suppose that a *s-a*-1 fault on an input of an AND gate is undetectable. This means that the function of the gate does not change in the presence of the fault, and we can permanently place constant 1 on that input. But, an n-input AND with a constant 1 value on one input is logically equivalent to the (n-1)-input AND obtained by removing the gate input with the constant signal. Similarly, if an AND input *s-a*-0 is undetectable, the AND gate can be removed and replaced by a signal of logic 0 value. Other simplification rules can be found in [1].

Redundant faults cause a real trouble in test generation. A test generation is a procedure where a test should be searched among all possible input patterns or sequences. For non-redundant faults we usually find the test quickly by tracing only a small part of the search space. For redundant faults we have to go through the whole huge space of possible patterns.

Hence, if we can exclude redundant faults from test generation we can significantly increase the test generation speed.

Fault equivalence. Two faults f and g are called **functionally equivalent** if $y^f(x) \equiv y^g(x)$.

A test t is said to distinguish between two faults f and g if $y^f(x) \neq y^g(x)$; such faults are **distinguishable**. There is no test that can distinguish between two functionally equivalent faults.

The relation of functional equivalence partitions the set of all possible faults into functional **equivalence classes.** For test generation it is sufficient to consider only one representative fault from every equivalence class.

With any n-input gate we can associate 2(n + 1) single stuck faults. For a NAND gate all the input *s-a*-0 faults and the output *s-a*-1 fault are functionally equivalent. These equivalent faults can be represented by a single fault in the test generation process. Hence, for test generation for an n-input ŃAND gate (n>1) we need to consider only n+2 single stuck faults.

This type of reduction of the set of faults based on equivalence relations is called **equivalent fault collapsing**.

If, in addition to **fault detection**, the goal of testing is **fault location** as well, we need to apply a test that not only detects the detectable faults but also distinguishes among them as much as possible. A **complete fault location test** distinguishes between every pair of distinguishable faults in a circuit.

A complete fault location test can diagnose a fault to within a functional equivalence class. This is the maximal diagnostic resolution that can be achieved.

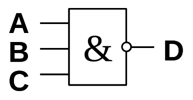
Fault dominance. If the objective of a testing is limited to fault detection only, then in addition to fault equivalence, another fault relation called **fault dominance** can be used to reduce the number of faults that must be considered.

Let T_g be the set of all test vectors that detect a fault g. A fault f dominates the fault g if f and g are functionally equivalent under T_g.

If f dominates g, then any test t that detects g will also detect f. Therefore, for fault detection it is unnecessary to consider the dominating fault f, since by deriving a test for g we automatically obtain a test that detects f as well.

Fault collapsing. The fault equivalence and fault dominance properties can be used for minimizing the whole set of faults to be considered in test generation and fault simulation.

Consider a 3-input NAND gate in *Figure 2-1*. Stuck-at-0 faults on inputs A/0, B/0, C/0 and stuck-at-1 fault D/1 on the output form an equivalent class of faults. On the other hand, the fault D/0 dominates faults A/1, B/1 and C/1. Any of the faults in the equivalence class can be chosen as the representative fault whereas all other faults can be excluded from consideration. Regarding the dominance classes only the dominating fault D/0 can be excluded.



A	B	C	D	Fault class	
1	1	1	0	A/0, B/0, C/0, D/1	Equivalence class
0	1	1	1	A/1, D/0	
1	0	1	1	B/1, D/0	Dominance classes
1	1	0	1	C/1, D/0	

Figure 2-1. Fault collapsing for a NAND gate

Consider now a simple combinational circuit in *Figure 2-2*. In the case of a circuit we can use the both rules of **fault collapsing** by combining them in a proper way. For example, for the stuck-at-1 fault on the output we first choose stuck-at-0 on the connection line between gates as the representative fault, and then we collapse this fault by using the dominance rule. As the result, we see that all the faults on the current path of the circuit are collapsed except stuck-at-1 fault on the input. In a similar case for stuck-at-0 fault on the output we see that all the faults on the current path will be collapsed except for stuck-at-0 fault on the input.

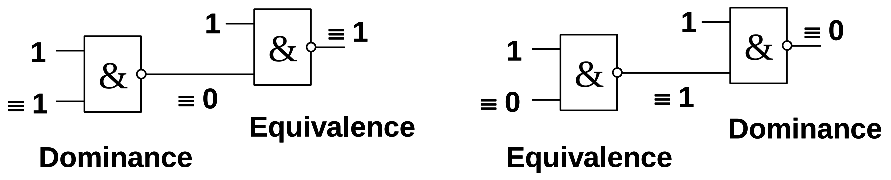


Figure 2-2. Fault collapsing for a combinational circuit

By generalizing this result in inductive way, we can easily show that for tree-like combinational circuits, only the stuck-at-faults on inputs are the essential faults for test generation and fault simulation.

2. FUNCTIONAL FAULT MODEL

The efficiency of test generation is highly depending on the system description and fault models used.

It has been shown that high SAF coverage cannot guarantee high quality of testing, for example, for CMOS integrated circuits [2]. The reason is that the SAF model ignores the actual behavior of CMOS circuits, and does not adequately represent the majority of real IC defects and failure mechanisms which often do not manifest themselves as stuck-at faults. To handle physical defects in fault simulation, we still need logic fault models for the following reasons: to reduce the complexity of simulation (many physical defects may be modelled by the same logic fault), a single logic fault model may be applicable to many technologies, logic fault tests may be used for physical defects whose effect is not well understood. The most important reason for logical modelling of physical defects is to get a possibility for moving from the lower physical level to the higher logic level which has less complexity.

In this subchapter, an approach to modelling physical defects by generic Boolean differential equations with the goal to map them from the physical level to the logic level is presented. Different transistor level faults will be analysed to show that this way of mapping is general and feasible enough. A new fault model is defined on that basis, called **functional fault model**. It is also shown how the functional fault model can be treated as a uniform interface for mapping faults from a given arbitrary level of abstraction to the next higher level in test generation processes.

2.1 Fault modelling with Boolean differential equations

Consider a Boolean function $y = f(x_1, x_2, ..., x_n)$ implemented by an embedded component (complex gate) G in a circuit. Introduce a Boolean variable d for representing a given physical defect in the component, which may affect the value y by converting the Boolean function f into another function

$$y = f^{d}(x_1, x_2, ..., x_n)$$

where, in fact, some of the arguments x_i can fall out, thus simplifying the function because of the defect.

Let us introduce a **generic parametric function**

$$y^* = f^*(x_1, x_2, ..., x_n d) = \overline{d} f \vee df^{d} \qquad (2\text{-}1)$$

for the component G as a function of the defect variable d, which describes the behavior of the component simultaneously for both fault-free and faulty cases. For the faulty case, the value of the defect variable d as a parameter is equal to 1, and for the fault-free case $d = 0$. In other words, $y^* = f^{d}$ if $d = 1$, and $y^* = f$ if $d = 0$.

The solutions of the **Boolean differential equation**

$$W^{d} = \frac{\partial y^*}{\partial d} = 1 \qquad (2\text{-}2)$$

describe the conditions which activate the defect d on a line y. The **parametric modelling** of a given defect d by equations (2-1) and (2-2) allow us to use the constraints $W^d = 1$, either in defect-oriented fault simulation, to check whether the condition (2-2) is fulfilled, or in defect-oriented test generation, to solve the equation (2-2) when the defect d should be activated and tested.

To find W^d for a given defect d we have to create the corresponding logic expression for the faulty function f^d, either by logical reasoning or by carrying out defect simulation directly, or by carrying out real experiments to learn the physical behavior of different defects.

Example 2-1. Let us have a transistor circuit as in *Figure 2-3* which implements the function

$$y = x_1 x_2 x_3 \vee x_4 x_5 \, .$$

A short defect as shown in *Figure 2-3* changes the function of the circuit as follows:

$$y^d = (x_1 \vee x_4)(x_2 x_3 \vee x_5).$$

Using the defect variable *d* for the short, we can create a generic differential equation for this defect and simplify the created expression as follows:

$$\frac{\partial y^*}{\partial d} = \frac{\partial((x_1 x_2 x_3 \vee x_4 x_5)\bar{d} \vee (x_1 \vee x_4)(x_2 x_3 \vee x_5)d)}{\partial d} = .$$
$$= x_1 \bar{x}_2 \bar{x}_4 x_5 \vee x_1 \bar{x}_3 \bar{x}_4 x_5 \vee \bar{x}_1 x_2 x_3 x_4 \bar{x}_5$$

From the equation three possible solutions follow: T = {10x01, 1x001, 01110}. Each of them can be used as a test pattern for the given short. On this contra-example, it is easy to show the inadequacy of the stuck-at fault (SAF) model for testing the transistor level faults. For example, the set of five test patterns 1110x, 0xx11, 01101, 10110, 11010 which test all the stuck-at faults in the circuit does not include any of the possible test solutions for detecting the short from the set T.

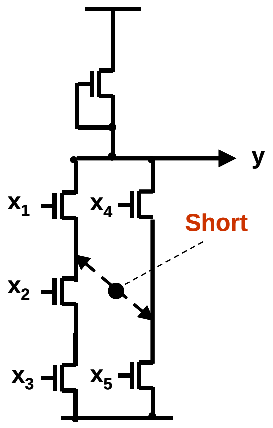


Figure 2-3. Transistor circuit with a short

Note that for the same purposes of finding the test for the defect *d* we also could solve the equation

$$f \oplus f^d =$$
$$= (x_1 x_2 x_3 \vee x_4 x_5) \oplus (x_1 \vee x_4)(x_2 x_3 \vee x_5) = 1$$

directly without introducing the defect variable *d*. However, solving the equation (2-2) will be much easier because of simplification possibilities resulting from specific properties of Boolean differentials [3].

2.2 Mapping physical transistor defects to logic level

The described method represents a general approach to map an arbitrary physical defect onto a higher (in this case, logic) level. By the described approach an arbitrary physical defect in a component can be represented by a logical constraint $W^d = 1$ to be fulfilled for activating the defect (*Figure 2-4*).

The event of erroneous value on the output y of a functional component can be described as $dy = 1$, where dy means **Boolean differential**. A **functional fault** representing a defect d can be described as a couple (dy, W^d). At the presence of a physical level defect d, we will have a higher level erroneous signal $dy = 1$ if the condition $W^d = 1$ is fulfilled.

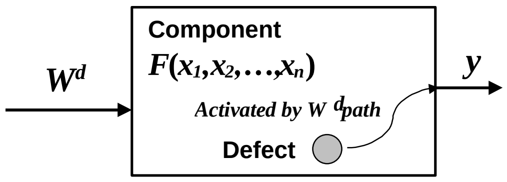
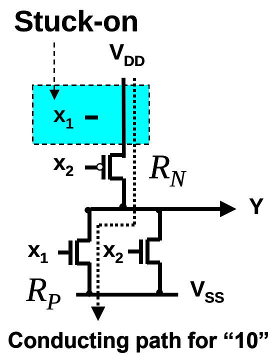


Figure 2-4. Transistor circuit with a short

From another point of view, the equation (2-2) can be interpreted as a mapping of a physical defect d from the transistor level to the logic level as an erroneous change of a logic value $dy = 1$ by means of activiting the physical defect d with condition $W^d = 1$.

The following examples will show the feasibility of using Boolean differential equations for mapping faults from physical transistor level to logic level.

Example 2-2. Transistor level **stuck-on faults**. The behavior of the transistor level NOR gate depicted in *Figure 2-5* cannot be described strictly logically. The input vector "10" produces a conducting path from VDD to VSS, and the corresponding voltage at the output node Y will not be equal to either VDD or VSS but will instead be a function of the voltage divider formed by the channel resistances of the conducting transistors:



x_1	x_2	y	y^d
0	0	1	1
0	1	0	0
1	0	0	V_Y/I_{DDQ}
1	1	0	0

Figure 2-5. Stuck-on fault in the transistor NOR gate

$$V_Y = \frac{V_{DD} R_P}{(R_P + R_N)}.$$

Depending on the ratio of these resistances along with the switching thresholds of the gates being driven by the output of the faulty gate *y*, the output voltage of the faulty gate may or may not be detected at a primary output. Denote by *Z* this ambiguous value on the gate output. The faulty function of the gate can be represented as follows:

$$y^d = \overline{x_1}\,\overline{x_2} \vee x_1 \overline{x_2} Z.$$

If $x_1 \overline{x_2} = 1$ then $y^d = Z$. Using now the expressions (1) and (2) we get:

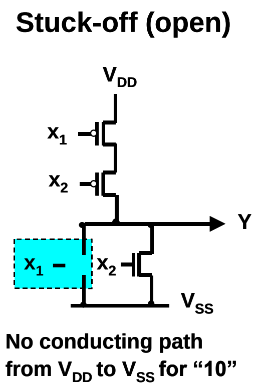
$$y^* = \overline{d}(\overline{x_1 \vee x_2}) \vee d(\overline{x_1}\,\overline{x_2} \vee x_1 \overline{x_2} Z)$$

$$W^d = \partial y^* / \partial d = x_1 \overline{x_2} Z = 1.$$

Consequently, the condition to activate the defect is $x_1 = 1, x_2 = 0$.

<u>*Example 2-3.*</u> Transistor level **stuck-open faults**. For the transistor stuck-open fault of the NOR gate in

Figure 2-6, there will be no path from the output node to either VDD or VSS for some input patterns. As a result, the output node will retain its previous logic value. This creates a situation where a combinational logic



x_1	x_2	y	y^d
0	0	1	1
0	1	0	0
1	0	0	Y'
1	1	0	0

gate behaves like a dynamic memory element.

Figure 2-6. Stuck-off (open) fault in the transistor NOR gate

The faulty function of the gate is: $y^d = \overline{x_1}\,\overline{x_2} \vee x_1\overline{x_2}y'$ where *y'* corresponds to the output value stored at the output of the faulty gate from the previous clock cycle. Using now the expressions (2-1) and (2-2) we have:

$$y* = \overline{d}(\overline{x_1 \vee x_2}) \vee d(\overline{x_1}\,\overline{x_2} \vee x_1\overline{x_2}y') =$$
$$= \overline{x_2}(\overline{x_1} \vee dy')$$
$$W^d = \partial y*/\partial d = x_1\overline{x_2}y' = 1.$$

It follows now that the condition to activate the defect is $x_1 = 1, x_2 = 0, y' = 1$. In other words, for testing the fault we need a test sequence of two patterns: "00" to get the value 1 on the output to be stored, and then "11".

2.3 Mapping interconnection defects to logic level

Consider now a component *C* representing a Boolean function $y = f(x_1, x_2, \ldots, x_n)$ embedded in a surrounding network given by a subset of lines $E_c = \{x_{n+1}, \ldots, x_p\}$. Introduce the same Boolean variable *d* for representing physical defects in the subcircuit (C,E_c), given by the block *C* with its neighborhood E_c, which may affect the value *y*. Let the defect *d* convert the Boolean function *f* into another function

$$y = f^d(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots x_p).$$

Let us introduce for modelling physical defects related to the subcircuit (C,E_c), a generic parametric function

$$y* = f*(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_p, d) =$$
$$(\overline{d} \wedge f) \vee (d \wedge f^d)$$

as a function of a defect variable *d*, which describes the behavior of the subcircuit for both fault-free and faulty cases simultaneously. For the faulty case the value of the defect variable *d* as a parameter is equal to 1, and for the fault-free case $d = 0$. In other words, $y* = f^d$ if $d = 1$, and $y* = f$ if $d = 0$. The solutions of the Boolean differential equation (2-2) describe the conditions which activate the defect *d* on a line *y*.

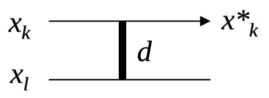


Figure 2-7. A bridging fault

<u>*Example 2-4.*</u> A short between two lines x_k and x_l in the circuit in *Figure 2-7*. The faulty function of x_k in the case of the defect d in accordance to the wired-AND fault model can be represented as $x_k{}^d = x_k x_l$. Introduce now a generic parametric function

$$x_k{*} = \overline{d}x_k \vee dx_k{}^d = \overline{d}x_k \vee dx_k x_l = x_k(\overline{d} \vee x_l)$$

as a function $x_k{*} = f(x_k, x_l, d)$ of a defect variable d, which describes the behavior of the interconnection network for both fault-free and faulty cases simultaneously. The solution of the Boolean differential equation

$$W^d = \partial x_k{*} / \partial d = x_k \overline{x_l}$$

describes the conditions (constraints) which activate the fault d on a line x_k (*Figure 2-7*). The condition $W^d = x_k \overline{x_l} = 1$ means that in order to detect the short between lines x_k and x_l on x_k we have to assign the value 1 to x_k and the value 0 to x_l .

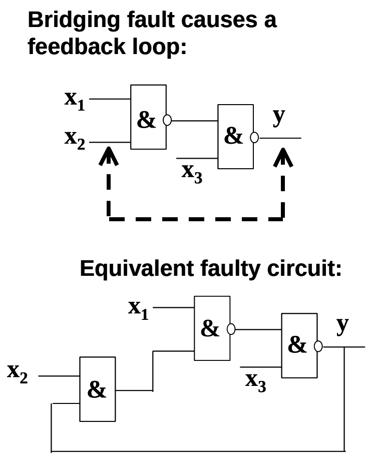

Bridging fault causes a feedback loop:
x1
x2
&
&
x3
y
Equivalent faulty circuit:
x1
x2
&
&
&
x3
y

Figure 2-8. A Bridging fault with feedback loop

<u>*Example 2-5.*</u> A short between two lines x_k and x_l in the circuit which creates a feedback loop. A circuit with such a loop and its equivalent faulty circuit corresponding to the wired-AND fault model is shown in *Figure 2-8*. The generic parametric function for describing the behavior of the circuit for both fault-free and faulty cases simultaneously has the following form:

$$\begin{gathered} y^* = \overline{d}(x_1 x_2 \vee \overline{x_3}) \vee d(x_1 x_2 y \vee \overline{x_3}) = \\ x_1 x_2 (\overline{d} \vee y')\overline{x_3} \end{gathered} .$$

The solution of the Boolean differential equation

$$W^d = \partial y^* / \partial d = x_1 x_2 x_3 \overline{y}' = 1$$

describes the conditions (constraints) which activate the fault *d* on a line *y* (Figure 2-8). The apostrophe at *y* means that the value of *y* belongs to the previous time moment. The condition

$$W^d = x_1 x_2 x_3 \overline{y}' = 1$$

means that we need a sequence of two patterns for testing the short. First, we have to set the value $y = 0$ (for example, by assigning $x_3 = 0$), then we have to apply the pattern $x_1 = 1$, $x_2 = 1$, $x_3 = 1$.

We can see from the example that in the general case the constraints for activating a fault may be spread over different time moments, and represent sequences of patterns.

We also see that the method for describing faults by generic Boolean differential equations allows us directly to attack the problem of testing so called ”**sequential faults**” which either convert combinational circuits into sequential ones or increase the number of states in sequential circuits. Test generators which are able to work with such faults are missing.

The functional fault model described as a couple (dy, W^d) can be regarded first as a method of mapping arbitrary physical defects onto the logic level, and second as a universal method of fault modeling in hierarchical approaches to test generation and fault simulation.

The conditions W^d for activating defects *d* can be used as constraints at the higher (logic or register transfer) levels either for fault simulation or for test pattern generation without paying attention to the physical origins of defects.

2.4 Hierarchical representing of faults

The method of defining faults by logic conditions W^d allows us to unify the diagnostic modelling of components of a circuit (or system) without going into structural details of components and into the diagnostic simulation of interconnection network of components. In both cases, the condition $W^d = 1$ describes how a lower level fault *d* (either a defect in a component or a defect in a network) should be activated at a higher level to a given node in a circuit (or system). The condition $W^d = 1$ can be used both in fault simulation and in test generation.

Consider a node k in a circuit in *Figure 2-9* as the output of a module M_k, and represented by a variable x_k. Associate with the node k a set of faults R_k = $R^F_k \cup R^S_k$ where R^F_k is the subset of faults in the module M_k, and R^S_k is a subset of structural faults (defects) in the "network neighborhood" of M_k. Denote by W^d the condition when the fault $d \in R_k$ will change the value of x_k. Denote by W^F_k the set of conditions W^d activating the defects $d \in R^F_k$ in components and by W^S_k the set of conditions W^d activating the defects $d \in R^S_k$ in the interconnection network.

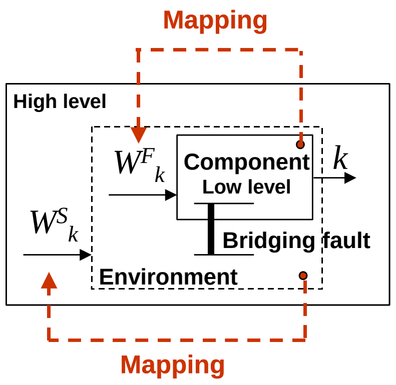


Figure 2-9. Mapping faults from lower level to higher level

By using the sets of conditions W^F_k and W^S_k we can set up a mapping of faults from a lower level to a higher level for test generation purposes, and also in opposite direction, from a higher level to a lower level for fault simulation or fault diagnosis purposes.

In test generation, to map the lower level fault $d \in R_k$ to the higher level variable x_k, a solution of the equation $W^d = 1$ is needed. In other words, if the condition $W^d = 1$ is fulfilled then the presence of the defect $d \in R_k$ will change the value of the variable x_k.

In fault simulation (or in fault diagnosis) an erroneous value of x_k (denoted by a Boolean differential $dx_k = 1$) can be formally explained by implication

$$dx_k \rightarrow d_1 W^{d1} \vee d_2 W^{d2} \vee ... \vee d_n W^{dn} \qquad (2\text{-}3)$$

where for $j = 1,2,\ldots n$: $d_j \in R_k$. If the condition $W^{dj} = 1$ is fulfilled, the higher level error $dx_k = 1$ implies the lower level defect d_j.

For hierarchical testing purposes we should construct for each module M_k of the circuit a list of faults R_k with logical conditions W^d for each fault $d \in R_k$. The set of conditions W^F_k for the functional faults $d \in R^F_k$ of the module can be found by low level test generation for the defects in the module. The set of conditions W^S_k for the structural faults $d \in R^S_k$ in the environment of the module can be found by Boolean differential analysis of generic fault-free/faulty functions as explained in previous Sections 2-1, 2-2 and 2-3.

In *Figure 2-10*, a **hierarchical test** conception based on parametric fault modelling and functional fault model for a 3-level system is illustrated. In the functional approach, only the information about the functional behaviour is used. In the structural approach, tests are targeted to detect the faults in the networked components and in the network interconnections.

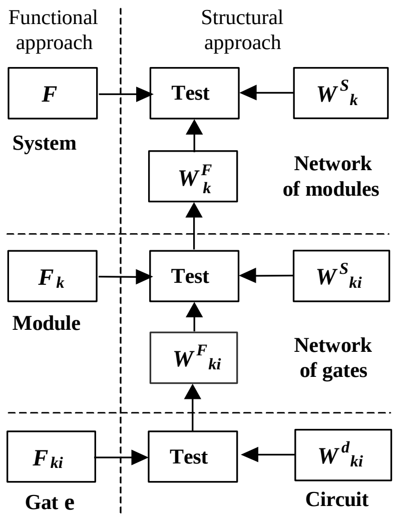


Figure 2-10. Hierarchical fault representing

Consider a task of defect oriented fault simulation in a system which is represented at three levels: register transfer, gate and defect levels.

Formally, if Y is the system variable representing an observable point (a register) of the system, y_M is an output variable of a logic level module and y_G is the output of a logic gate with a physical defect d, then the condition to detect the defect d on the observable test point Y of the system is

$$W = \partial Y/\partial y_M \wedge \partial y_M/\partial y_G \wedge W^d = 1,$$

where $\partial Y/\partial y_M$ means the fault propagation condition calculated by high-level modeling, $\partial y_M/\partial y_G$ is the fault propagation condition (Boolean derivative) calculated by gate-level modeling, and W^d is the functional fault condition calculated for physical defects from 2-2 during the gate pre-analysis.

3. DEFECT MODELLING

4. HIGH-LEVEL FAULT MODELS

To increase the speed of fault coverage evaluation, high-level (functional or behavioral) fault models have been developed. High-level faults represent the effects of physical defects on the operation of a system represented on the functional or behavioral level. A high-level fault model can be considered good if the tests generated using this model provide a high coverage of stuck-at-faults or physical defects.

The main idea of the high-level fault modelling is to obtain from the high-level description of the system an incorrect version by introducing a fault into the description. This approach is called **model perturbation** [4]. The models can be "perturbed" in certain ways, e.g. by truth-table modification, micro-operation modification, etc. In some or other way, this idea is implemented in different high-level fault models, developed, for example, in a very dedicated way for microprocessors [5,6,7], in a more general way for systems represented in register-transfer languages [8,9] or other hardware description language like VHDL [10,11,12], etc. Some attempts to develop dedicated functional fault models for different data-flow network units like decoders, multiplexers, memories, PLAs, etc. are described in [13].

A high-level fault model can be explicit or implicit [1]. An **explicit model** identifies each fault individually, and every fault in this model will be a target for test generation. An **implicit model** identifies classes of faults with "similar" properties, so that all faults in the same class can be detected by similar procedures. The advantage of an implicit fault model is that it does not require explicit enumeration of faults within a class.

Most of the high-level faults presented in this subchapter can be covered by so called **addressing faults** [1]. Typical examples include: addressing a word in a memory, selecting a register according to a field in the instruction word of a processor, decoding an op-code to determine the instruction to be executed.

The common feature of these schemes is the use of a n-bit address to select one of 2^n possible items. Whenever item i is to be selected, the presence of an addressing fault may lead to: a) selecting no item, b) selecting item j instead of i, c) selecting item j in addition to i. More generally, a set of items $\{j_1, j_2, \dots, j_k\}$ may be selected instead of, or in addition to, i.

An important feature of this fault model is that it forces the test generation process to check whether the intended function is performed and also whether no extraneous operations occur. This fundamental aspect of functional testing is often overlooked by heuristic methods.

4.1 Microprocessor functional fault model

In [5,6] a fault model for various units of the data processing section and the control section of microprocessors was presented. Faults affecting the operation of microprocessor can be divided into the following classes:

- addressing faults affecting the register decoding function;
- addressing faults affecting the instruction decoding and instruction sequencing functions;
- faults in the data-storage function;
- faults in the data-transfer function;
- faults in the data-manipulation function.

Addressing faults affecting the register decoding function

For multiplexers under a fault, for a given source address any of the following events may happen:

F1: No source is selected;

F2: A wrong source is selected;

F3: More than one source is selected and the multiplexer output is either a wired-AND or a wired-OR function of the sources, depending on the technology.

For demultiplexers under a fault, for a given destination address:

F4: No destination is selected;

F5: Instead of, or in addition to the selected correct destination, one or more other destinations are selected.

Faults affecting the instruction decoding and instruction sequencing functions

An instruction *I* can be viewed as a sequence of *micro-instructions*, where every micro-instruction consists of a set of *micro-orders*, which are executed in parallel. Micro-orders represent the elementary data transfer and data manipulation operations.

Addressing faults affecting the execution of an instruction may cause one or more of the following fault effects:

F6: One or more micro-orders are not activated by the micro-instructions of *I*.

F7: Micro-orders are erroneously activated by the micro-instructions of *I*.

F8: A different set of micro-instructions is activated instead of, or in addition to, the micro-instructions of *I*.

This fault model is general, as it allows for partial execution of instructions and for execution of “new” instructions, not present in the instruction set of a microprocessor.

Fault model for data storage function

The data storage facility is usually implemented as a memory. Therefore, here the fault model developed in [7] can be used[1]. Under a fault any of the following situations may happen to the memory cell array:

F9: One or more cells are stuck at 0 or 1;

F10: One or more cells fail to make a $0\rightarrow1$ or $1\rightarrow0$ transitions;

F11: Two or more pairs of cells are coupled; by this we mean a transition from x to y in one cell of the pair, say cell i, changes the state of the other cell, say j, from x to y or from y to x, where $x \in \{0,1\}$, and $y = \overline{x}$.

Fault model for data transfer function

The data-transfer function implements all the data transfers along the buses between the registers and functional units of a microprocessor.

For buses under a fault:

F12: One or more lines can be stuck at 0 or 1;

F13: One or more lines may form a wired-OR or wired-AND function due to shorts or spurious coupling.

Fault model for data manipulation function (F14)

In the case of the data processing functional units, no specific model has not been proposed because the wide range of existing designs would only tend to complicate any general model. It is assumed that a complete test set can be derived for the functional units by some other techniques.

The main disadvantage of the described approach is that only microprocessors are handled and the fault classes defined cannot be extended to cover the general digital systems test problem.

4.2 Register-transfer-level functional fault model

A register-transfer-level (RTL) functional fault model is set up with respect to certain sets of functional faults considered. The set of faults is derived from a fault analysis for all distinct RTL statements of the device under test.

A formal definition of a RTL statement is defined as [4]:

$$K\colon (T,C)\ R_d \leftarrow f(R_{S1}, R_{S2}, \ldots, R_{Sn}),\ \rightarrow N,$$

where K is the RTL statement label, T is the timing, and C is the logic condition to execute this statement, R_d is the destination register, R_{Si} is

[1] Memory faults are profoundly discussed in Chapter 3 in the section devoted to memory test

the *i*-th source register, f is an operation on source registers, $\leftarrow$ represents data transfer, and $\rightarrow N$ represents a jump to statement N.

Based on the above notation, nine categories of functional faults can be identified for the register transfer level as follows:

RT1: label faults denoted by (K/K'), which means that the label K will be changed to K' due to the low-level faults such as SAF, bridging or pattern sensitive faults,

RT2: timing faults (T/T'),

RT3: logic condition faults (C/C'),

RT4: register decoding faults (R_i/R_i'),

RT5: function decoding faults (f/f'),

RT6: control faults ($\rightarrow N$/$\rightarrow N$'),

RT7: data storage faults ((R_i)/(R_i)'), which means that the content of the register R is changed from (R) to (R)' due to the low-level faults,

RT8: data transfer faults ($\leftarrow$/$\leftarrow$'), which means that the fault occurs in the transfer path between the sources and the destination,

RT9: data manipulation (function execution) faults ((f)/(f)', which means the operation execution fault – the operation f is executed, but the result of the operation is wrong.

This set of derived functional faults is comprehensive because the internal functional behavior of any digital system can be described by a sequence of RTL statements. Functional fault dominance and fault collapse analysis may be applied to shrink the size of the fault set.

It can be justified that the above functional faults are the manifestation of physical faults (e.g., stuck at faults, bridging faults, etc.) at circuit level into functional faults at the RTL level. Using the above functional fault model, the RTL technique can be comprehensively developed to consider more practical functional faults.

4.3 Fault Modelling by Decision Diagrams

All the approaches described above lead to using very specific fault models dedicated to special classes of systems or components, and hence, to different mathematics and test generation procedures for each fault model. The diversity of fault types makes it difficult to develop test generation algorithms with possibility to treat all faults by standard procedures as it is done for stuck-at faults in the gate-level case. Test generation based on a lot of different types of fault models will be more complicated compared to the case when only one generic fault model is used. Such a general and uniform fault model can be defined when a digital system will be represented by **decision diagrams** [18,19].

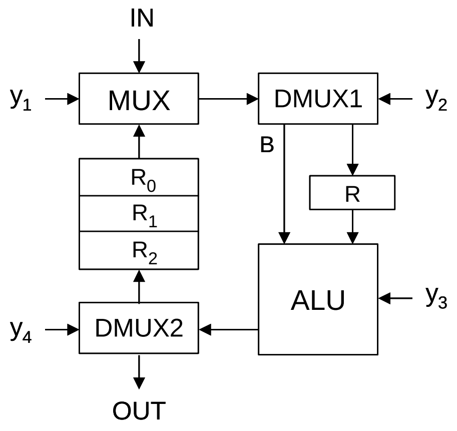


Figure 2-11. A data-path of a digital system

Table 2-1. Behaviour of the components in Figure 2-11.

MUX		DMUX1		ALU		DMUX2	
y_1	Function	y_2	Function	y_3	Function	y_4	Function
0	MUX = R_0	0	B = MUX	0	ALU = B	0	R_0 = ALU
1	MUX = R_1	1	R = MUX	1	ALU = B + R	1	R_1 = ALU
2	MUX = R_2	2	∅	2	ALU = B + 1	2	R_2 = ALU
3	∅	3		3	∅	3	OUT = ALU

Decision diagrams. Consider a digital system represented in *Figure 2-11.* with functionalities of its components in *Table 2-1*. The behavior of the register R_0 is represented by the decision diagram in *Figure 2-12.*

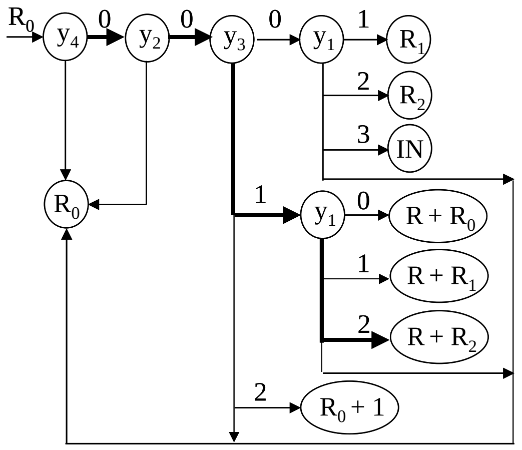


Figure 2-12. Decision Diagram for the register R_0 in *Figure 2-11.*

The **non-terminal nodes** of the DDs are labeled by control variables, whereas the **terminal nodes** are labeled by functional expressions or data variables. The edges of non-terminal nodes are marked by the values of the node variables. Missing of the value on the edge means "all other values".

The data-path is controlled by microinstructions consisting of 4 fields: y_1 - for decoding of the source register, y_2 - for loading data to the buffer register R, y_3 - for controlling ALU (selecting a microoperation), and y_4 - for selecting the destination register.

The graph represents the functional behavior of the register R_0 for a given clock cycle whereas the terminal nodes show how the new content of the register is calculated. To calculate a new value for the register in the current clock cycle means to traverse a path in the DD from the root node to a terminal node. The values of the node variables for the current microinstruction decide the directions of tracing the nodes. As an example, the calculation of the new content of R_0 for the microinstruction $y_1\, y_2\, y_3\, y_4 =$ 2010 is illustrated by bold lines in *Figure 2-12* which corresponds to a microperation $R_0 = R + R_2$.

Each path in a DD describes the behavior of the system in a specific mode of operation. The faults having effect on the behavior can be associated with nodes along the given path whereas each node represents a structural unit or subcircuit of the system. A fault of the node causes an incorrect leaving the path activated by a test.

Fault model for Decision Diagrams

Using DDs it is possible to introduce a simple generic fault model of nodes for digital systems represented at different levels in a similar way as the logic level stuck-at-1 and stuck-at-0 faults are related to Boolean variables in corresponding logical expressions.

The fault model for DDs can be represented as a set of 3 different fault types D1, D2 and D3 described as follows:

D1 - the output edge of a node is always activated;

D2 - the edge of a node is always broken;

D3 - instead of the activated edge, another edge or a set of edges is activated.

This model can be easily interpreted for non-terminal nodes of the decision diagram. It can be used also for terminal nodes, however it will be not very practical for these nodes because of the high number of possible values of the data variables. The faults related to terminal nodes of DDs can be managed hierarchically by the functional fault model as discussed in Section 2.2.

Different fault models for different representation levels of digital systems can be covered by this uniform node fault model defined for DDs.

The physical meaning of faults associated with a particular node depends on the meaning of the node.

For example, the fault model of nodes labeled by Boolean variables covers the classical stuck-at fault model in gate-level representations. The fault model for non-terminal nodes represents uniformly decoding faults, instruction decoding or sequencing faults of microprocessors [5,6,7], label, timing, condition, register, function or control decoding faults in RTL models [8,9], case construction faults in procedural models of systems [10,11,12], or simply the functional faults of decoders, multiplexers and demultiplexers [13].

Relationships between different fault models

In *Table 2-2.* the correspondence of the DD-based fault model to RT-level and microprocessor fault classes discussed in Sections 2.4.1 and 2.4.2 is shown.

Table 2-2. RT level and microprocessor faults covered by DD-model

DD-model faults	RT level faults	Microprocessor faults	
		Instruction level	Microinstruction level
D1	RT1-RT6	F1,	F1,F6
D2		F2,F4	F2,F4
D3		F3,F5	F3,F5,F7,F8
Terminal nodes	RT7-RT9	F9-F13	F9-F13

For RTL faults the classes RT2-RT5 correspond directly to the faults of non-terminal nodes in DDs, which easily can be interpreted as timing and logic conditions or as decoding of registers or operations. For example, in *Figure 2-12,* the node y_1 represents source register decoding, y_3 represents (micro) operation decoding, y_2 and y_4 can be easily interpreted as a condition (timing or logic).

The fault classes RT1 (label faults) and RT6 (control faults) refer to the errors in the control unit which can be also represented as a decision diagram. An example of a FSM state transition and output table with the DD of this FSM is shown in *Figure 2-13*. The graph represents a vector function $q.Y = \delta(x, q').\lambda(x, q')$ of the FSM, where x – is the Boolean input variable, q' is the current state variable, q is the next state variable, and Y is the output variable (microinstruction). The fault classes RT1 and RT6 are covered by the faults of the node *q*' in the DD.

State	Logic condition	Next state	Micro-instruction
q'		q	
0		1	∅
1	x = 0	2	Y1
1	x = 1	3	Y2
2		4	Y3
3		4	Y4
4		0	Y5

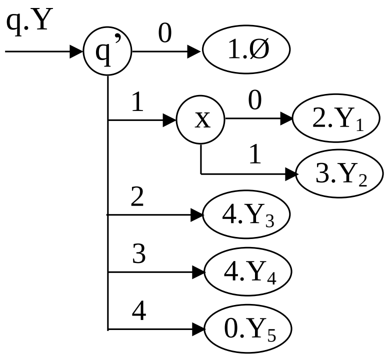


Figure 2-13. Finite State Machine and its Decision Diagram

The correspondence of faults in the terminal nodes of the DD in *Figure 2-12* to the fault classes RT7, RT8 and RT9 is shown in *Table 2-3*.

Table 2-3. Relationship between RT fault classes and DD faults

RTL faults	Terminal nodes in the DD
RT7	R_0, R_1, R_2
RT8	R_1, R_2, IN
RT9	$R + R_0$, $R + R_1$, $R + R_2$, $R_0 + 1$

In *Figure 2-14* a simple instruction set of 10 instructions of a hypothetical microprocessor and the corresponding DD representing the behavior of the register *A* of the microprocessor are given. The microprocessor fault classes are related to the faults of nodes in the DD.

I_1: MVI A,D	**A = IN**
I_2: MOV R,A	**R = A**
I_3: MOV M,R	**OUT = R**
I_4: MOV M,A	**OUT = A**
I_5: MOV R,M	**R = IN**
I_6: MOV A,M	**A = IN**
I_7: ADD R	**A = A + R**
I_8: ORA R	**A = A ∨ R**
I_9: ANA R	**A = A ∧ R**
I_{10}:CMA A,D	**A = ¬ A**

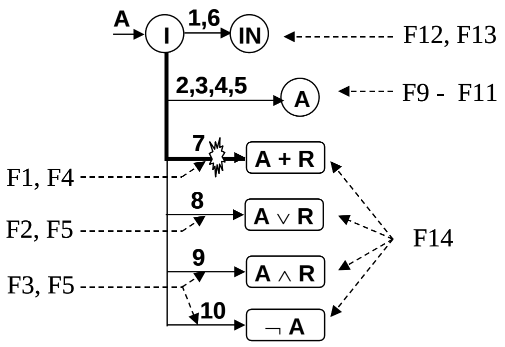


Figure 2-14. Microprocessor instruction set and DD-model

Assume that the edge 7 of the internal node labelled by the instruction variable I is activated. The faults at the node I cover the microprocessor fault classes F1 - F5. For example, the fault classes F1 (no sorce is selected) and F4 (no destination is selected) correspond to the broken edge 7. The fault classes F2 (wrong source is selected) and F5 (wrong destination is selected) correspond to the activated edge 8. The fault classes F3 or F5 (more than one source or destination is selected) correspond to the activated edges 9 and 10. The faults of the terminal node *IN* cover the fault classes of buses F12 and F13, and the faults of the node *A* cover the data storage fault classes F9 - F11. The faults of other terminal nodes belong to the general data manipulation fault class F14 of microprocessors.

The fault classes F6 - F8 correspond to the microinstruction level, and can be discussed on the example of *Figure 2-12*. All the broken edges in the DD correspond to the fault class F6 (microinstructions are not activated). Erroneously activated edges in the DD correspond to the fault classes F7 and F8.

From above it follows that the fault model defined on DDs can be regarded as a generalization of the classical gate-level stuck-at fault model for higher level representations of digital systems. The SAF model is defined for Boolean variables whereas the generalized DD fault model is defined for the nodes of DDs as a high-level model of digital systems.

REFERENCES

1. Abramovici M., Breuer M.A., Friedman A.D. Digital Systems Testing & Testable Designs. *Computer Science Press*, 1995, 653 p.
2. Maly W., Shen J.P., and Ferguson J. System. Characterization of Physical Defects for Fault Analysis of MOS IC Cells. *Proc. Int. Test Conf.*, 1984, pp. 390-399.
3. Thayse A. Boolean Calculus of Differences. Springer Verlag, 1981.
4. Gupta A.K., Armstrong J.R. Functional Fault modelling and Simulation for VLSI Devices. 22nd Design Automation Conference, 1985, pp.720-726.
5. Thatte S.M., Abraham J.A.. Test Generation for Microprocessors, IEEE Trans. On Computers, Vol. C-29, No. 6, pp.429-441, June 1980.
6. Brahme D., Abraham J.A. Functional Testing of Microprocessors. IEEE Trans. On Computers, Vol. C-33, No.6, pp.475-485, June 1984.
7. Thatte S.M., Abraham J.A. Testing of Semiconductor Random Access Memories. Proc. of 7th Int. Symp. on Fault-Tolerant Computing, Los Angeles, June 1977, pp. 81-87.

8. Su S.Y.H., Lin T. Functional Testing Techniques for Digital LSI/VLSI Systems. 21st Design Automation Conference, 1984, pp.517-528.
9. Shen L., Su S.Y.H. A Functional Testing Method for Microprocessors. IEEE Transactions on Computers, Vol.37, No. 10, 1988, pp.1288-1293.
10. Ward P.C., Armstrong J.R. Behavioral Fault Simulation in VHDL. 27th ACM/IEEE Design Automation Conference, 1990, pp.587-593.
11. Ghosh S., Chakraborty T.J. On Behavior Fault Modelling for Digital Designs. Kluwer Academic Publishers.J. of Electronic testing: Theory and Applications, 2, 1991, pp. 135-151.
12. Giambiasi N. et. al. Test pattern generation for behavioral descriptions in VHDL. *Proc. of the VHDL conference*, Stockholm, 1991, pp. 228-234.
13. Abraham J.A. Fault modeling in VLSI. *VLSI testing. North-Holland* 1986, pp.1-27.
14. Nigh P.and Maly W. Layout - Driven Test Generation. *Proc. ICCAD*, 1989, 154-157.
15. Jacomet M. and Guggenbuhl W. Layout-Dependent Fault Analysis and Test Synthesis for CMOS Circuits. *IEEE Trans. on CAD*, 1993, **12,** 888-899
16. Lee J. and Patel J.H. Architectural level test generation for microprocessors. *IEEE Trans. CAD*, vol.13, no.10, pp.1288-1300, Oct. 1994.
17. Santucci J.F. et al. Speed up of behavioral ATPG, *30th ACM/IEEE DAC*, pp. 92-96, 1993.
18. Ubar R. Test Synthesis with Alternative Graphs. *IEEE Design and Test of Computers.* Spring, 1996, pp.48-59.
19. Ubar R., Moraviec A., Raik J. Cycle-based Simulation with Decision Diagrams. IEEE Proc. of Design Automation and Test in Europe. Munich, March 9-12, 1999, pp.454-458.