Chapter 2
Circuits and Testing

This chapter gives the basic information about circuits and testing of circuits. The role of *Automatic Test Pattern Generation* (ATPG) in the production test is presented in Sect. 2.1. Section 2.2 gives information about the used abstraction level of circuits and shows the modeling as well as the basic notations for the circuit representation used. In Sect. 2.3, the meaning of a fault model is described and relevant fault models are introduced, while classical algorithms for test pattern generation for these fault models are presented in Sect. 2.4. Section 2.5 briefly reviews the industrial test environment.

2.1 Post-Production Test

The manufacturing process of a circuit is very vulnerable to defects of different kinds. As a matter of fact, defects created during the manufacturing process can not be avoided, especially due to shrinking feature sizes. However, the delivery of defective chips to customers has to be prevented. Therefore, each manufactured chip is tested for its functional correctness by a post-production test.[1] The test of a circuit is depicted in Fig. 2.1. Input stimuli are applied consecutively at the *Primary Inputs* (PIs) of the *Circuit Under Test* (CUT) and the responses are monitored at the *Primary Outputs* (POs). If a received response differs from the expected one, the chip is classified as erroneous. Input stimuli applied in order to test the correct function of the CUT are also referred to as tests or test patterns. A test vector is referred to as a single assignment of the PIs. In contrast, a test or test pattern can be a single test vector or multiple test vectors applied consecutively.

However, a complete test of all possible input stimuli is not feasible. The number of possible input stimuli is exponential in the number of inputs. For example, a chip

[1] Note that it is basically assumed for this test that the design itself is free of errors.

S. Eggersglüß and R. Drechsler, *High Quality Test Pattern Generation and Boolean Satisfiability*, DOI 10.1007/978-1-4419-9976-4_2,

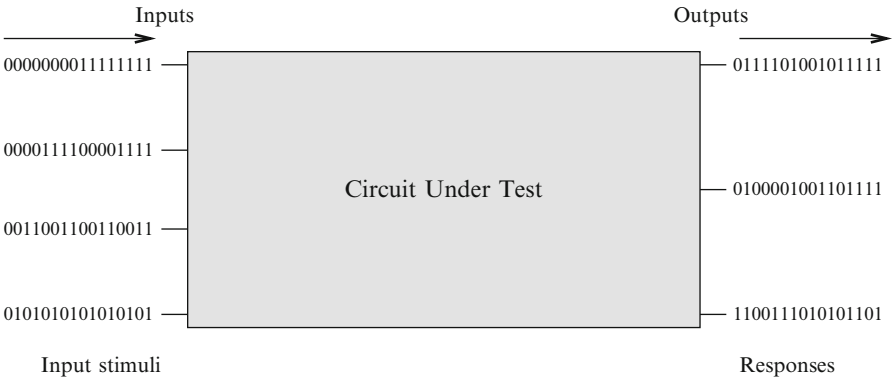

Fig. 2.1 Circuit under test

with 64 inputs has $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$ possible test vectors. This is not acceptable with respect to the test time as well as to the test costs. In order to overcome this problem, a subset of all possible input stimuli has to be chosen, i.e. the test set. This test set should be small at the one hand but on the other hand be able to detect a large number of possible defects.

Crucial for the test set computation is the use of *fault models*. A fault model is a mathematical abstraction of a physical defect and describes the logic behavior of the defect. The use of fault models allows the application of efficient algorithms. Fault models are essential for an efficient test set computation due to the large number of possible physical irregularities. The fault model most widely used in practice is the *Stuck-At Fault Model* (SAFM). Here, a signal line (or connection) in the circuit is permanently "stuck" at a specified value, i.e. 0 or 1. The SAFM is well-understood and known to detect a wide variety of physical defects [JG03]. In an optimal test set, each fault of a particular fault model, e.g. a stuck-at-0 and a stuck-at-1 fault for each connection in the CUT, is detected by at least one test pattern contained in the test set or is known as untestable.

The computation of the test set is generally known as ATPG. The ATPG process involves complex calculations and is typically performed only once for each design. The generated test set is applied to each manufactured chip. Different ATPG techniques can be used for the computation of the test patterns. *Random Test Pattern Generation* (RTPG) randomly generates input stimuli for the CUT, i.e. a test pattern. A fault simulator is used in order to detect which faults are detected by this test pattern. RTPG is fast, but suffers from the circumstance that, in most cases, only a low fault coverage can be reached in reasonable time. Furthermore, untestability cannot be proven. A high fault coverage is mandatory to maintain a certain quality level for chips delivered to customers.

In contrast, *Deterministic Test Pattern Generation* (DTPG) takes a particular fault as input and generates a test pattern which detects this fault explicitly or proves that no test pattern for this fault exists. By this, high fault coverage can be achieved because each single fault is targeted. However, DTPG is more complex

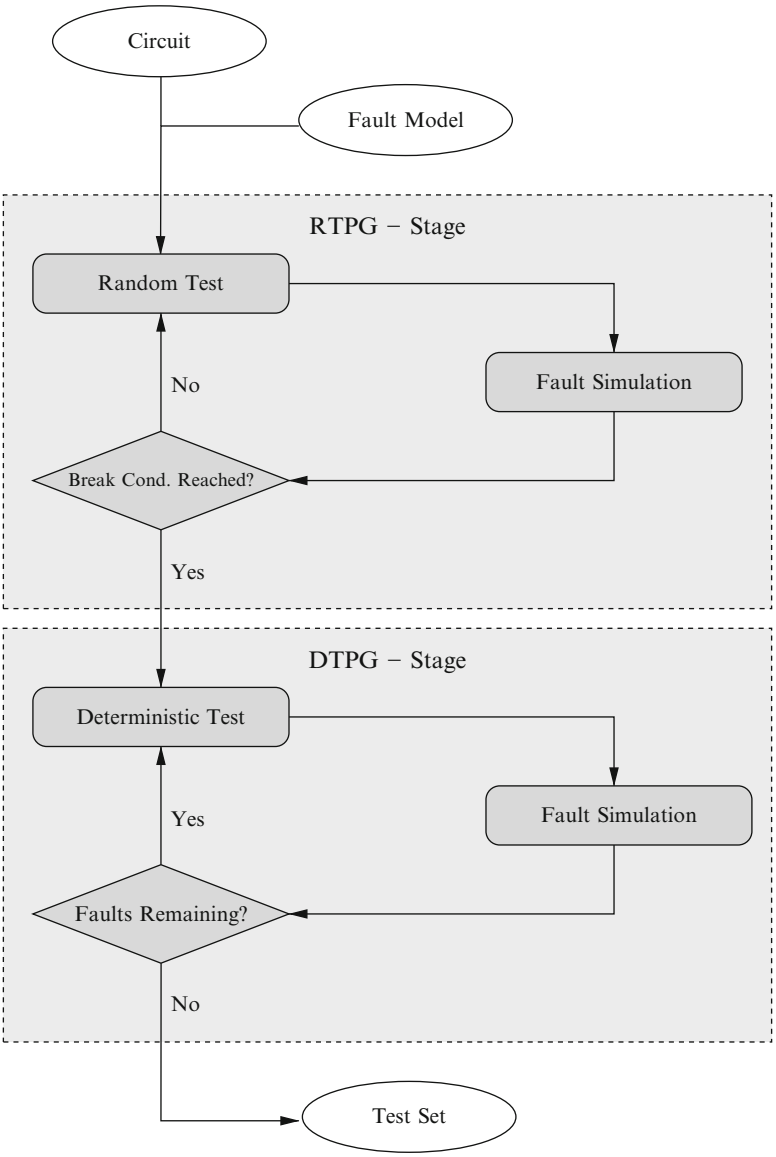

Fig. 2.2 Simple representation of an ATPG system

than RTPG and consequently needs more run time for a single call. In fact, DTPG for a particular fault is proven to be an NP-complete problem for arbitrary digital circuits [FT82]. In modern ATPG systems, a combination of fault simulation, RTPG and DTPG is employed resulting in a good trade-off between quality and run time. Figure 2.2 shows a simplified version of such an ATPG system.

First, RTPG is started as a pre-process to generate test patterns for the easy-to-detect faults. By this, a large number of faults does not have to be targeted by DTPG. For the remaining faults, DTPG is invoked. The fault simulator is used to compute additionally detected faults for each generated test pattern. DTPG has not to be executed anymore for these faults, since there already exists a test pattern that detects them. If all faults are either untestable or detected by a generated test, the ATPG process is finished. The most time consuming task of this procedure is the DTPG-stage. The improvement of DTPG is topic of this book. In the following, DTPG is generally referred to as ATPG.

2.2 Circuits

A circuit $\mathscr{C}$ can be represented in different abstraction levels, for instance in *register-transfer level, gate level, switch level* or *physical level* [WWW06]. Circuits are usually modeled in gate level representation for test generation.

Definition 2.1. A combinational circuit in gate level representation is a directed acyclic graph $\mathscr{C} = (\mathscr{G}, \mathscr{S}, \mathscr{I}, \mathscr{O})$, where

- $\mathscr{G}$ is the set of gates,
- $\mathscr{S}$ is the set of signal lines or connections,
- $\mathscr{I}$ is the set of primary inputs and
- $\mathscr{O}$ is the set of primary outputs.

Gates are usually denoted by lower case latin letters. Here, additional indices can be used for ordering. A gate g of circuit $\mathscr{C}$ is often denoted by $g \in \mathscr{C}$. The underlying graph structure of $\mathscr{C}$ is created by signal lines between gates. A signal line s connects exactly two gates g, h to each other (denoted by $s = g \times h$). Gate g is the *predecessor* of h, while h is referred to as *successor* of g. Signal lines are usually denoted by the notation of the predecessor gate, i.e. g. Additionally, if variables are assigned to a signal line, the same notation is used.

Each gate has a specific type defining the function of the gate. The basic gates or gate types used in this book are depicted in Fig. 2.3. The gates INV (NOT), AND, OR and XOR correspond to the logical operators (see also Sect. 3.1). The gates NAND, NOR and XNOR (EQUIVALENCE) correspond to the inverted versions of these operators. The gate BUF represents the identity function.

More complex primitive gates are the multiplexer (MUX) and the BUS or busdriver (BUSDRV), respectively. Multiplexer and busdriver are non-symmetric gates. Therefore, a unique order of the inputs, i.e. the predecessors, is necessary. Additionally, BUS and BUSDRV are not Boolean gates but tri-state elements with an additional state of high impedance. All gate types presented until now have only one successor. FANOUT gates can have multiple successors and are employed to represent branches in $\mathscr{C}$. In particular, FANOUT gates are important for fault modeling. The outgoing branches of a FANOUT gate are typically not denoted by the predecessor gate, but have own notations.

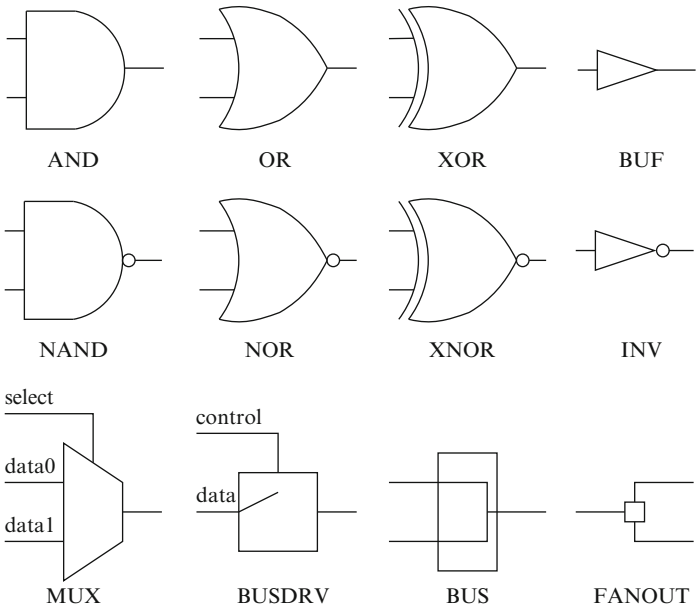

Fig. 2.3 Basic gates

A gate with no incoming connections is a PI, while a gate with no outgoing connection is a PO of $\mathscr{C}$. The PIs of $\mathscr{C}$ are often denoted by $i_1, \ldots, i_n$, while the POs are often denoted by $o_1, \ldots, o_m$. The *transitive fanin* of a gate g is the set of gates on which g is structurally dependant. This is denoted by $\mathscr{F}(g)$.

In contrast to a combinational circuit, a sequential circuit $\mathscr{C}$ contains state elements or flip-flops. A sequential circuit is a circuit whose output value depends not only on the applied input values but also on the current state of the circuit, i.e. values stored in the state elements. The next state of the circuit is likewise computed by the combinational part, the input values and the current state.

Definition 2.2. A sequential circuit is described as $\mathscr{C} = (\mathscr{G}, \mathscr{S}, \mathscr{I}, \mathscr{O}, \mathscr{F})$ where $\mathscr{G}, \mathscr{S}, \mathscr{I}, \mathscr{O}$ denote the elements of a combinational circuit as given in Definition 2.1 and $\mathscr{F}$ is the set of flip-flops contained in $\mathscr{C}$.

Flip-flops store the result of the combinational part of the circuit in time frame t_i and propagate it in the next time frame t_{i+1}. Here, a time frame is specified as the duration of a clock period. Generally, this is referred to as *sequential behavior*. A schematic view of a sequential circuit is shown in Fig. 2.4.

2.2.1 Scan-Based Testing

Scan-based testing – in contrast to functional testing – is used to decrease the complexity of the test of sequential circuits. The use of storage elements like

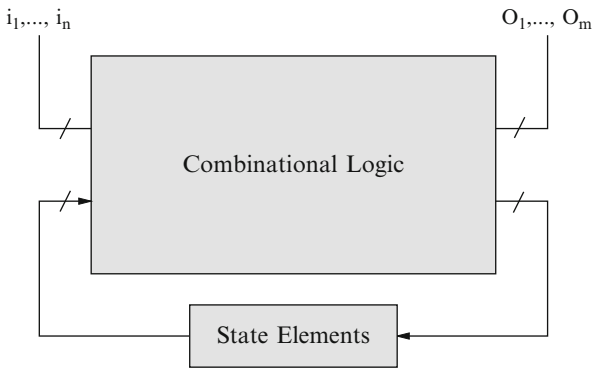

Fig. 2.4 Sequential circuit

flip-flops requires pre-initialization in order to detect certain faults. This means, the storage elements must be loaded with the desired values by the application of a sequence of vectors. This imposes overhead for the test generation as well as to the test application. In contrast, scan techniques are used to reduce this overhead. Using scan-based testing, the flip-flops are connected in several *scan-chains* [WA73, EW77]. Due to the use of scan chains, each flip-flop $f \in \mathscr{F}$ can be pre-initialized with an arbitrary value during a shift-mode before the application of the test. After the test has been applied, the response value of the combinational logic $\mathscr{F}(f)$ can be shifted out, i.e. an observation is possible. As a consequence, each flip-flop $f \in \mathscr{F}$ is splitted into a *Pseudo Primary Input* (PPI) – output of f – and a *Pseudo Primary Output* (PPO) – input of f. That means, for a one-vector-test, each PPI (PPO) can be treated as a PI (PO).

For two-vector-tests which are usually applied in delay testing, different scan modes were introduced. The most prevalent scan modes are *enhanced-scan* [DS91], *launch-on-shift (skewed-load)* [SP93] and *launch-on-capture (broad-side)* [SP94].

In the enhanced-scan mode, no relation between both patterns is required. An arbitrary vector-pair can be applied which makes the test generation easier. However, this requires a special scan-design (hold-scan) which causes overhead. In a skewed-load scan mode, the second test vector is a shifted version of the first vector. During broad-side testing, the second vector is the functional response of the first vector. Throughout this book, the broad-side testing technique is assumed. Tests for this scheme are more complex to compute since the logical behavior of two time frames has to be considered.

2.3 Fault Models

During the manufacturing process, a large range of physical defects may enter the design causing malfunctions. Chips with malfunctions have to be filtered out by the post-production test before being delivered to customers. However, testing for each

single known defect is impractical because of the large number of potential defects and the associated excessive computational effort. Therefore, "logical" fault models are introduced. A fault model is a mathematical abstraction and models the logical behavior of physical defects.

The use of a fault model has several advantages [ABF90]. Different physical defects can be modeled by a single fault model. Even physical defects which are not fully understood can be covered. Additionally, fault models are technology-independent. The developed test generation methods do not have to be modified when the underlying technology changes. Furthermore, the complexity of test generation is significantly reduced due to the logic modeling. Therefore, fault models are essential for an efficient test set computation. Relevant fault models are introduced in this section.

First, the basic *Stuck-at Fault Model* (SAFM) is introduced in detail in Sect. 2.3.1. Then, the most common delay fault models are described in Sect. 2.3.2. For a detailed overview on a large range of existing fault models, it is referred to [BA00].

2.3.1 Stuck-at

The SAFM [Eld59, GNR61] is well-understood and the fault model most widely used in practice. Although being very simple, the SAFM is known to detect a large number of potential defects. The SAFM belongs to the group of static fault models. Static fault models affect the logic function of the circuit.[2] A signal line f (short: line) is assumed to be "stuck" at a fixed value and does not depend on the input values or on $\mathscr{F}(f)$, respectively, anymore. There are two different faults associated to each line. When the line is stuck at the value 0, it is called *stuck-at-0* (s-a-0) fault. Otherwise, when the line is stuck at the value 1, the fault is called *stuck-at-1* (s-a-1) fault. The number of stuck-at faults in a circuit $\mathscr{C}$ is linear in the number of signal lines, i.e. for n signal lines in $\mathscr{C}$, there are $2n$ possible stuck-at faults. Note that for FANOUT gates, each outgoing line counts as possible fault location for which a test has to be generated. Formally, a stuck-at fault F is denoted by a tuple (f, v) where f is the faulty signal line and v is the fault value, i.e. 0 for a s-a-0 fault and 1 for a s-a-1 fault. This is shown in the following example.

Example 2.1. Consider Fig. 2.5 where a simple example circuit is presented. Figure 2.5a shows the correct circuit, while in Fig. 2.5b, c, s-a-0 faults are injected on each outgoing "branch" of the FANOUT gate, respectively. By injecting the fault, i.e. a constant value, the branch is disconnected from the AND gate a.

The *Multiple SAFM* assumes the presence of multiple stuck-at faults. However, this fault model is rarely used in practice due to the huge number of faults.

[2] Other static fault models are e.g. the bridging fault model or the cellular fault model. In this book, we consider only the SAFM. The results can be transferred.

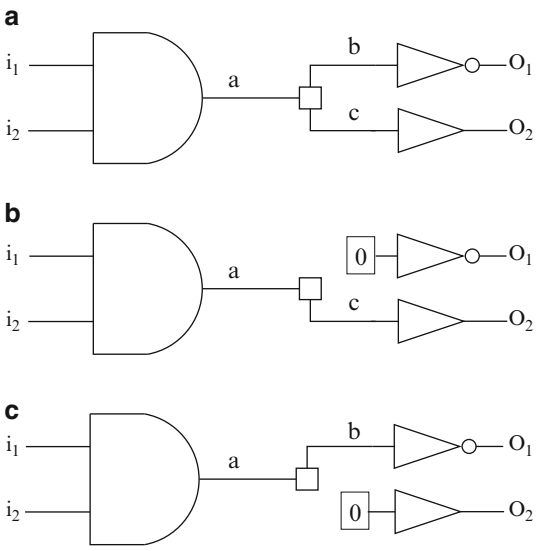

Fig. 2.5 SAFM example. (**a**) Correct circuit, (**b**) stuck-at-0 fault on branch *b* and (**c**) stuck-at-0 fault on branch *c*

2.3.1.1 Test Generation

A test pattern or simply "test" for a stuck-at fault consists of one input vector V which activates the fault at the fault site and propagates the fault effect to an observation point, i.e. a PO or PPO. The chosen propagation path has to be logically sensitized for the propagation of the fault effect. A method for creating test patterns is the computation of the *Boolean Difference* [SHB68]. Here, the correct and the faulty circuit form a circuit similar to a *miter* circuit [Bra93], as it can be used for combinational equivalence checking. The inputs of both the correct and faulty circuit are connected in a miter to ensure that both circuits assume the same input values. The outputs of both circuits are compared using XOR gates. The results of the comparisons serve as inputs of an OR gate. Finally, the output of the OR gate is constrained to the value 1. By this, it is ensured that at least one original output produces a different result in the correct and faulty circuit.[3] The following example demonstrates the modeling.

Example 2.2. Consider the correct circuit given in Fig. 2.5a and the circuit with an injected stuck-at fault $(b,0)$ presented in Fig. 2.5b. The resulting miter circuit is shown in Fig. 2.6. In this simple example, four different input vectors exist.

[3]For reason of simplicity, the method is described by using gates. Boolean expressions are used instead of gates in the original work.

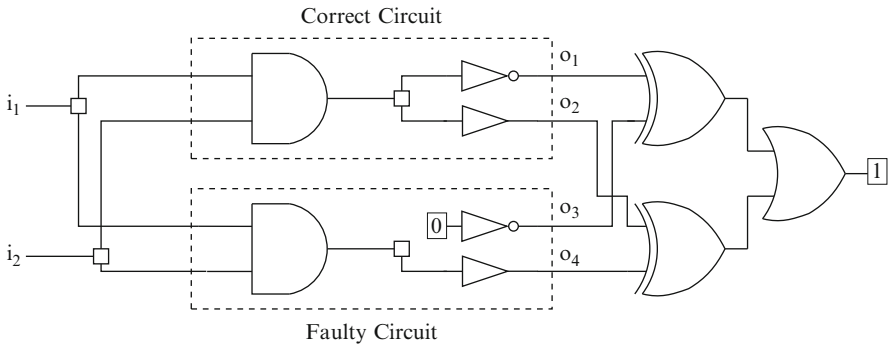

Fig. 2.6 Boolean difference

Table 2.1 Input vectors

	Inputs		Outputs				
Vector	i_1	i_2	o_1	o_2	o_3	o_4	OR
V_1	0	0	1	0	1	0	0
V_2	0	1	1	0	1	0	0
V_3	1	0	1	0	1	0	0
V_4	1	1	0	1	**1**	1	**1**

Table 2.1 shows the input vectors and the resulting output values. Only input vector V_4 produces a difference on the output. Therefore, the input vector V_4 is a test pattern for the stuck-at fault $(b, 0)$.

Faults for which at least one test pattern exists are called *testable*. If no test pattern exists, the fault is called *untestable* or *redundant*. The question whether a fault is testable or untestable is an NP-complete problem [FT82]. More elaborated and more efficient ATPG algorithms are described in Sect. 2.4.

2.3.1.2 Fault Collapsing

The number of faults which have to be considered by the ATPG process is typically very huge. Therefore, *fault dominance* and *fault equivalence* relationships are exploited in order to reduce the number of faults, which have to be processed by ATPG.

Definition 2.3. A stuck-at fault F_1 *dominates* a fault F_2 if every test pattern that detects F_2 also detects F_1. When F_1 dominates F_2 and F_2 dominates F_1 as well, F_1 and F_2 are said to be *equivalent*.

If F_1 dominates F_2, a test has to be generated only for F_2. If F_1 and F_2 are equivalent, a test can be generated either for F_1 or F_2 to detect both faults. The procedure to reduce the target fault set by exploiting fault dominance relationships is called *fault collapsing* [ABF90].

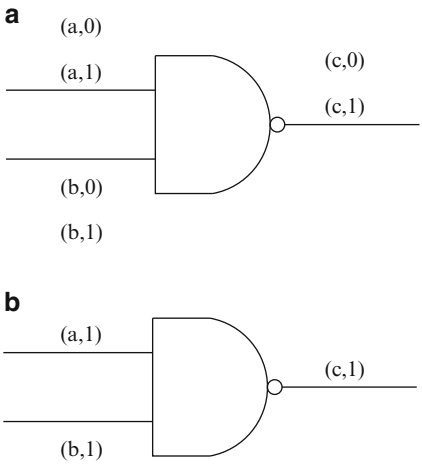

Fig. 2.7 Fault collapsing. (**a**) Uncollapsed fault set and (**b**) collapsed fault set

Usually, fault equivalence and fault dominance relationships are exploited only locally since the overhead for identifying all relationships is too high [Lio92]. The following example shows the local relationships of stuck-at faults at a NAND gate.

Example 2.3. Consider the 2-input NAND gate in Fig. 2.7a. At the output c and at each input a, b of the NAND gate, two stuck-at faults (s-a-0, s-a-1) are possible. Fault $(c, 1)$ can only be detected when both inputs assume the non-controlling value 1. This is also necessary for detecting $(a, 0)$ and $(b, 0)$. Because this test pattern is the only existing test pattern for these faults, $(c, 1)$, $(a, 0)$ and $(b, 0)$ are equivalent. Therefore, only one of them has to be targeted, e.g. $(c, 1)$.

The fault set can be further reduced by using fault dominance relationships. A test for $(a, 1)$ and $(b, 1)$ necessitates the controlling value 0 at one input and therefore always detects $(c, 0)$, but not vice versa. $(a, 1)$ and $(b, 1)$ dominate $(c, 0)$. Consequently, $(c, 0)$ needs not to be considered in the fault set. The collapsed fault set of the NAND gate is shown in Fig. 2.7b.

2.3.2 Delay

Due to the high operating speed of modern designs, delay testing is performed to detect timing defects as well as to guarantee that the design meets the performance specification. Clock signals are used to synchronize the inputs and all outputs are expected to assume their final value within a defined clock period. For this purpose, several delay fault models were developed to detect inconsistencies in the temporal behavior. Delay faults must be considered under different aspects than stuck-at faults, since the timing of the circuit is ignored during stuck-at testing. The most common delay fault models, i.e. the *Path Delay Fault Model* (PDFM) and the *Transition Fault Model* (TFM), are introduced in this section.

2.3.2.1 Path Delay

The PDFM [Smi85, LR87] is the most accurate delay fault model. A manufactured circuit is said to be free of timing defects if every path from a PI to a PO propagates its transitions in less time than the specified clock cycle. A *Path Delay Fault* (PDF) models a distributed delay on a structural path $\mathscr{P}$ from a PI to a PO of $\mathscr{C}$. A PDF occurs if the size of the delay defect on $\mathscr{P}$ exceeds the slack. The slack of a path is the difference between the clock cycle and the specified path delay. This fault model is suitable for detecting small as well as large delay defects.

Definition 2.4. A structural path $\mathscr{P}$ is defined as the sequence of gates $g_1, \ldots, g_k$, where g_1 is a PI or PPI and g_k is a PO or PPO. The path $\mathscr{P}$ must be complete. That means, each gate g_i on $\mathscr{P}$ with $0 < i < k$ must be an input of g_{i+1}. If a gate g_i is located on path $\mathscr{P}$, it is denoted by $g_i \in \mathscr{P}$.

A PDF occurs when the cumulative delay of all gates and signal lines along $\mathscr{P}$ exceeds the time for a specified clock cycle. Because the effects of a physical fault on the delay may be different for both types of transitions, two different fault types are modeled by the PDFM. According to the direction of the transition at the beginning of path $\mathscr{P}$, there is a rising PDF as well as a falling PDF for each structural path. A rising transition goes from logic 0 in the *initial* time frame to logic 1 in the *final* time frame. Analogously, a falling transition goes from logic 1 to logic 0. Formally, a PDF F is a tuple $(\mathscr{P}, t)$ where $\mathscr{P}$ is a structural path and $t \in \{\uparrow, \downarrow\}$ denotes the direction of the transition at g_1, i.e. $\uparrow$ is used for a rising transition and $\downarrow$ for a falling transition.

A test pattern for a PDF consists of two input vectors V_1, V_2 applied in two consecutive time frames t_1, t_2. The vector V_1 is applied in the initial time frame t_1 and places the initial transition value at the fault site. The vector V_2 is then applied at operating speed in the final time frame, launches the transition at g_1 and propagates it to g_k by sensitizing a path from g_1 to g_k. The transition has to arrive at g_k in the specified clock cycle before the sampling time. Otherwise, a delay fault is detected at $\mathscr{P}$.

However, if multiple delay faults are present in $\mathscr{C}$, a test pattern might not detect the fault because other delay faults may mask the targeted PDF. Therefore, sensitization criteria were developed to classify tests according to their fault detection abilities, i.e. the quality. A test is called *robust* if and only if it detects the fault independently of other delay faults in the circuit [Smi85, LR87]. *Non-robust* tests guarantee the detection of a fault if there are no other delay faults in the circuit [LR87]. Robust tests are more desirable to obtain since they provide a higher quality. Detailed discussions concerning the classification of PDF tests and further sensitization criteria for a more detailed differentiation, e.g. strong robust or functional sensitizable, can be found in [KC98].

Robust and non-robust tests differ in their constraints on the side inputs of $\mathscr{P}$. A side input s of $\mathscr{P}$ is an input of gate $g_i \in \mathscr{P}$ with $1 < i \leq k$ which is not on $\mathscr{P}$ ($s \notin \mathscr{P}$). The sensitization criteria for robust and non-robust test patterns are shown in Table 2.2. The values shown in this table correspond to the five-valued

Table 2.2 Sensitization criteria for robust and non-robust test patterns

	Robust		
Gate type	Rising	Falling	Non-robust
AND/NAND	$X1$	$S1$	$X1$
OR/NOR	$S0$	$X0$	$X0$

logic $\mathscr{L}_5 = \{S0, S1, X0, X1, XX\}$ originally proposed in [LR87]. The values $S0$ and $S1$ denote static values. The letter 'X' in a value's name denotes a don't care. This means that the initial value is don't care at $X0/X1$, while both, the initial and final value are don't care at XX.

All side inputs of $\mathscr{P}$ have to assume a non-controlling value in the final time frame for a non-robust test. The non-controlling value of a gate is the opposite value of the controlling value. The controlling value of a gate g is the logic value which, when assumed at any input of g, determines the output value of g regardless of the values on other signals, i.e. 0 for AND/NAND and 1 for OR/NOR.

The constraints on the side inputs for a robust test depend on the transition of the on-path input g_i. If the transition on g_i goes from the non-controlling value to the controlling value of gate g_{i+1}, the side inputs of g_{i+1} have to assume a static non-controlling value (denoted by $S0/S1$). A static non-controlling value at the side inputs guarantees that a delayed on-path transition of g_i cannot be masked by a transition or a glitch on the side inputs. If the on-path transition goes from the controlling value to the non-controlling value, the side inputs have to assume a non-controlling value only in the final time frame – as specified for the non-robust sensitization model. In this scenario, a delayed transition cannot be masked by the side inputs, because line g_{i+1} switches to the non-controlling value not until g_i and all side inputs switch to the non-controlling value. The following example demonstrates the different sensitization criteria.

Example 2.4. An example circuit is presented in Fig. 2.8. Let $(\mathscr{P}, \uparrow)$ with $\mathscr{P} = (i_2, a, b, d, o)$ be the PDF under test. The following constraints are needed for non-robust sensitization (see Fig. 2.8a). The side inputs of gate a and gate o are constrained to $X1$, while the side input of gate d is set to $X0$. This ensures that the non-controlling value is assumed in the final time frame. Additionally, a rising transition is assumed at i_2. This results in the following test pattern:

$$V_1 = \{i_1 = X, i_2 = 0, i_3 = X, i_4 = X\}$$
$$V_2 = \{i_1 = 0, i_2 = 1, i_3 = 1, i_4 = X\}$$

In contrast, the side inputs of gate d and gate o have to be sensitized by a static non-controlling value in the robust sensitization model (shown in Fig. 2.8b). This is due to the transition from the non-controlling value of the gate to the controlling value. Note that the transition is inverted after an inverting gate, e.g. gate d in this example. A robust test satisfying these constraints is for example:

$$V_1 = \{i_1 = 0, i_2 = 0, i_3 = X, i_4 = 1\}$$
$$V_2 = \{i_1 = 0, i_2 = 1, i_3 = 1, i_4 = 1\}$$

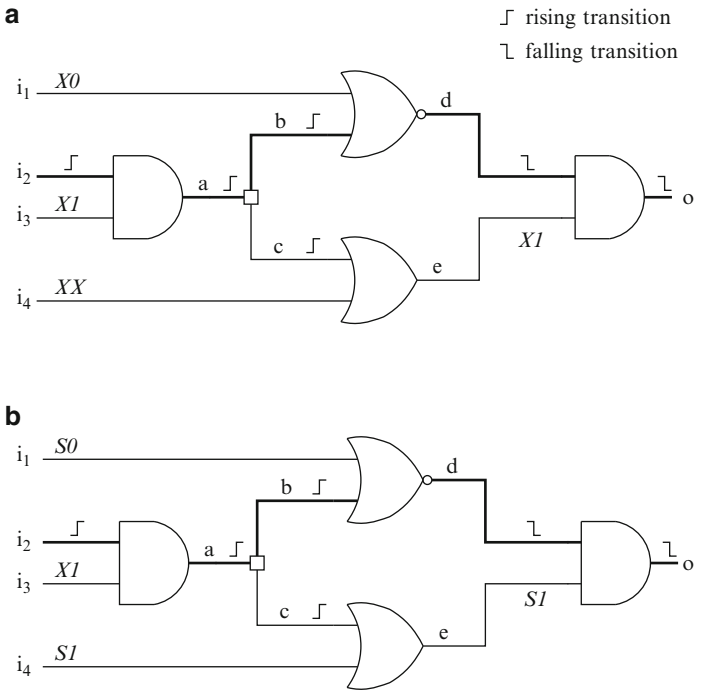

Fig. 2.8 Non-robust and robust sensitization. (**a**) Non-robust and (**b**) robust

The major limitation of the PDFM – which is the most accurate delay fault model – is the large number of paths in modern designs. The number of paths may be exponential in the number of gates. Performing ATPG for each single PDF is not practical due to the excessive computational effort and the resulting large size of the test set. Therefore, only a subset of all PDFs – faults on so-called critical paths – are considered for test generation in practice [LRS89, SP02].

2.3.2.2 Transition

In contrast to the PDFM, the TFM [BR83, LM86, SB87, WLRI87, Che93] is a local or "lumped" delay fault model. This fault model assumes a local delay defect, which affects only one single gate in the circuit. Similar to the PDFM, two different faults are associated with one fault site. The *slow-to-rise* fault models a delayed rising transition at the fault site, whereas the *slow-to-fall* fault models a delayed falling transition. The TFM is a specialization of the *Gate Delay Fault Model* (GDFM) [HRVD77, SB77, PR88] and has replaced the GDFM in industrial practice. The main difference between both fault models is that the GDFM includes the size of the delay defect while the TFM assumes that the delay defect is sufficiently large for detection.

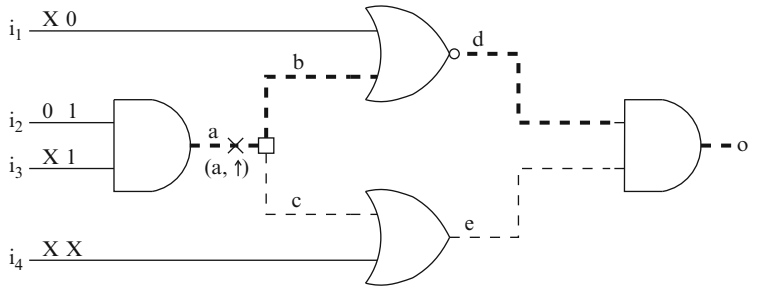

Fig. 2.9 Propagation paths – transition fault

Formally, a *Transition Fault* (TF) F is a tuple (f,t) with f as the affected gate or its outgoing connection, respectively, and $t \in \{\uparrow,\downarrow\}$. Since the size of the delay defect is not considered during test generation, the TFM is regarded as an idealized delay fault model. In order to test a TF at gate g, the fault must be activated at the affected gate and propagated to a PO. The activation path as well as the propagation path is chosen by the test generation algorithm. An activation path of a TF (f,t) is defined as a path from a PI to f which propagates the transition to the affected gate. A propagation path is the path which propagates the transition from the fault site f to a PO.

An advantage of the TFM is the similarity to the SAFM. Two consecutive time frames t_1,t_2 have to be considered for the TFM. The initial transition value is placed at the fault site in the initial time frame t_1. The transition is launched at operating speed and propagated to a PO in the final time frame t_2. This is done by logically sensitizing a path from the affected gate to a PO. In t_2, the TF behaves like the corresponding stuck-at fault, i.e. the slow-to-rise fault $(f,\uparrow)$ behaves like the s-a-0 fault $(f,0)$ and the slow-to-fall fault $(f,\downarrow)$ behaves like the s-a-1 fault $(f,1)$. The test vector V_2 is therefore identical to the pattern detecting the corresponding stuck-at fault. Figure 2.9 shows an example for the propagation of a TF.

Example 2.5. Consider the example circuit shown in Fig. 2.9, which has been already used to exemplarily present test generation for the PDFM. Assume that there is a TF $(a,\uparrow)$ in the circuit. The test generation algorithm can choose between two possible propagation paths:

$$\mathscr{P}_1 = (a,b,d,o)$$
$$\mathscr{P}_2 = (a,c,e,o)$$

A test for $(a,\uparrow)$ has to activate the transition at a and logically sensitize at least one of the shown propagation paths. The propagation along path $\mathscr{P}_2$ is not possible, because the final value on line d (caused by the transition on line a) would block path $\mathscr{P}_2$. A test for the TF $(a,\uparrow)$ using $\mathscr{P}_1$ as propagation path is for example:

$$V_1 = \{i_1 = X, i_2 = 0, i_3 = X, i_4 = X\}$$
$$V_2 = \{i_1 = 0, i_2 = 1, i_3 = 1, i_4 = X\}$$

Note that the test pattern is identical to the non-robust test pattern presented in Example 2.4. That is because the same activation path as well as the same propagation path is used.

Since the TFM is very similar to the SAFM, existing efficient solutions for the SAFM can be reused with slight modifications. However, the fault collapsing rules are more restrictive due to the consideration of two time frames [WLRI87]. As a result, the number of faults in the collapsed fault set is usually much higher for the TFM than for the SAFM.

The advantage of the TFM in contrast to other delay fault models is that the number of faults is linear in the number of gates and that, due to the similarity to the SAFM, existing ATPG solutions targeting stuck-at faults can be reused. The TFM is widely used in industrial practice providing a good fault coverage. As disadvantage, the TFM is not very accurate. This is because of the assumption that the increased delay is large enough to be detected by any propagation path which is not realistic. Furthermore, the general detectability of small delay defects, which becomes more and more important with the advancing technology, is low.

Several specializations of the TF were proposed. For example, the fault model ALAPTF (As Late As Possible Transition Fault) [GH04] was proposed to address the fault activation condition. It requires that the TF is launched via the longest robust segment ending at the fault site to accumulate small delay defects.

2.4 Classical ATPG Algorithms

Sophisticated ATPG algorithms were developed for the efficient generation of test patterns. An overview on the existing ATPG algorithms for each fault model described in the previous section is given below. Special attention is paid to the algorithms for stuck-at faults because the techniques introduced for this fault model are reused for other fault models as well.

2.4.1 ATPG for Stuck-at Faults

A symbolic test generation method for stuck-at faults based on the Boolean difference [SHB68] was presented in Sect. 2.3.1. However, the symbolic computation of test patterns turned out to be impractical for larger circuits. Subsequent ATPG approaches, see for example [SB91, SSAM93, Bec98], based on *decision diagrams* or *Binary Decision Diagrams* (BDDs) [Bry86], respectively, improve the run time behavior significantly but suffer from their excessive memory consumption when applied to modern circuits. Due to these shortcomings, these approaches did not achieve acceptance in industry. Therefore, this section concentrates on *path-oriented* ATPG algorithms which have been applied in industry for decades.

Table 2.3 Meaning of the values of $\mathscr{L}_5$

Value	Good circuit	Faulty circuit
0	0	0
1	1	1
D	1	0
$\overline{D}$	0	1
X	X	X

Algorithm 1 Outline of the D-algorithm

```
 1: Select_fault();
 2: while Untried_D-chain_exists() do
 3:     D-drive();
 4:     Consistency(); /* Involves decision making and backtracking if necessary */
 5:     if D-chain_is_justified() then
 6:         return TESTABLE;
 7:     else
 8:         Backtracking();
 9:     end if
10: end while
11: return UNTESTABLE;
```

2.4.1.1 D-Algorithm

The first seminal ATPG algorithm was the D-Algorithm [Rot66, RBS67]. The D-algorithm introduces an ATPG method based on the computation and propagation of D-values. The algorithm operates on a five-valued logic $\mathscr{L}_5 = \{0,1,D,\overline{D},X\}$ which is intended to represent the value of a signal line in both the correct or "good" circuit and the faulty circuit simultaneously. The concrete meaning of each value of $\mathscr{L}_5$ is presented in Table 2.3. The values D and $\overline{D}$ are used to represent signals which behave differently in the good and in the faulty circuit. The aim of the D-algorithm is to form a so-called *connected D-chain* (or short: *D-chain*) from the fault site to a PO. A D-chain is a complete path from the fault site to a PO, where all gates on this path assume either D or $\overline{D}$. By this, the propagation of the fault effect to an output is guaranteed. The outline of the D-algorithm is shown in Algorithm 1.

First, a target fault is selected and the initial D-value is injected at the fault site (line 1). That means D for a s-a-0 fault and $\overline{D}$ for a s-a-1 fault. Then, a D-chain is formed by selecting a propagation path from the fault site to a PO (line 3). This is done by driving a *D-frontier* towards the outputs. The D-frontier contains all gates which output values are X but one or more predecessors have a D-value, i.e. either D or $\overline{D}$. A gate g_d from the D-frontier is chosen to propagate the fault. The output of g_d is set to the corresponding D-value and the D-frontier is updated. This procedure is called *D-drive*.

After a D-chain was selected, the D-values on the D-chain have to be justified, i.e. a consistent input assignment has to be searched. This is done by the *Consistency* procedure in line 4. If such an input assignment is found, the algorithm returns with

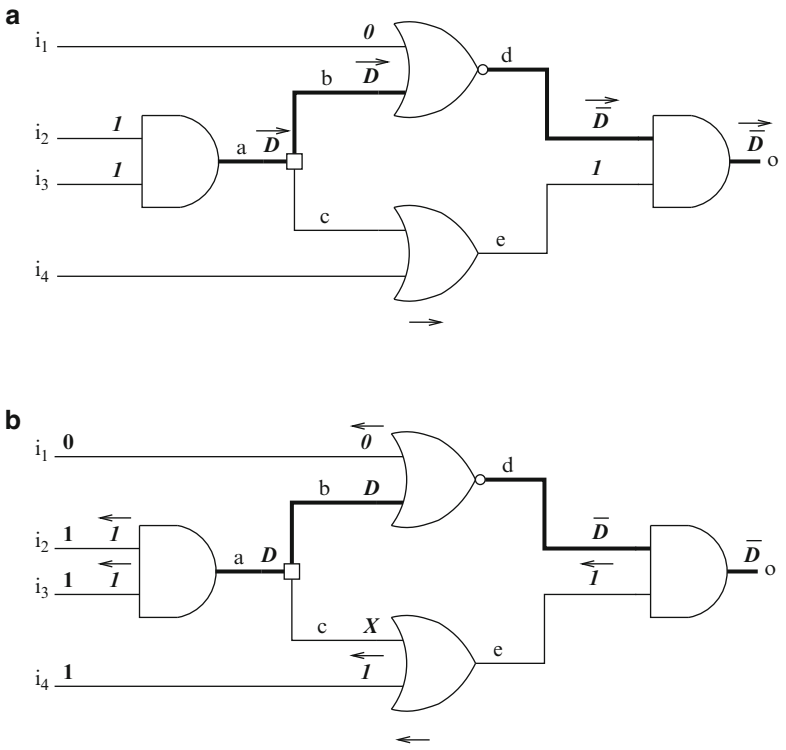

Fig. 2.10 D-algorithm steps. (**a**) D-drive and (**b**) consistency

the resulting test vector (line 6). Analogously to the D-frontier, the term *J-frontier* is introduced to keep track of the unjustified lines in the circuit [ABF90]. In contrast to the D-frontier, the J-frontier is driven towards the inputs. If the D-chain cannot be justified, *Backtracking* is performed and a new D-chain is computed (D-drive). If all potential D-chains were tried and no test pattern could be found, the fault is untestable.

Note that the Consistency procedure and the D-drive procedure involve decision making. At some point, implications are not possible and a decision has to be made. For example, to produce the value 1 at the output of an 2-input OR gate, either one input or both can assume the value 1. A decision tree is used to branch and bound through the search space. If a conflict situation happens, e.g. due to reconvergent paths, backtracking is performed to return to a previous decision point. The branch-and-bound procedure continues until either a test vector is found or the fault is proven to be redundant (untestable), i.e. the complete search space was traversed without finding a test. The following example demonstrates the procedure:

Example 2.6. Consider the circuit shown in Fig. 2.10. A s-a-0 fault on line *a* is targeted for test generation using the D-algorithm. Figure 2.10a shows the D-drive procedure. The D-frontier is driven from the fault site *a* to the output *o*. At the fanout

gate, line b is selected. The resulting D-chain is (a,b,d,o). During the D-drive, value assignments necessary for the propagation of the D-values are gathered. The assignment $i_1 = 0$ is necessary for the propagation of the D-value from line b to line d, whereas the assignment $e = 1$ is necessary for the propagation from line d to the output o. The inputs i_2 and i_3 have to be assigned the value 1 to activate the fault.

When a D-chain is found, the Consistency procedure is invoked to justify the necessary assignments. This is shown in Fig. 2.10b. Here, an input assignment is searched which is consistent with the selected D-chain. In this example, only line e is not an input. Because line e is the output of an OR gate and the controlling value is assumed, it is sufficient that only one predecessor assumes the value 1. In this example, i_4 is chosen, whereas the line c is assigned to X. The generated test pattern is:

$$V_1 = \{i_1 = 0, i_2 = 1, i_3 = 1, i_4 = 1\}$$

The five-valued logic $\mathscr{L}_5$ introduced in [Rot66] was extended in [Mut76]. Here, four additional values were introduced (resulting in a nine-valued logic), which cover the cases where one (either the good or the faulty) value is known, but the other is unknown. By this, the nine-valued logic is more precise.

2.4.1.2 Improved Path-Oriented Algorithms

With the growing complexity of the circuits and the increasing number of reconvergences, the D-algorithm in its original form became very inefficient in the late 1970s. PODEM [Goe81] (*Path-Oriented DEcision Making*) was introduced to improve the performance of ATPG. The main reason of the inefficiency of the D-algorithm was the excessive number of backtracks which have to be performed. In PODEM, the test generation problem is formulated as a search of the n-dimensional 0–1 state space of primary input patterns of an n-input combinational circuit. Therefore, contrary to the D-algorithm, PODEM restricts the decision making to the PIs of the circuit. Thus, the complexity of test generation is reduced from $O(2^s)$ (D-algorithm) to $O(2^n)$ (PODEM), since decisions are carried out on a subset of signals only. This is possible, because all lines can be assigned by propagating the input assignment through the circuit. Here, s denotes the number of signals (including PIs, POs, and internal signals) in the circuit and n denotes the number of PIs – typically $n \ll s$.

Because decisions are restricted to the PIs, the *Consistency* procedure is replaced by a *Backtracing* procedure. Instead of directly assigning a value to a signal line (*objective*) and searching a consistent input assignment as done in the D-algorithm, the backtracing procedure traces a path from the objective signal line backwards to a PI. Backtracing is guided by heuristics and observability/controllability measurements (see e.g [Sne77, GT80]) and the selected PI is likely to help satisfying the objective. Then, forward implication is performed only. A further improvement of PODEM is the *X-path-check*. This is an early test whether the D-frontier still exists, i.e. there is at least one path of X's from the D-frontier to a PO. By this, conflicts can be detected earlier and the overall procedure is improved.

The algorithm FAN (FAN-out oriented test generation algorithm) [FS83] improves the efficiency of test generation by further reducing the number of backtracks and shortening the process time between single backtracks. Special attention is paid to fanout points respectively headlines. A headline is the output of a fanout-free sub-circuit. The following improvements are introduced in FAN.

- *Backward Implications* – In order to detect inconsistencies early, forward and backward implication is performed to assign as many values as possible.[4]
- *Unique Sensitization* – When the D-frontier consists only of a single gate, it often happens that all paths beginning at this gate go through one site or one specific path. If such a path is found, this path is partially sensitized in advance.
- *Multiple Backtracing* – Instead of backtracing a single path as suggested in PODEM, multiple paths are concurrently traced in a breadth-first manner. Decisions are carried out on headlines or fanout points of the circuit.

The FAN algorithm still forms the basis of many ATPG algorithms applied today in industry. However, several improvements were proposed to increase the efficiency of the test generation algorithm, e.g. [KM87, STS88, SA89, RC90, GB90, MGÖD90, WSGM90, KP93, MS94, HP99]. FAN is able to generate tests for a large number of "easy" faults very fast. Therefore, the majority of the proposed improvements aim to reduce the computational effort for "difficult" faults. In particular, significant improvements were achieved by the efficient determination of necessary assignments and the development of powerful learning concepts. The notion of *Learning* is used to describe the derivation of additional implications which cannot be obtained using the common forward and backward implication techniques alone. The basic learning concepts are outlined in the following.

2.4.1.3 Learning Concepts

The ATPG system SOCRATES (Structure-Oriented Cost-Reducing Automatic TESt pattern generation system) [STS88] improves the techniques introduced in FAN which are enumerated above. In particular, a global learning scheme was proposed. Indirect implications were identified in a pre-process carried out on the circuit structure using pre-defined rules. The additional "learned" implications denote relationships between signals which are not obvious. The application of these implications helps the algorithm to prevent "bad" decisions leading into a non-solution sub-space. As a result, the number of costly backtracks can be reduced and test generation is accelerated.

Example 2.7. An example (taken from [STS88]) is presented in Fig. 2.11. The figure shows a small circuit part. Simple forward implications are depicted in

[4] Backward implications were already an inherent feature of the D-algorithm, but discarded in PODEM.

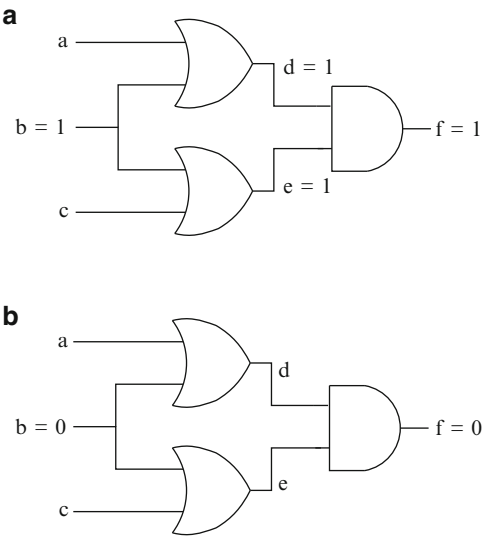

Fig. 2.11 Learning of indirect implications. (**a**) Direct implication of $b = 1$ and (**b**) indirect implication of $f = 0$

Fig. 2.11a. Assigning the value 1 to line b leads to an assignment $d = 1$ and $e = 1$, which in turn imply $f = 1$. Therefore, the direct implication $(b = 1) \rightarrow (f = 1)$ holds. The application of the law of contraposition yields the indirect implication $(f = 0) \rightarrow (b = 0)$, which is shown in Fig. 2.11b. This implication is indirect, because it cannot be obtained directly using backward implication.

An improved dynamic learning technique is presented in [SA89]. Here, the learning process is not performed once as a pre-process but dynamically during the search in each branching step, i.e. after each decision. As a result, the number of identified implications is much higher and test generation can be performed with fewer backtracks. As disadvantage, huge computational effort is needed for large circuits to perform dynamic learning in each branching step. The approach presented in [KP93] improves the dynamic learning technique by restricting the dynamic learning process to the "active area". This so-called *oriented dynamic learning* reduces the computational effort without sacrificing much important learned information.

An elaborated learning approach is shown in [KP94]. Here, the concept of *Recursive Learning* is presented. Instead of using a decision tree to keep track of the tried combinations of signal values, a recursive learning technique is used, i.e. recursively calling certain learning functions. The recursive learning technique is complete by itself. That means, all necessary assignments and logic relations between signals can be identified given enough recursions. As a result, a test can be found without any backtracks. However, recursive learning is very time-consuming when applied as stand-alone engine. In practice, this type of learning is combined with the FAN algorithm and only applied to leave a non-solution space as fast as possible.

The algorithm LEAP (LEvel-dependant Analysis in Path sensitization) presented in [MS94] introduces the techniques *failure-driven assertions* and *dependency-*

directed backtracking to prune search space in path sensitization problems such as ATPG. These techniques learn dynamically from conflicting signal assignments by analyzing the conflict and improving the backtracking procedure by leaping multiple decision points.

2.4.1.4 Boolean Satisfiability

In contrast to the ATPG approaches working on a structural circuit representation at gate level, methods based on *Boolean Satisfiability* (SAT) work on a Boolean formula. Different to algebraic methods (such as the Boolean difference method) which also work on a problem instance represented as a formula, SAT methods do not perform costly symbolic manipulations. SAT methods used for test generation can be grouped into two different categories which can be distinguished by the underlying data representation.

- *Conjunctive Normal Form* (CNF) – The CNF model represents the logic functionality of the circuit as a Boolean formula. Highly efficient SAT solvers were developed to evaluate such problems. A major reason for the robustness of modern CNF-based SAT solvers is the powerful conflict analysis (see Chap. 3 for more information). As disadvantage, structural information is lost, which is used by path-oriented ATPG algorithms to speed up the search.
- *Implication Graph* (IG) – An IG is a directed acyclic graph which nodes are the true or complemented signals or signal states, respectively. Implications are represented by edges between nodes. The advantage of the IG model over a topological model is that the IG abstracts from the pure structural description and combines structural and functional information in one model. However, IG-based methods do not incorporate conflict analysis techniques and do not benefit from the latest advances in SAT solving.

CNF-based SAT algorithms are e.g. TEGUS [SBS96], CGRASP/TG-GRASP [GSM99, MS97], PASSAT [DEF+08] and TIGUAN [CPL+10]. Because CNF-based algorithms are the main focus of this book, these approaches will be introduced more detailed in Chap. 4. A recent SAT-based ATPG approach is TIGUAN (Thread-parallel Integrated test pattern Generator Utilizing satisfiability ANalysis) [CPL+10]. State-of-the-art ATPG algorithms are single-threaded. The current trend in microprocessor design is the development of multi-core processors. Therefore, single-threaded ATPG algorithms do not use parts of the available computing power. However, TIGUAN uses the multi-threaded SAT solver MiraXT [LSB07] and can, therefore, utilize the performance of multi-core processors and control the number of threads. Since the use of multiple threads is a clear overhead for easy-to-test faults, TIGUAN proposes a two-stage procedure. In the first stage, TIGUAN is run only single-threaded within a short time interval. If no solution can be found within this short period, the approach utilizes thread parallelism for classifying the remaining hard-to-detect faults. In [CPE+09], TIGUAN is extended to be applicable in ATPG using dynamic compaction.

ATPG using an IG as problem representation was first introduced in TRAN [CAR93] and Nemesis [Lar92]. These first approaches modeled only binary relations between signals as implications. The IG model was incomplete, since ternary and k-nary ($k > 3$) relations had to be checked explicitly. This was improved by IGRAINE (Implication GRaph-bAsed engINE) [TGA00]. Here, the graph model was extended to ternary relations by using additional $\wedge$-nodes. k-nary relations can be transformed into multiple ternary relations. Graph analysis techniques are used to derive indirect implications. Furthermore, IGRAINE introduced a general method to transfer circuits represented in various logics to the IG model and provides a framework for efficient justification and propagation. IGRAINE was integrated into the ATPG tool TIP.

The IG model was further extended by SPIRIT (Satisfiability Problem Implementation for Redundancy Identification and Test generation) [GF02]. This approach improves the data structures in such a way that k-nary relations can directly be represented in the IG model. Furthermore, common structural ATPG techniques such as X-path check, unique sensitization and learning techniques are transferred to the IG model. Therefore, SPIRIT combines advantages of both worlds. However, the disadvantage of the proposed data structures is the large overhead for complex gates and gates with many inputs, respectively.

2.4.2 ATPG for Delay Faults

Classical approaches for ATPG for delay faults are presented in this section. Generally, many ATPG algorithms for delay faults are based on algorithms originally proposed for the SAFM. However, the quality aspect and the involvement of consecutive time frames require a different handling of the test generation procedures. First, test generation algorithms for the PDFM are presented. Then, algorithms for the TFM are reviewed.

2.4.2.1 Path Delay

Four different classes of ATPG algorithms for PDFs are identified:

- Algebraic algorithms
- Non-enumerative algorithms
- Structure-based algorithms
- SAT-based algorithms

Furthermore, the PDF test generation problem can be formulated as a stuck-at fault test generation problem as shown in [SBS92, GBA97]. However, this is not further discussed here, since the focus in this book is on direct PDF test generation formulation.

Algebraic algorithms do not work on the circuit structure, but on Boolean expressions, e.g. represented as BDDs. The approach in [BAA92] converts the circuit and the constraints that have to be satisfied for a delay test to BDDs. A pair of constraints is considered for each fault. Each constraint corresponds to one of the two time frames. Robust as well as non-robust tests are then obtained by evaluating the BDDs. The tool BiTeS [Dre94] constructs BDDs for the strong robust PDFM, i.e. for generating hazard-free tests. Instead of generating one single test pattern for a PDF, the complete set of tests for one fault is generated directly using BDDs.

Non-enumerative ATPG algorithms do not target any specific path, but generate tests for all PDFs in the circuit. Hence, the problem of the exponential number of paths in a circuit is avoided. The first non-enumerative ATPG algorithm was NEST [PRU95]. NEST considers all single lines in the circuit rather than the exponential number of paths and uses a greedy approach combined with fault simulation to generate many tests quickly. The approach does not perform well on poorly testable circuits due to the greedy nature. The approach ATPD [TK99] improves the greedy procedure of NEST in order to detect more PDFs by one test. RESIST [FPR94] does not use a greedy procedure but exploits the fact that PDFs are dependent, because many paths share sub-paths. Therefore, RESIST does not enumerate all possible paths, but sensitizes sub-paths between two fanouts, between input and fanout, or between fanout and output, respectively. The approach sensitizes each sub-path only once and, consequently, decreases the number of sensitization steps.

Both algebraic and non-enumerative approaches have serious shortcomings when applied to today's large circuits. As for the SAFM, algebraic algorithms suffer from their large memory consumption. Non-enumerative algorithms are outdated since a test set containing tests for all testable PDFs would be of excessive size. Furthermore, test generation is performed for critical paths only as described in Sect. 2.3.2.

Structure-based ATPG algorithms for PDFs work similar to the structural path-oriented algorithms for the SAFM. However, they differ in the following points. While for stuck-at fault test generation, path selection (propagation) and justification have to be performed, only justification has to be considered for PDF test generation since the path is completely specified. Additionally, algorithms for PDF test generation work on more complex multiple-valued logics. Those are needed because two time frames have to be considered. Furthermore, they were developed to ensure high quality delay tests. The justification procedures applied for path delay test generation are very similar to those applied for the SAFM. However, these procedures are performed on the basis of the dedicated multiple-valued logics. The following example shows such a multiple-valued logic.

Example 2.8. Figure 2.12 shows the Hasse diagram of the ten-valued logic proposed for robust test generation in [FWA93]. A value of this logic determines the signal behavior during two consecutive time frames. The value s denotes a static value, while $\overline{s}$ describes a non-static value. The don't care value is denoted by X and an unknown value is described by U. The lowest level shows the basic values (leaf nodes) of the logic and the upper levels present the composite values.

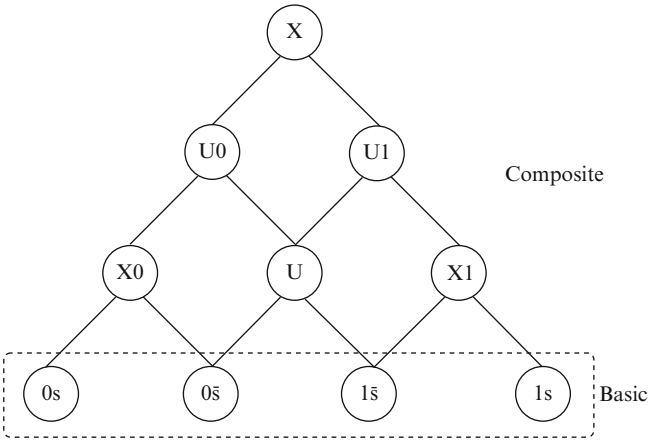

Fig. 2.12 Hasse diagram of ten-valued logic [FWA93]

Composite values are sets of basic values. For instance, the value $X0$ in the Hasse diagram describes the value set $\{0s, 0\bar{s}\}$ and the value X covers all other values contained in this logic.

The approach in [LR87] uses a five-valued logic to generate test patterns and introduces the general robust sensitization criterion (see Sect. 2.3.2). The underlying algorithm for the test generation is PODEM. The approach DYNAMITE [FFS91] is based on SOCRATES and proposes a ten-valued and a three-valued logic for robust and non-robust test generation, respectively. DYNAMITE provides a stepwise path sensitization procedure which is capable of proving large numbers of paths as untestable by a single test generation attempt. Consequently, DYNAMITE is very effective for circuits with a large number of untestable PDFs. The approach in [FWA93] enhances this scheme by using five different logic systems, e.g. a ten-valued and a 51-valued logic system, suitable for various test classes such as non-robust, robust and restricted robust. The work in [BAA98] is concerned with the derivation of an optimal set of logic states to minimize the number of backtracks during PDF test generation.

SAT-based algorithms work differently from those presented in this section since they do not work on the circuit structure. The problem of generating a test for a PDF is transformed to a Boolean SAT problem. The SAT problem is solved by a SAT solver. The SAT solution is then transformed into a solution of the original problem, i.e. a PDF test pattern. A detailed description of the basic SAT concepts is given in Chap. 3. The basic techniques for SAT-based ATPG are presented in detail in Chap. 4. The relevant SAT approaches are briefly described in the following for the sake of completeness.

The first SAT-based approach for PDF test generation was proposed in [CG96] where a seven-valued logic is used to generate robust tests for PDFs in combinational circuits. A Boolean encoding is applied for the transformation into

a Boolean SAT problem. The underlying SAT engine was TEGUS. SAT-based learning techniques are applied in [KWMS00, CH05] to speed up PDF test generation. These approaches are all CNF-based algorithms. The IG-based approach IGRAINE [TGA00] was applied to target non-robust and robust test generation. The tool KF-ATPG is presented in [YCW04]. Unlike the above mentioned SAT-based approaches, KF-ATPG uses the circuit-based SAT solver presented in [LWCH03]. Therefore, KF-ATPG is able to exploit structural knowledge of the problem to speed up test generation. SAT-based approaches for PDF test generation will be described in more detail in Sect. 8.1.

2.4.2.2 Transition

Early approaches for the GDFM, e.g. [HRVD77, IRS88, PR88, SA89, MGÖD90, MC92, PM92, Mah93], suffer from the circumstance that the delay defect size involves costly calculations. Unlike for the GDFM, timing information does not have to be leveraged during the search for a TF test pattern. Concentrating on the logical behavior of the delay fault and disregarding the actual delays leads to more efficient test generation. Furthermore, existing test solutions can basically be reused with slight modifications since the TFM is very similar to the SAFM. Both "simplicity" and the reuse of existing algorithms is the reason for the wide use of the fault model – in spite of the lesser accuracy, i.e. the gross delay fault assumption.

The first deterministic ATPG algorithm for the TFM was presented in [LM86], where a modified D-algorithm was used for TF test generation in combinational circuits. The work presented in [Che93] deals with the particularities of testing TFs in sequential circuits. Since the existing stuck-at test generation algorithms are adopted for TF testing, TF testing consequently benefits strongly from the increasing efficiency of stuck-at test generators. However, as mentioned above, the efficiency comes at the expense of the accuracy and the ability to detect small delay defects. Stuck-at fault test algorithms are highly optimized to generate tests as fast as possible. Usually, shorter propagation paths are chosen since these paths are easier to sensitize.

This is contrary to the desired test quality requirement for the TFM where longer paths are preferred since small delay defects can be accumulated. This is called the *criticality* problem in [SPR02]. Therefore, many approaches were proposed to raise the quality level of TF tests. The gate delay test generator DTEST_GEN [PM92] can be considered as a first approach for test quality enhancement for the TFM. This approach can be seen as point of origin for several approaches with the aim to generate a TF test set with increased quality. In particular, a good trade-off between high quality tests, e.g. tests sensitizing longer paths, high fault coverage and efficient test generation, is desired.

The criticality problem of TF test generation is highly related to the problem of identifying critical paths for PDF testing. This issue is not included in depth in this book. The work [SPR02] presents the path-oriented TF test generator POTENT.

This approach uses structural PDF test techniques to search for the longest testable path passing through the fault site, as well as a PODEM-based test generation algorithm to generate a TF test pattern.

The approach TranGen [YCW04] employs a circuit-based SAT solver (see Sect. 3.5) to generate tests for path-oriented TFs. That means, the longest propagation path is identified and the ATPG algorithm will try to sensitize the path in a static manner, i.e. all side inputs have to assume static values. If the longest path is not sensitizable, the next longest path is chosen and so on. Conventional TF ATPG is performed if no path is sensitizable. Learning is done by identifying unsensitizable path segments which are stored in a circuit graph. However, the approach is based on path enumeration and suffers from the excessive number of paths in modern circuits.

An ATPG framework for generating high quality tests for TFs is presented in [KMT+06]. This framework is based on two methods: activation-first and propagation-first. Activation-first finds and fixes the longest functional activation path and then performs path propagation, whereas propagation-first finds and fixes the longest functional propagation path and performs fault activation afterwards. For each fault, it is dynamically decided which method is probably more promising. The underlying ATPG is based on SOCRATES.

Timing-aware ATPG is proposed in [LTW+06]. This approach works similar to the propagation-first method from [KMT+06]. Timing information is calculated in a pre-process. During test generation, the fault effect is propagated to a PO through the longest path first. Then in a second phase, all lines needed for justification of the propagation path are justified. Timing information is further used to maximize the transition arrival time at the fault site. As a result, the transition is launched as late as possible. The disadvantage of this method is the high run time as reported in [YCT08]. A SAT-based ATPG approach using timing information is proposed in [SCS+11]. Here, the path length calculation is encoded into the SAT instance and the enumeration of testable paths is focused. However, no transition-dependent delays are encoded using the approach.

2.5 Industrial Test Environment

Generally, a test pattern generation is not executed as a stand-alone program in industry, but is part of a larger test environment. Such a basic industrial test environment is exemplarily introduced in this section. The general flow in a test environment is given in Fig. 2.13. The flow can be divided roughly in the following three phases: netlist compilation, *Design-For-Test* (DFT) insertion and ATPG. The last one is described in more detail, because the techniques proposed in this book are applied in this phase.

First, the design is read in the *netlist compilation* phase from a hardware description language and compiled into a netlist consisting of a set of primitive gate types (see for example the gate types given in Fig. 2.3). Next, *DFT insertion* is performed. Test-specific components are inserted into the design in this phase.

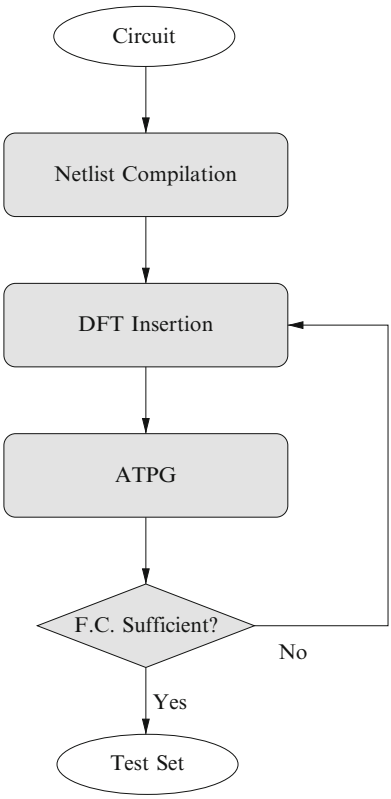

Fig. 2.13 Flow in industrial test environment

This includes for instance scan-chains and test points. After the DFT insertion, the ATPG phase is executed and a test set is generated. If the obtained fault coverage is too low, it is returned to the DFT insertion phase. Here, additional efforts can be undertaken to increase the fault coverage, e.g. test point insertion. Then, the ATPG phase has to be executed again.

In the ATPG phase, several methods interact to obtain a fast and effective ATPG system. Initially, a fault list is created containing all faults for a certain fault model which have to be tested. Random test patterns are generated and fault simulated (*Random Test Pattern Generation*, RTPG) . Fault simulation is very efficient – linear in the number of gates. All faults detected by these pattern can be removed from the fault list. By this, a large number of faults can easily be pruned. RTPG is typically performed until a defined fault coverage is reached. The remaining yet undetected faults are then targeted by *Deterministic Test Pattern Generation* (DTPG). Each generated test pattern is again fault simulated to find additional faults detected by this pattern. All detected faults are removed from the fault list until the fault list is empty.

Depending on the level of test set compaction which is executed, the interaction of the methods differs. Test set compaction is important, because a large test set

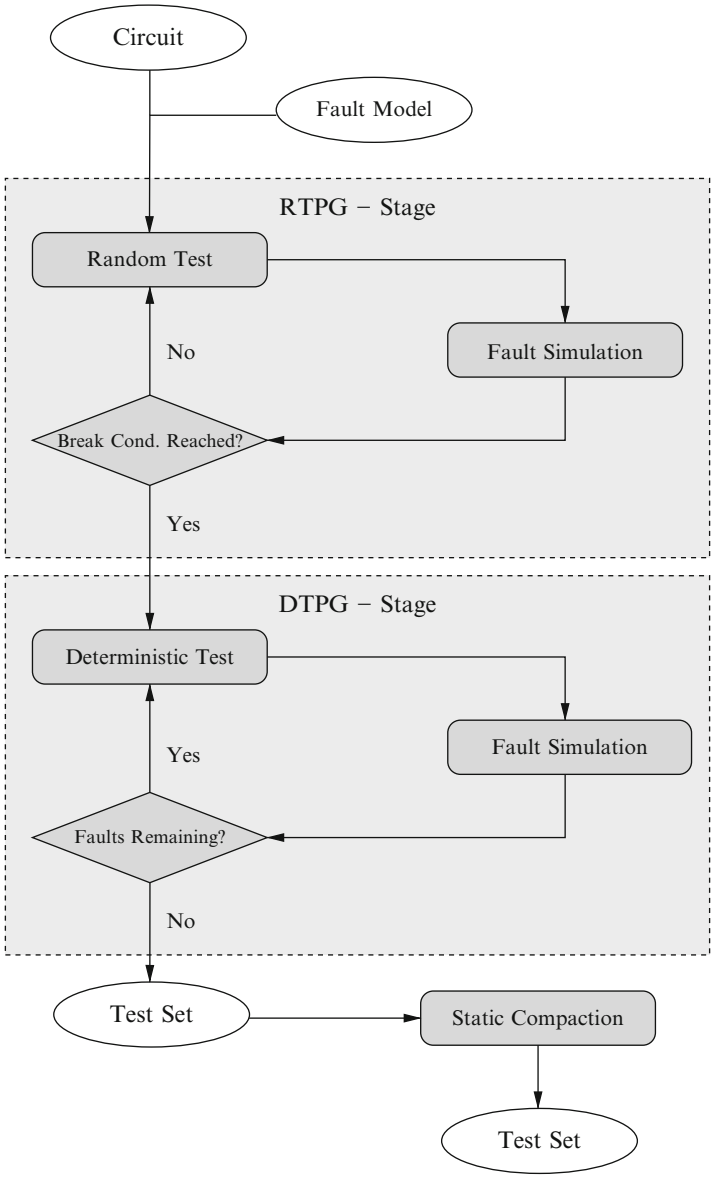

Fig. 2.14 ATPG phase – low compaction

leads to more test costs. In this description, it is distinguished between low and high compaction. Both flows are shown in the figures below. For low compaction (shown in Fig. 2.14), RTPG and DTPG are performed as described above. A static compaction method is applied after both stages have been finished. Here, those test patterns are removed, which only detect faults that are also detected by at least one

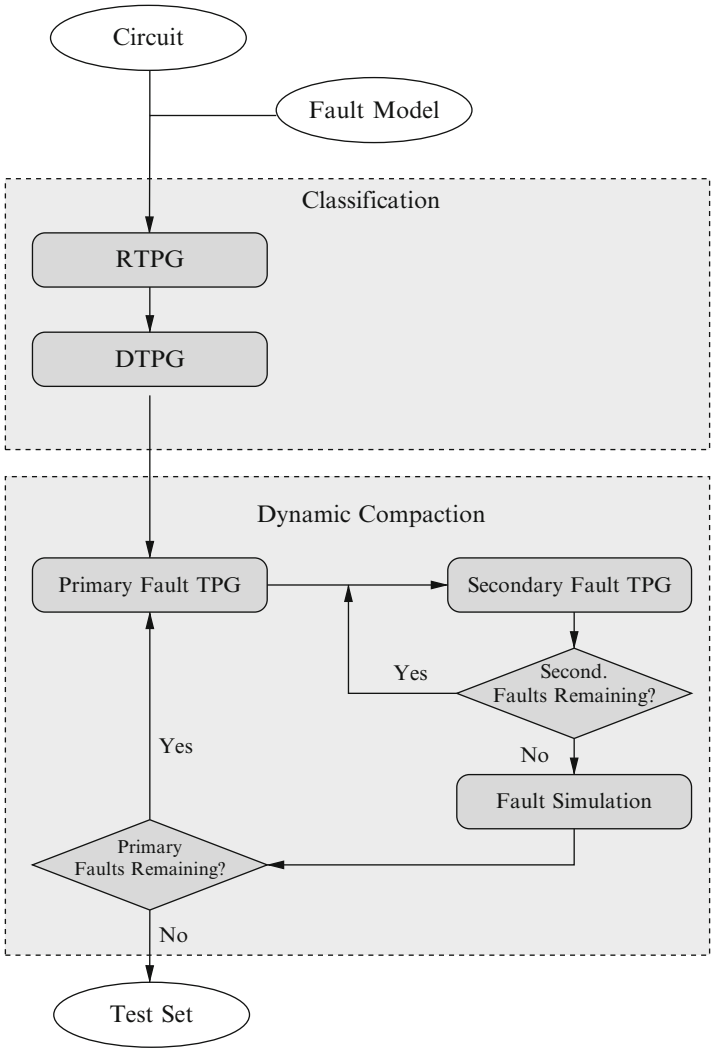

Fig. 2.15 ATPG phase – high compaction

other test pattern. Additionally, the tests are merged. Multiple test patterns which all have non-conflicting input assignments can be merged into a single test pattern. The number of don't care assignments should be generally as high as possible in order to achieve a good static compaction rate.

A dynamic compaction scheme is applied in the high compaction stage (shown in Fig. 2.15). The DTPG procedure is modified in order to generate a test pattern which detects one primary fault and as many secondary faults as possible. This is usually done by generating a test pattern for the primary fault, fixing the calculated input assignments and trying to detect other secondary faults under the assumption

that the fixed inputs assignments hold. For detecting secondary faults, it is looped over all the yet undetected faults of the fault list. Again, a low number of don't cares is desirable to achieve a good compaction rate. In order to reduce the computational overhead of this method, all untestable and too-hard-to-detect faults are filtered out in a *classification stage*. In this phase, RTPG, DTPG and fault simulation interact as described above for static compaction. However, the test patterns are not retained.

http://www.springer.com/978-1-4419-9975-7

High Quality Test Pattern Generation and Boolean Satisfiability

Eggersglüß, S.; Drechsler, R.

2012, XVIII, 193 p., Hardcover
ISBN: 978-1-4419-9975-7