Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN
SCIENZE DELL'INGEGNERIA

CICLO XII

COORDINATORE Prof. Stefano Trillo

A SAT based test generation method for delay fault testing of macro based circuits

Settore Scientifico Disciplinare ING-INF/05

Dottorando	**Tutore**
Dott. Mele Santino	Prof. Favalli Michele
_______________________________	_______________________________

Anni 2007/2009

Contents

Chapter 1

Introduction

The main activities performed during the tree years of my PhD are related to the timing failures problems in module-based CMOS VLSI circuits.

The attention to module-based (or block-based) circuits follows the current VLSI physical design trends that attempt to limit the parametric failures due to the scaling of technology toward nanometric feature sizes. In such technologies, in fact, the traditional design paradigms that are based on small (i.e gate level) cells may produce high levels of variability, thus resulting in parametric defects. The use of highly regular cell structures, called *logic bricks* has been proposed to solve these problems thus increasing the yield of VLSI circuits. A brick comprises a logic function created from a small set of logic primitives that are mapped on to a micro-regular fabric. Such logic function is typically more complex that those implemented in traditional VLSI libraries.

Field Programmable Gate Array (*FPGA*) technology also exploits a module based design approach. Unlike logic bricks, FPGAs are completely programmable, because they are based on look up tables (a n-bit LUT can accomplish every n-bit function), but the drawback is related to the implementation of the LUT, that is unknown to designer and not optimized for regularity.

In this scenario, the delay fault testing became a big issue, since it is very difficult to study a circuit built using modules whose implementation in not known, either for technological and for intellectual property reasons. Moreover, the aggressive construction technologies and the high speed of clocks make the need for delay fault testing more relevant.

The main PhD activity, that will be explained in detail in this thesis work, is related to a new method that we propose to generate *test vectors* for *path delay faults* in *circuits based on modules.* Such method exploits a functional approach at module level and a structural one at circuit level.

In particular, we consider *single path delay fault testing* in a combinational circuit or in an (enhanced) full-scan one that is composed of functional blocks whose implementation is not known. We identified suitable conditions so that a test pair is able to propagate a transition through the path under test, in order to detect a path delay fault. Also, additional conditions to prevent invalidation of tests by hazards have been identified. We suppose that the dynamic behavior of the block is modeled using input delays such as in the *timing arc delay model.*

We target simple combinational blocks such as logic bricks, that are expected to present up to 8-10 inputs and a low logic depth. The used method is scalable, to generate conditions for path delay fault tests also at gate level.

In order to assess the feasibility of the proposed approach, I realized a software, written in *C/C++*, that permits to find out *robust* and *non-robust test pairs*, starting from the *BLIF* description of a *module based circuit.* Such a software uses a *BDD* description of the blocks' functions on which we apply *Boolean Differences* to obtain local sensitization conditions at module level. Since there are circuits whose BDD structure may be very large and it may be inefficient (in some cases also infeasible) to treat it, we translate functions obtained at macros level to a *CNF* description. After that, a *SAT solver* generates the test pairs at circuit level starting from the conjunction of all the CNF functions.

The software tool was used to verify the proposed approach on a set of benchmarks (both combinational or full-scan) from ITC'99 and ISCAS'85 sets. Such benchmarks allowed to show the feasibility of the proposed approach, although they are not fully representative of the target circuits for which the method was developed.

Another significant work, carried out during my PhD period, also deal with testing of macro-based circuits, but it concerns specifically *logic bricks.* In particular, a method for high quality *functional fault simulation and test generation* for such circuits was conceived and a software tool that implements it was developed.

For both the approaches, results showed the feasibility of them, but also highlighted several possibilities to improve and extend the work done. This thesis is organized in the following way: in section 2 a presentation of main digital ICs testing issues involved in this work is provided, specially whom related to macro-based circuits. In section 3 the motivation and the aims of the work is presented, while the conceptual model is discussed in more detail in section 4, followed by a round-up of the mathematical and technical instruments used to model it, in section 5. Presentation and discussion of the results, obtained from experimental measurements, are given in section 6. Conclusive remarks and future works are presented in 7.

Finally, in appendix A high quality *functional fault simulation and test generation* method for logic bricks will be presented.

Chapter 2

Digital ICs testing in macro based circuits

Several kinds of tests are performed on a digital device during its life cycle, depending on the life phase of the circuit, on the specific application of the circuit and on others economical issues. In general, the entity that performs the different types of tests also varies.

When a new circuit is designed and realized, it needs a verification of the design and the test procedures; this is the *verification testing* and aims to validate the correct functioning of the circuit. Verification testing is performed generally with a big involvement of the chip designers.

After this phase, the chip goes in the production phase, when it needs to be checked mainly for manufacturing defects; the set of such tests is called *manufacturing testing*.

Finally, the chip is tested by the customer (often a systems manufacturer), that execute the so called *acceptance testing*, or *incoming inspection*.

It is possible to identify two big classes of tests [10], *parametric* and *functional*.

Parametric Testing. These tests verify if the values of AC and DC characteristics of the circuit are within the operational range. Examples of DC parameters that can be tested are: output current and voltage level, leakage current, threshold levels and input/output impedance; the AC parameters can be: propagation delay, setup and hold time, bandwidth and noise. Since these test are related to circuit's characteristics, they are generally *technology-dependent*.

Functional Testing. These tests aim to verify if the circuit is able to accomplish the function for which it was designed. They are performed stimulating the circuit with input vectors and checking if the values of outputs or internal nodes are correct. This work deals with these types of tests, in particular with the generation of the test vectors for path delay faults.

2.1 Fault Models

As stated above, the *functional test* of a combinatorial circuit consists of the sensitization of the circuit by a test vector and the subsequent check of the circuit's nodes logic values. Such nodes can be internal or can be the outputs of the logic net. Since it is impossible to generate test patterns for real defects, abstracts models of defects are needed.

A *fault model* is a representation of a circuit's defect (or class of defects) behavior. The *fault model* is necessary to the test pattern generation process and to the output response check. As stated in [62] a good fault model should satisfy two criteria: (1) it should accurately reflect the behavior of defects, and (2) it should be computationally efficient in terms of fault simulation and test pattern generation.

Although several categorization concerning fault models have been proposed, in this paragraph only a brief list of the most known models will be given, while a focus will be posed on delay fault models, later in this chapter.

Stuck-at faults. This model represents faults that have substantially three properties:

1. the fault site can be any node of the circuit (both an *input* or an *output* of a gate)
2. the faulty line is *permanently* set to *0* or *1*
3. if *only one line* can be faulty it is called *single stuck-at model*, else it is called *multiple stuck-at model*.

A schematic representation of stuck-at faults is shown in figure 2.1.

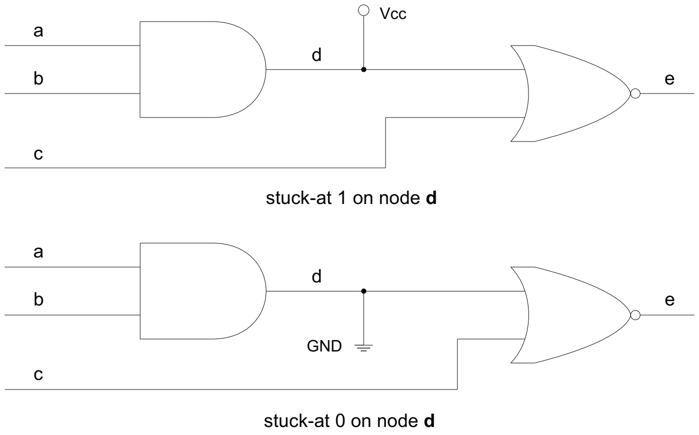


Figure 2.1: Examples of stuck-at faults

Transistor faults. These are CMOS device faults models; they are divided in:

- *stuck-open faults*; they represent a broken line in a transistor and can drive to a sequential behavior of the gate
- *stuck-short faults*; they represent a short between two lines in a transistor and can drive to an anomalous power consumption in steady state.

Schematic examples are shown in figure 2.5.

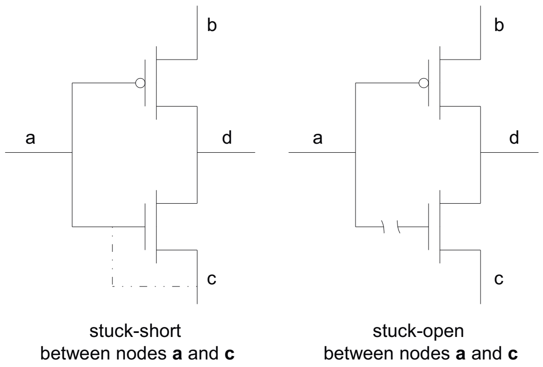


Figure 2.2: Examples of transistor faults

Bridging faults. These models identify a low resistance path between two nodes of the circuit; they are modeled by inserting a short (a 0 Ω resistance) between the nodes. They take different names depending on which kind of nodes are bridged together. A bridging fault can arise between:

- *two terminal of a transistor*; this category falls in that of transistor faults
- *two nodes of a gate*; this category is divided again depending upon the presence of a *feedback path* originated from the bridging fault.

Schematic examples are shown in figure 2.3.

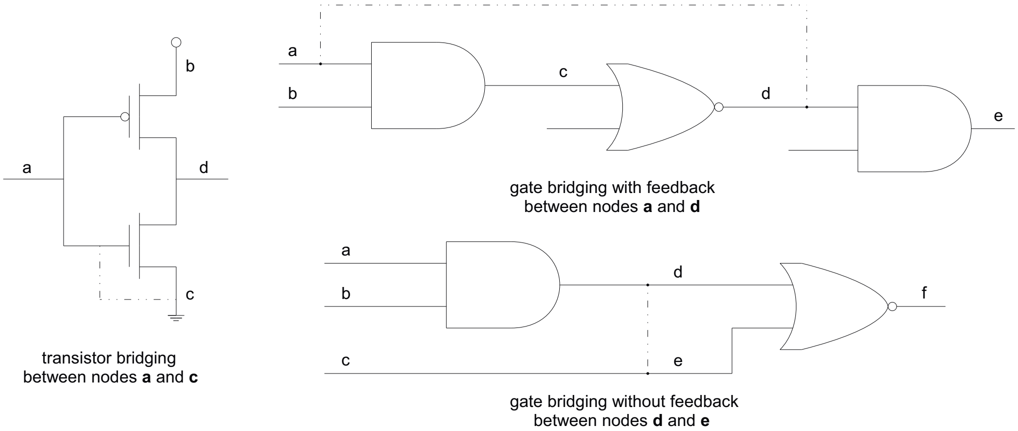

b
a
d
c
transistor bridging
between nodes **a** and **c**
a
b
c
d
e
gate bridging with feedback
between nodes **a** and **d**
a
b
c
d
e
f
gate bridging without feedback
between nodes **d** and **e**

Figure 2.3: Examples of bridging faults

Delay faults. They model faults that change the timing of a circuit (or a part of it) without changing its functionality. A delay fault can arise when there is an extra delay on a gate, interconnection, or memory element of the circuit. So, when the circuit is tested with a slow clock, it works properly, while testing it with at-speed clock, it may fail, because the circuit timing constraints have been violated.

Later in this chapter, the delay fault model will be discussed in more detail, with particular regard to the path delay fault model, since it will be exploited in this work.

2.2 Automatic Test Pattern Generation (ATPG)

As seen before, the aim of the testing is to identify faulty circuits; to do so it is necessary to feed the circuit under test with a certain number of *test patterns*, that are input vectors capable to sensitize and find out a specific class of faults.

Since VLSI circuits are growing in dimension and density, an exhaustive test (i.e., a complete check of all the possible sites of faults under all the possible sensitization conditions) is infeasible. So an automated and sophisticated system to find test patterns efficiently is required. This task is called *Automatic Test Pattern Generation.*

Test pattern generation is a computationally complex problem, and is a central question in testing, because of the relevant theoretical concerns related to the algorithm development, but also for the implication in the circuits design. To develop an ATPG algorithm, a list of faults to be revealed is required; this list can be derived from the circuit description and a list of fault models.

The target in the ATPG algorithm development is to have an high *computational efficiency* and an high *test vectors quality.* The computational efficiency is composed of two parts, the efficiency in test generation phase and during the application of the test vectors to the circuit (in general this latter component is proportional to the number of the test vectors to apply). The quality of the test vectors is evaluated in terms of *fault coverage*, that is the percentage of detected faults on the total number of faults, and in terms of *defect coverage*, that is the amount of fault models that the algorithm can detect.

The efficiency of ATPG algorithms varies a lot, depending on the fault model considered (and its abstraction level), the type of circuit under test, and the required test quality. In brief, the purpose of a good ATPG algorithm is to individuate a (possibly) small set of vectors with the (possibly) higher coverage for the considered set of faults, in the smallest amount of time possible.

It must be noted that the goal to find a little set of vectors can be, except for small circuits, often infeasible, due to the high computational cost, but during the *fault simulation*, a compaction of test vector is generally carried out.

Although there are automatic test pattern generation algorithms for both *sequential* and *combinatorial* circuits, the generation of test patterns for sequential circuits is more difficult as several vectors are required both to activate and to propagate a fault, because also the state of the circuit must be taken into account.

There are several methods proposed in literature to address the ATPG for sequential circuits. Some of those aim to bring the sequential circuit to a combinatorial description, using specific techniques.

When it is impossible to have a combinatorial-like situation, there are approaches that extend combinatorial algorithms to the sequential condition (e.g. building sequences of the circuit's combinatorial part). Others techniques use a functional description of the circuit, to work on a state machine of the circuit.

There are design techniques that aim to enhance the testing capability of the circuit. Design of circuits using some of such techniques is called *Design For Testability (DFT)*. In particular the *scan techniques*, between others things, permit to treat a sequential circuit as a combinatorial one, from the point of view of testing. In fact, since the flip-flops delays can be always included in the combinatorial part of the circuit, then it is often possible to study every combinatorial stage as a single combinatorial circuit, from the point of view of delay faults.

Combinatorial ATPGs are divided in two big categories, listed above.

Pseudo random. These techniques generate random test patterns; when a predefined coverage value is reached, the algorithm is stopped.

Deterministic. These techniques generate test patterns starting from the logic conditions that permit the detection of faults. There are several algorithms falling in this category; a class of them bases the test pattern search on the *structure* of the circuit; the most known ones are *D-Algorithm* [46], *PODEM* [25] and *FAN* [23]. There are also algorithms basing the search on the *function* of the circuit, i.e. independently from the topological information; this is the case of *boolean differences*-based approaches, as described in [32].

2.3 Delay Fault Testing

In order to obtain fault free operations in *synchronous sequential circuits*, a signal have to propagate its logic value correctly within a specified time (the clock period) along a path between two memory elements. If the signal propagation time exceeds the clock period, then a delay fault occur. A generic sequential circuit can be viewed schematically as shown in figure 2.4, that is a certain number of combinatorial areas between registering stages (composed by flip-flops or latches).

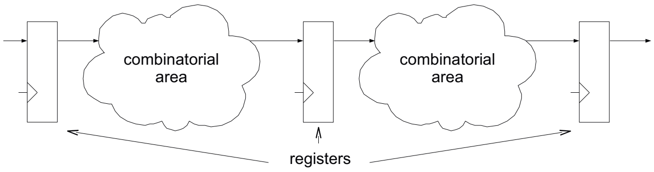


Figure 2.4: Scheme of a sequential circuit

Delay fault testing consists of propagating a transition through a block of combinatorial logic and checking if the resulting transition arrives within a predefined amount of time.

The example in figure 2.5 shows the propagation of a transition trough a combinational stages. Two possible scenarios are shown; in the first, the propagation delay is shorter than the clock period, while in the second, the combinatorial delay exceeds the time budget. The effect is that the flip flop B sample a wrong value on output c.

Unlike the logic faults, delay faults need two test vectors (a test pair) to be checked, the first one to propagate the initialization value (often given with a slower clock) and the second at speed, to verify if the outputs change their values before the clock event occur. The scheme used to obtain this, is illustrated in figure 2.6, where a combinatorial block is fed by a latch to load input values while another latch stores output values.

DF Models. Before to present the principal delay fault models, some terminology remarks must be given.

A *path* is a route between a *primary input (PI)* and a *primary output (PO)* of a circuit. A path can be represented in a schematic way as shown in figure 2.7.The primary input of a path under test is called *path root*, the

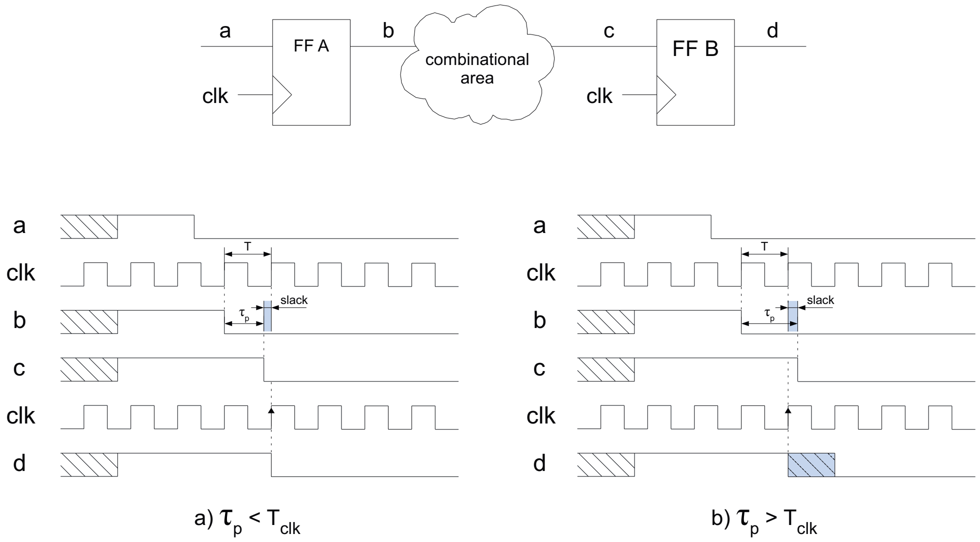


Figure 2.5: Propagating a transition

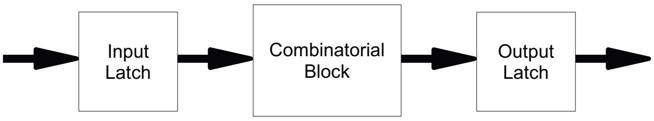


Figure 2.6: Delay fault testing

input on the path of a single gate/block is called *on-path input*, while the others inputs of the gates/blocks are called *side inputs*.

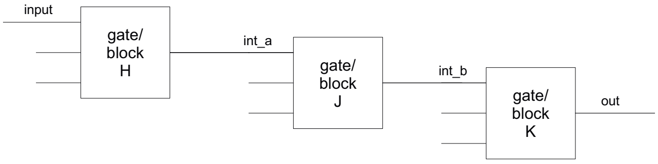


Figure 2.7: Schematic representation of a path

Note that the on-path input of the first gate/block of the path is the path root. Figure 2.8 shows a graphical representation of such concepts.

There are three main types of delay fault models, namely *gate delay fault model*, *transition fault model* and *path delay fault model*. A *gate delay fault*, as stated in [52], is a gate defect that results in at least one path

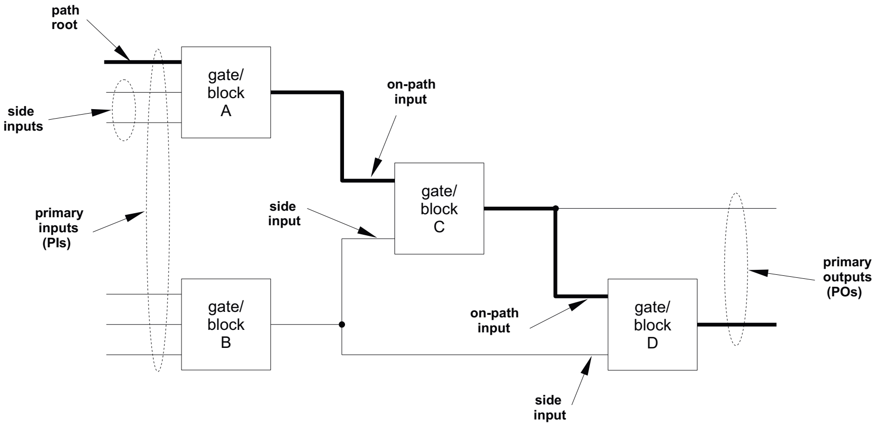


Figure 2.8: Path related nomenclature

fault, while a *path delay fault* occur when a transition applied to a path input does not arrive to the path output in time to set the output memory element. The major problem with the path delay fault model is the number of paths to be considered, in fact, in a circuit, the number of paths can became exponential in the number of gates.

The *transition faults* was defined in [61] by means of a *slow-to-rise* or a *slow-to-fall* fault on the output of a gate. The first models a delay defect during a rise transition, while the latter during a fall transition. If the defect on the gate is large enough to affect any path passing through it, then a transition fault exists on the specific gate.

It must be emphasized that a path delay fault is associated with an entire path, so it models a *distributed defect*, while gate delay faults and transition faults represent *local defects*; also the latter can be viewed as stuck-at faults on circuit's signals.

2.3.1 Path Delay Fault Testing

"A *path delay fault* is a path of the combinational network between input and output latches for which a transition in the specified direction as initiated by the setting of an input latch does not arrive at the path output in time for proper setting into an output latch."[52]

A circuit path has a delay fault if the timing budget of the path is greater than the clock period; such a situation may (or may not) result in a functional error at the output(s) of the entire circuit, because of the sensitization conditions, that are not justified.

The *path delay fault model* considers the delay of the entire path, hence the potential delay fault is seen as distributed along the entire path. In this way, the major limitation of the others delay fault models is overcome, because the gate delay fault and the transition delay fault address local defects, while path delay fault, models global faults.

Since the path delay fault arise when the sum of the delays of the gates along the path exceed the clock interval, no matter the size of the delay of the single gate is. Also, in the path delay fault model no assumptions about the localization of the fault within the path are made, so such a model comprise also the gate delay fault model. Moreover, the amount of the exceeding time is not relevant for testing.

In order to construct a test for path delay fault in a circuit, two test vectors are required. First, the *initialization vector*, brings the circuit to the initial state using a clock with a period that allows to all signals in the circuit to stabilize (i.e., not necessarily at speed). Then the *propagation vector* activates the fault; it permits to see the fault effects on the primary outputs. The latter stage is performed using the system clock. Must be noted that, during path delay fault testing, the circuit is assumed to be fault free, obviously except for delay faults that may affect the circuit.

For every path in a circuit, up to two path delay faults can exist, one related to the propagation of the *rise transition* (i.e. $0 \rightarrow 1$) applied to the on-path input and the other one related to the application of a *fall transition* (i.e. $1 \rightarrow 0$) to the on-path input. Hence the maximum number of possible path delay faults in a circuit is $2 * n_\pi$, where n_π is the number of paths in the circuit. In the following of this work, the association between a path and one of transitions will be referred to as *pattern*, so, for every path, two possible patterns can be sensitized. Note that this is true only in case of gate level circuits which does not use *xor* gate; in fact, as it will be observed in more depth later, *xor* gates or more complex functions may permit to propagate a transition both in a positive way and in a negated one.

Not all the paths in a circuit can be tested for path delay faults, in fact, as shown in figure 2.9, from [62], a falling transition on path *a*-*b*-*c*-*e* can be propagated but a delay fault cannot be highlighted; paths with this characteristic are called *statically unsensitizable*. Instead, a path that does not permit the propagation of both the transitions, is called *false path*.

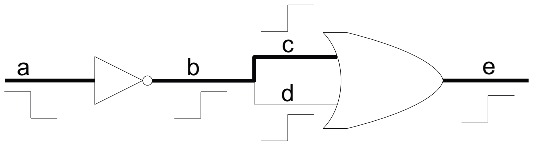


Figure 2.9: A static unsensitizable but not false path

For example, in figure 2.10, the node *b* and the node *d* cannot be at high logic state (i.e. logic '1') at the same time. Thus, the path *a-c-e* is a *false path*. Note that a false path is always statically unsensitizable, but not vice versa [62], in fact, path *a-b-c-e* in figure 2.9 is not a false path (i.e. at least a transition can be propagated).

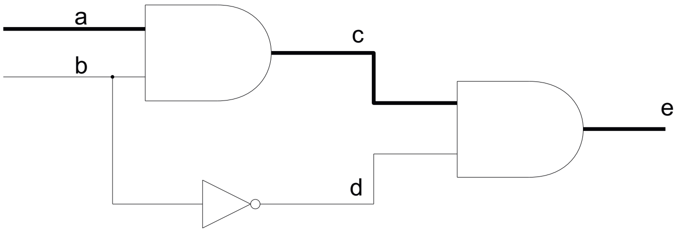


Figure 2.10: An example of false path

Sensitizability is the main requirement to test the paths for path delay faults, but ATPG algorithms have to find test pairs for paths with particular sensitization properties. Such properties are related to the constraints required from paths to be tested, hence are related to the test quality. These matters will be discussed in more detail in the next paragraph.

2.3.2 Test Pattern Generation for Path Delay Fault Testing and Robustness

The delay fault test generation task consists in the search of the test pairs $\langle u, v \rangle$, such that, after applying u to the PIs of the circuit, the application of v produces a wrong output, if sampled with the test clock. ATPG for path delay fault, however, has a peculiarity, in fact test pairs can be characterized depending to the constraints needed for paths testability.

To do such a characterization, the *robustness* concept, defined in [35] is required. In [51], a possible classification of robustness classes was proposed; a schema is shown in figure 2.11, from [51]. The basic concept is that smaller is the set of path delay faults, less restrictive constraints are required and hence higher is the quality of the test pairs associated.

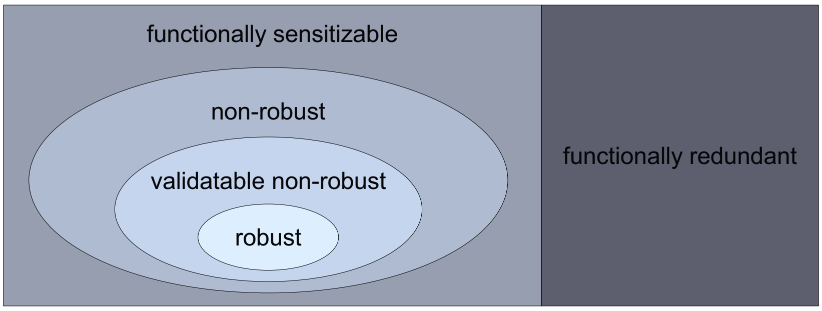


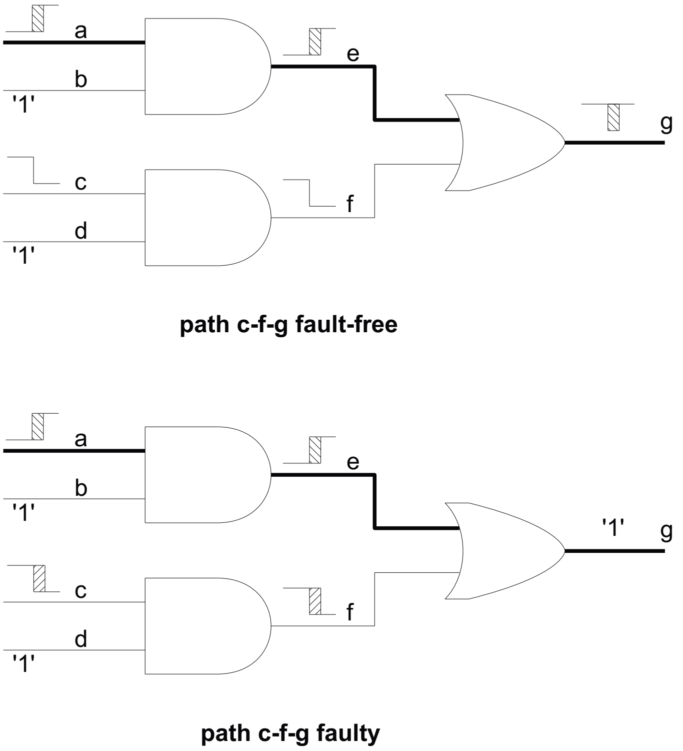
Figure 2.11: A classification of path delay faults

Robust Testability. If a test pair is able to find the path delay fault regardless to the presence of a delay fault in the others paths of the circuit, this test pair is a *robust test*. In figure 2.12 is explained a situation in which a delay on one of the off-paths can mask the detection of the delay fault of the path under test.

The path under investigation is *a-e-g*, the test pair V_{nr}={⟨*a=0, b=1, c=1, d=1*⟩,⟨*a=1, b=1, c=0, d=1*⟩} can detect a delay fault on path *a-e-g*, but if there is a fault on path *c-f-g*, it could be impossible to establish that a delay fault arise. In such a situation, the couple of vectors V_{nr} is a *non-robust test* for the path *a-e-g*.

In figure 2.13, there is an example of robust test, instead. The circuit is the same and the pattern too. In order to robustly sensitize the path *a-e-g*, the test pair V_r={⟨*a=0, b=1, c=0, d=1*⟩,⟨*a=1, b=1, c=0, d=1*⟩} is required, in fact, as can be seen, a delay fault in path under investigation (*a-e-g*) is still detected, even if the path *c-f-g* is also affected by a path delay fault. A path that have at least one robust test is said to be *robustly testable*.

So, in the cited case, even if the path *a-e-g* has a non-robust test, it is robustly testable anyway. As can be seen in graphic representations, robust



path c-f-g fault-free

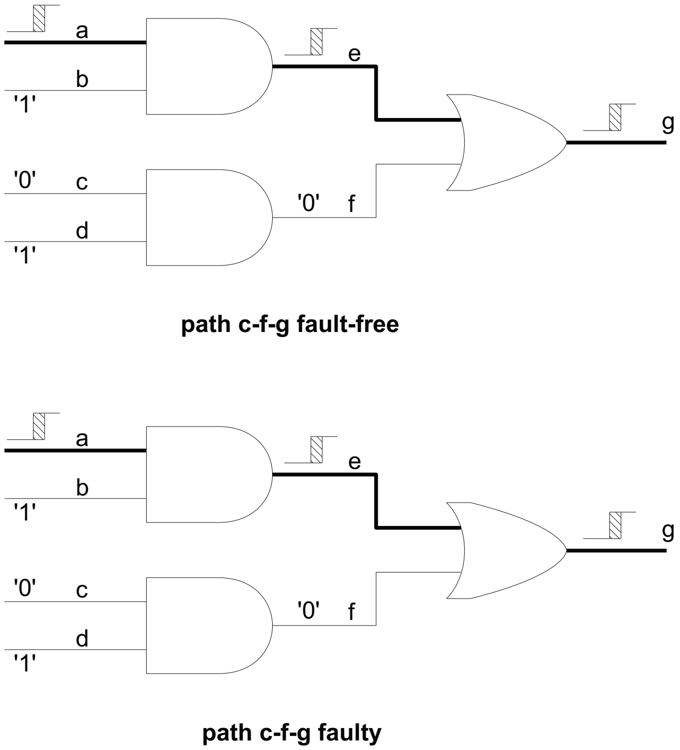
path c-f-g faulty

Figure 2.12: Non-robust sensitization

testability conditions can also be viewed as the non-robust ones, with the constraint to have no hazard propagating trough the on-path.

Between robust and non-robust paths, there is an intermediate class of testable paths, called *validatable non-robust testable (VRN testable)* [44]. As stated above, a non-robust test can be invalidated by the presence of delay faults on certain side paths. If it is possible to ensure that such side paths are fault-free, then the non-robust test is said *validatable non-robust (VRN)* [51]. Note that the side paths may be either robustly or VRN testable themselves; in the latter case the rule is applied transitively. So, a VNR test, as a robust test, is able to detect a delay fault regardless of the presence of delay faults on side paths [51].

If a test is able to reveal a path delay fault only when there is another path delay fault, then such a test is called *functionally sensitizable (FS)*. In figure 2.14 the same circuit is sensitized with the test pair V_{fs}=$\{\langle$*a=0, b=1, c=0, d=1*$\rangle$,$\langle$*a=1, b=1, c=1, d=1*$\rangle\}$. As can be seen, only if there is a delay fault of the path *c-f-g*, a delay fault on path *a-e-g* can be detected.



path c-f-g fault-free

path c-f-g faulty

Figure 2.13: Robust sensitization

All the path delay faults that do not belong to one of the previously cited categories (i.e., are not sensitizable), are called *functional redundant (FR)*.

In brief, a *robust* test pattern has the highest test quality (it will detect a defect no matter faulty conditions on off-input paths), the *non-robust* detects a defect if there is only a single fault on the path, and *FS* test patterns can only detect multiple defects on paths [36]. Last, *functional redundant* cannot detect any path delay fault.

2.4 Functional block design

With the scaling of the CMOS feature sizes, specially toward nanoscale, and the consequent increasing of integration, the number of ways in which a chip can fail, increases. In fact, failures arise due to manufacturing defects and reliability faults, but also the impact of parametric failures is increasing.

Application specific IC (ASIC) development requires the construction of ad hoc masks for every layer; this permits to have an high grade of flexibility

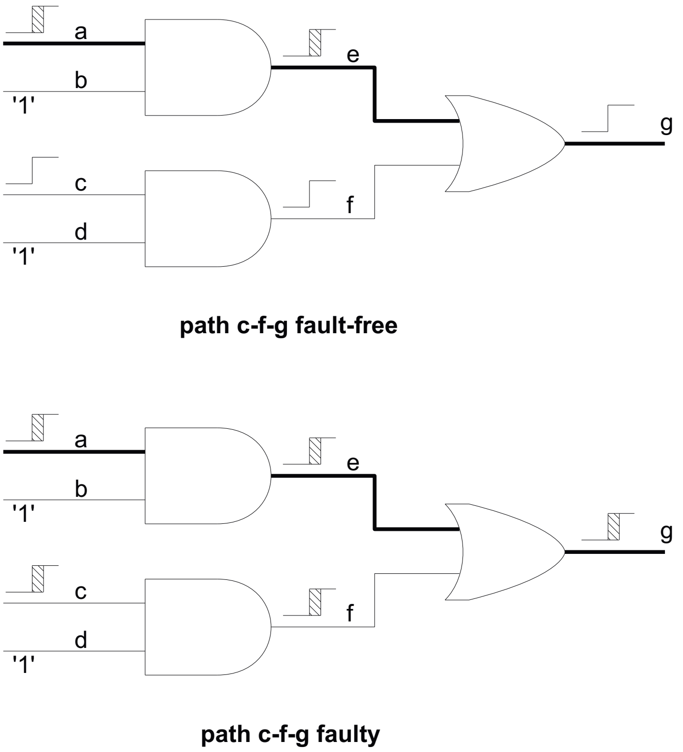


Figure 2.14: Functional sensitization

during the design, but it may became a disadvantage for the high production costs and the long time-to-market. In fact, specially with the growing of the ICs integration level and the consequent shrinking of the feature sizes, the costs for the corrective steps and silicon re-spins are greatly increased. As an example, the figure 2.15 from [65] shows the estimated trend in ASIC mask set costs.

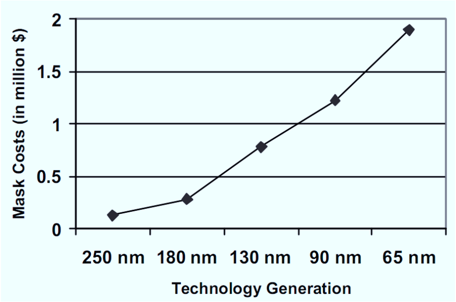


Figure 2.15: Mask Cost vs. Technology Generation

The above mentioned problems with ASICs (i.e. the design costs and the high time-to-market) are driving designers to choose solutions based on more customizable devices. In particular, the trend is addressed towards configurable and programmable solutions.

There are a lot of programmable devices, but in this work the attention will be posed on *Field Programmable Gate Arrays (FPGAs)* and *logic bricks*, since these are between the most promising technologies, specially towards *system on chip (SoC)* paradigm and *nanoscale* design. The main drawback with such solutions is the loss in terms of performance, power and die area [42]; as an example, diagrams in figure 2.16 form [42] represent a comparison between FPGA and ASIC technologies.

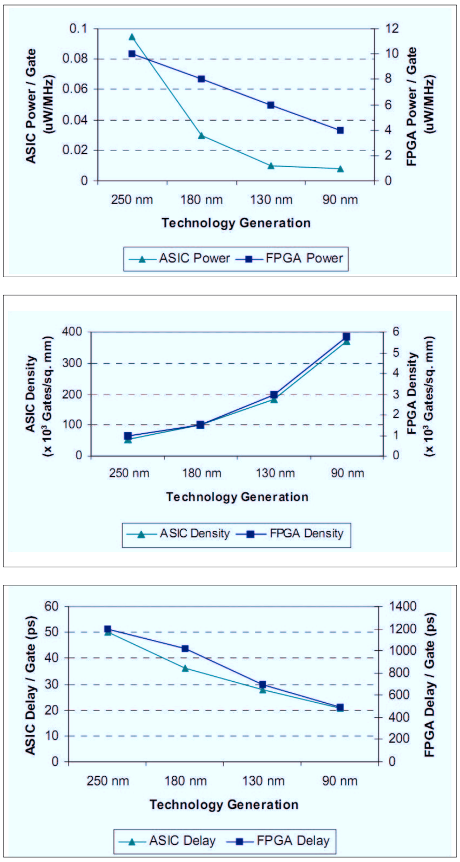


Figure 2.16: Comparison of FPGA vs. ASIC

Field Programmable Gate Arrays can offer a valid solution to the ASICs limits, in fact they have all the advantages of programmability; besides lower design costs, thanks to the low time-to-market, they offer a further economic gain. The main drawback is the loss of efficiency, above all in terms of die area and speed.

Another possible solution to ASICs weak points could be a full-mask-set design methodology, based on *logic bricks*, an hybrid solution between standard cell ASICs and configurable arrays, proposed in [28]. As analyzed in [60], designing an IC using regular logic bricks permits to improve manufacturability, since regularity has been recognized to be a valid mean to limit the variability of circuit's parameters, that can cause malfunctioning, specially with the shrinking of ICs sizes.

2.4.1 Field Programmable Gate Array

Field Programmable Gate Arrays are programmable devices constituted essentially by a matrix of *programmable logic blocks (PLB)*, and *programmable interconnections* between them, connecting logic blocks between them and to *programmable IOs*. Over the years, several hardware cores have been added to the basic structure, such as microprocessors, DSPs (digital signal processors), blocks of RAM and FIFO (first-in first-out) memories or IP (intellectual property) cores (e.g. Ethernet mac core).

The principal characteristic of FPGAs is *programmability*. To allow programmable blocks to change their own behavior, a memory is used, the so-called *configuration memory*. Depending on the construction technology of the configuration memory, different types of FPGAs exist.

The principal classification of FPGAs is related to the programmability of the memory; in fact they can be divided in *volatile* and *non-volatile*, in *one-time programmable* and *erasable*, but they can also be *in-system programmable* or not.

Fuse technology uses bipolar transistors, while *antifuse* technology is CMOS-based; both are one-time programmable. *EPROMs* are erasable only with UV (ultra violet) light, while *EEPROMs* are electrically erasable, so depending from the package, they can also be in-system programmable. Better than the EEPROMs (with regard to programmability and package size), the *Flash* memories are erasable, but not all of them are in-system programmable. Finally *SRAM-based* FPGAs are in-system programmable and re-programmable, but they are non-volatile.

So, the *SRAM-based FPGAs* are the most versatile; for this reason, in the last years, the use of such devices has grown, also because of its higher density, more features and lower costs compared to non-volatile FPGAs.

One of the major problem with SRAM-based FPGAs is that, unlike the non-volatile ones, they lose configuration when powered off, so an external memory is required to store the configuration information.

The configuration is performed by loading a stream of bits, called *bitstream* in the configuration memory. After the boot of the device, an internal state machine configures all the programmable blocks, depending upon the information held by bitstream.

Figure 2.17 shows a schematic representation of the structure of an FPGA with a microprocessor and some hardware cores.

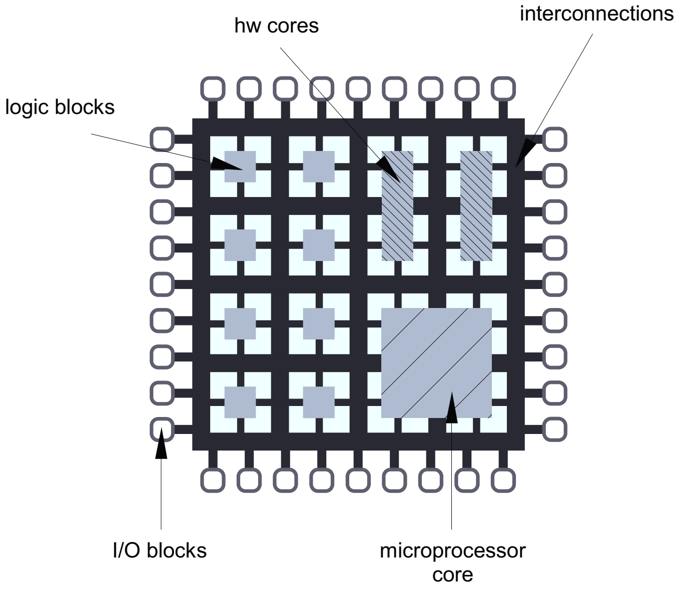


Figure 2.17: An FPGA schematic representation

FPGA Testing. In order to perform tests on FPGAs, the structure of such devices must be taken into account. In fact, as mentioned above, there are several kind of structures in a typical FPGA.

Testing of SRAM-based FPGAs has recently been a subject of several research studies. As stated in [30], there are two possible kinds of testing techniques for in-system reconfigurable FPGAs, *external testing techniques*, and those employing *built-in self test (BIST)* approaches.

Also, it is possible to identify testing techniques for FPGAs as *application independent testing* or *application dependent testing* [30]. The former stimulate all the FPGA parts with no regards to the task that the device will perform (i.e., the application), while the latter tests the specific configuration loaded by the user. So, application dependent testing is less general then application independent one, but it is considerably better with respect to delay faults, and in particular with path delay fault.

The advantage of application dependent testing results from the structure of the FPGAs interconnections. FPGA interconnections are *hierarchical*, that is: various matrix of interconnections exist, both global and locals, that permit to route the signals between different zones of the device. With the miniaturization trend in the manufacturing technology of FPGAs, the interconnection delay within a path can became the major contribution to the total path delay. Because it is very difficult to test all the possible interconnections during the application independent testing, doing the test for only the configured ones is more feasible.

2.4.2 Logic Bricks

One of the newest approach to increase yield of the VLSI circuits construction is based on the use of highly regular structures called *logic bricks*.

"A *brick* comprise a logic function created from a small set of logic primitives that are mapped onto a micro-regular fabric"[28].

The new design methodology studied in [28], is based on a library of macro components, that must be built following two main directives. The set of primitives must be:

1. flexible enough to realize the logic required
2. small enough to limit the geometric shape within the bricks.

Logic bricks have been conceived to improve manufacturability of the circuits [28], because they are constructed following some rules to obtain the maximum regularity, e.g. all wires on a given layer are unidirectional, have fixed pitch, and are wide enough to avoid notches and landing pads. Figure 2.18, from [28] shows an example of these constraints.

As explained in [58], bricks layout is created in two phases consisting of *transistor placement* and *routing*. *Transistor placement* is performed

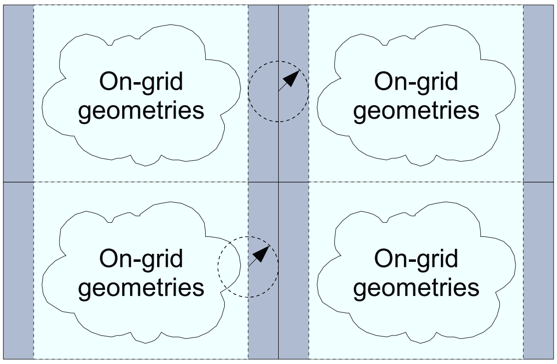


Figure 2.18: Regular Bricks. Micro-regularity constraints within bricks and compatible borders (shown in gray) provide macro-regularity compatible over radius of influence

using an extended branch-and-bound technique that minimizes wire length and cell area, both of which are estimated during brick creation. *Routing* for a logic brick is accomplished by transforming the problem to a SAT formulation.

In figure 2.19, from [28], an example brick is showed. As can be seen, the number of points that can be programmed (marked with a X) is very low, compared with the full-customizable FPGA basic block, shown in figure 2.20. In fact the latter use a n-bit *look up table (LUT)* that can accomplish every n-bit function; the drawback is related to the implementation of the LUT, that is unknown to designer and not optimized for regularity and area occupation.

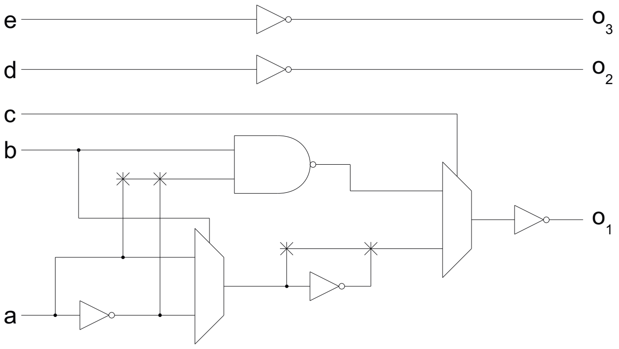


Figure 2.19: Example of a logic brick

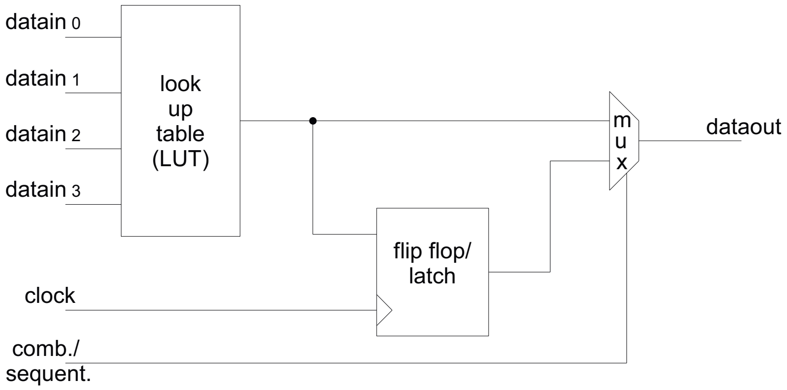


Figure 2.20: Schema of a FPGA basic block

Chapter 3

Aim of the Project

Delay fault testing and at-speed testing are widely used to verify the timing of synchronous digital IC's. The importance of these techniques is still growing because of the relevant IC's parameters uncertainties which characterize the current technologies [5, 6, 11].

In order to drive this process, several fault models and test generation techniques have been developed that target different trade-offs between accuracy and efficiency.

The largest fraction of these approaches is based upon gate level descriptions of the circuit [51]. In case the basic building blocks are more complex than logic gates and their implementation is not known, *functional level approaches* have been proposed [43, 64]. For instance, this is the case for *look-up tables based Field Programmable Gate Arrays (FPGAs)* and it may be a perspective for deep submicron circuits that exploit *logic bricks* as basic building blocks [28]. This class of circuits has been referred to as *macro* [43] or *module based* [64].

The *path delay fault model* [52] can be used in both gate and macro based combinational (full-scan) circuits to detect distributed defects resulting in timing violations. Path delay fault testing verifies the timing of paths by providing sensitization conditions ensuring that errors are produced when transitions are propagated through paths whose delays exceed their timing budget.

3.1 Overview

Robust testing [35] is the main paradigm ensuring the verification of a path's timing independently from the timing of other paths in the circuit. This goal is typically achieved by preventing from the propagation of *hazards* along the path under test.

In this regards, the specific case of module/macro based circuits, however, presents two main differences with respect to the gate level case.

The first one depends on the possibly complex internal structure of macros which, differently from simple CMOS gates, may contain different internal paths that let a transition on a specific input to propagate to a macro's output. For this reason, there are delay fault models that, if the macro implementation is not known, conservatively assume that every input configuration sensitizing a path from the macro's input to its output is characterized by an independent value of delay to be verified in the testing phase [43, 64].

The second difference regards the problem of hazard generation that, in macro based circuits, is more complex than in gate based ones. In gate level circuits, in fact, a hazard can be generated only because of skews between the arrival times of suitable transitions of at least two gate inputs (*function hazard*) [39]. In the case of macro based circuits, reconverging paths within the macro may generate an output hazard even if only a single input switches (*logic hazard*).

3.2 Review

In this section, a brief review of the works strictly related to the activity presented in this thesis and that partially stimulate the interest in such matters, will be given.

In [43], a functional delay fault model has been developed for combinational circuits containing macros. In such a work, *function robust path delay faults* have been defined to describe the conditions for the *robust* propagation of a transition through a macro. These conditions are computed by using a *signal representation* that relates a triple to each signal denoting:

1. the value to which the signal eventually stabilizes when applying the

first test vector;

2. the value during the transition;
3. the value to which the signal stabilizes after the application of the second test vector.

Robust conditions are then computed by analyzing the truth table of the macro. These conditions are computed for each macro along a path and the path delay fault is considered to be tested by a set of test pairs justifying all the possible combinations of robust conditions. This kind of approach will be thereafter referred to as *exhaustive path verification*.

In [64], a delay fault test generation method for modular circuits is presented. Such a method exploits the ease of boolean expressions manipulation that is provided by ordered *binary decision diagrams (BDDs)* [8]. In order to reduce the occurrence of logic hazards, the robust sensitization conditions used in this method are more restrictive than in [43]. Because of this, the robust conditions described in [64] do not scales down to gate level conditions when modules implementing simple elementary logic functions are considered. Also, this method considers exhaustive conditions for robust path delay fault testing. In addition, the authors provide a technique for the design of hazard-free logic modules ensuring that the proposed test generation method detects all *robustly testable* paths in gate level implementation of modules.

3.3 Motivation

The methods proposed in the literature mainly focus on robust tests which ensure high levels of test quality. However, when *robust tests* are not available, *non-robustly sensitizable* paths and even *functionally sensitizable* paths [51] may result in timing violations and should be verified by delay fault testing.

In this regards, during my PhD period, a new method for delay fault testing of circuits based on modules that allows to account for robust, non-robust and functional sensitization conditions was proposed. This approach can support any kind of delay fault model including the exhaustive conditions considered in [43] and [64]. However, the approach was specifically applied to the case of small combinational modules whose timing can be reasonably modeled using the *timing arc delay model* [17]. The use of such an approximate model is justified by the growing relevance of interconnect delays in submicron digital ICs [27] and by the relevant sources of timing defects that may affect such interconnects such as those due to breaks or opens [12, 3, 34].

Our method is based on a *signal representation* that allows to describe path delay fault testing constraints.

For each on-path macro, the proposed method uses *boolean differential calculus* supported by the BDD package described in [53] to compute different kinds of side input sensitization conditions. Robust, non-robust and functional sensitization conditions are expressed as a function of the macro's input signals.

The set of test pairs detecting the considered path delay fault can be computed by expressing the sensitization conditions of all the macros along the path as a function of the variables describing the values of PIs and by performing the conjunction of such functions [64]. This approach may be expensive for some class of circuits that cannot be efficiently described in terms of ordered BDDs [9].

To approach this problem, an alternate technique based on *boolean satisfiability* [1] was used. This kind of technique has been successfully applied to delay fault test generation both in *combinational* [13] and *sequential* [20] *ICs*.

In a first step, the proposed approach uses BDDs and boolean differences to compute the different kinds of sensitization conditions for each on-path macro as a function of macro's input signals. Then it translates these functions to *conjunctive normal forms (CNFs)*. The constraints on the side inputs of on-path macros are justified by using additional CNFs that describe the consistent operations of each macro in the circuit with the initialization and the launch test vector. An additional CNF describing the (function) hazard generation and propagation is also computed for each macro in the circuit.

Finally, the CNF that describes the whole set of test pairs detecting the considered path delay fault is computed as the conjunction of all the above mentioned CNFs. If existing, a test pair for the considered path delay fault can be found by invoking a *SAT solver* [40] on such a final CNF.

The SAT approach to test pairs generation becomes attractive when searching for only a subset of all the possible test vectors; in fact, for circuits that can be efficiently managed using BDDs, the methods completely based on BDDs [64] are expected to be more effective in evaluating all the possible test vectors than the proposed one.

The proposed approach is here instantiated by considering an example of fault model that well matches with the case of small combinational macros. Such a model has been suggested in [30] to the purpose of FPGA testing. It supposes that the propagation delay from an input to the output of a macro depends only on the kind (rise/fall) of input and output transitions. Therefore, the fault is considered to be tested when all the possible consistent combinations of rising and falling transition of on-path signals have been verified. Each of them will be referred to as a *pattern* of the considered path. The CNFs describing the kind of signal transitions for a specific pattern can be simply added to the CNF describing sensitization conditions and circuit's operations. If existing, a test pair detecting such a fault can be found by invoking any SAT solver (zChaff [40] is used in this instance).

Within the considered delay model, the proposed approach allows for a correct management of function hazards. In case of more complex circuits, the timing arc approximation becomes not realistic and the generation of internal logic hazards cannot be neglected.

Finally, it must be noted that the proposed *signal representation* allows to describe the problem of robust and non-robust test generation as pseudo boolean [1] optimization problem.

The feasibility of the proposed approach has been verified by applying it to a set of macro based combinational benchmarks that has been obtained by modifying the ISCAS'85 [7] benchmarks and the full-scan, synthesized version of some of the ITC'99 benchmarks [16].

Chapter 4

A new model for delay fault testing in macro based ICs

This work addresses single path delay faults in a combinational circuit or in an (enhanced) full-scan [18] one, that is composed of *functional macros* whose implementation is not known.

As stated before, when testing for a path delay fault, two test vectors $\langle u, v \rangle$ are applied to the circuit, allowing a transition to propagate through the faulty path; these vectors constitute a *test pair* and are called *initialization* and *launch* (or *propagation*) test vectors respectively. The *initialization vector u* is applied at a low test rate, thus ensuring that each line of the circuit has enough time to reach its steady state value. Then, the *propagation test vector v* is launched and the circuit outputs are captured using a fast test rate. The time elapsing between the launch of v and the capture of POs signals is the *clock period T*; for the sake of simplicity, timing parameters of flip-flops are not considered.

In order to detect a path delay fault (i.e. a path delay exceeding the timing constraints), the used test pair has to propagate a transition through the path under test. This kind of test may be invalidated by hazards, so, to prevent from this kind of problems, suitable conditions have been identified [51].

As explained in section 2, the robust testing paradigm ensures that a path delay fault, that makes the path's delay larger than T, is detected by a robust test independently of any other delay inside the circuit. This definition has been initially introduced in the case of gate level networks in [35], and in [43] it has been extended to circuits composed of functional macros.

The *macros* along the path implement functions $f : \mathbb{B}^n \rightarrow \mathbb{B}$ ($\mathbb{B} = \{0, 1\}$) and the dynamic behavior of the single block is modeled using input delays such as in the *timing arc delay model* [17]. Such a model is widely used in timing analysis, but, differently from timing analysis, these delays are not quantified in path delay fault model, that considers them as *unbounded*.

This timing model, illustrated in figure 4.1, is not well suited for complex combinational blocks, because it misses both the possibly different paths within a block and the possible paths' recombination between blocks, giving rise to the generation of logic hazards.

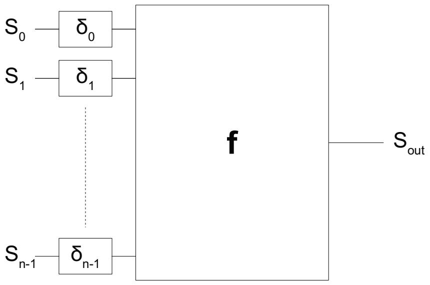


Figure 4.1: Simple timed model of a functional block (macro)

This work target simple combinational macros, such as logic bricks, that are expected to present up to 8-10 inputs and a low logic depth. In such modules, the generated logic hazards should present a reduced width, thus being easily filtered out by the logic in their transitive fan-out.

4.1 Signal Representation

In order to characterize the sensitization conditions of macro's side inputs, the following signal representation is used.

Let u_i and v_i be the final ($t \rightarrow \infty$) values of a signal s_i belonging to a path p when u and v are applied to the circuit, respectively.

In particular, let t_0 and $t_0 + T$ be the instants triggering the launch of v and the capture of the corresponding POs, respectively. Let also δ_i be the actual slack of s_i with respect to the path's end when the considered test pair is applied. Depending on the path's side inputs, the logic value sampled at path's end may be sensitive to the value w_i of s_i at $t_{s_i} = t_0 + T - \delta_i$ (this value will be referred to as the *sampled* value of s_i). A logic error may be sampled if $w_i \neq v_i$.

In order to characterize signals from the point of view of the sampled value (an unbounded delay model is used), a boolean variable (α_i) is related to each signal. In particular, $\alpha_i = 0$ denotes the case where $w_i = v_i$ while $\alpha_i = 1$ denotes the case where a delay fault may make $w_i \neq v_i$. Now, the case of hazards is not considered and therefore, when $\alpha_i = 1$, $w_i = u_i$. In a hazard-free and single path context, α_i characterizes the timing behavior of s_i with the current test pair.

Using α_i, the value of a signal s_i under the possible presence of a delay fault is characterized by:

$$w_i = (\alpha_i u_i + \alpha_i' v_i). \tag{4.1}$$

Note that α_i is not referred to a particular defect location, but to the timing of the whole path.

Considering b, a single output combinational block, described under the simplified *timing arc delay fault model* in figure 4.1 (this model will be refined in section 4.4), $S_b = (s_0, s_1, \ldots, s_{n-1})$ is the ordered set of the input signals of a block and $f_b \ : \ \mathbb{B}^n \rightarrow \mathbb{B}$ is the implemented function. If $U_b = (u_0, u_1, u_2, \ldots, u_{n-1})$ and $V_b = (v_0, v_1, v_2, \ldots, v_{n-1})$ are the ordered sets of variables denoting the values of signals belonging to S when the test vectors u and v are applied, respectively, then the correct value sampled at the block output signal (s_{out}) is $v_{out} = f_b(v_0, v_1, v_2, \ldots, v_{n-1})$.

Finally, let $\alpha_b = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ be the ordered set of the variables determining the value of w_i for each input of the block.

If the sampled values of the input signals of b are denoted as $W_b = (w_0, w_1, w_2, \ldots, w_{n-1})$, then, in the actual circuit, the following relationship holds: $w_{out} = f_b(w_0, w_1, w_2, \ldots, w_{n-1})$.

For instance, in case $f_b = v_0 v_1 + v_2$, sampled values are given by:

$$\begin{aligned} w_{out} &= (\alpha_0 u_0 + \alpha_0' v_0)(\alpha_1 u_1 + \alpha_1' v_1) + (\alpha_2 u_2 + \alpha_2' v_2) = \\ &= \alpha_0' \alpha_1' v_0 v_1 + \alpha_0' \alpha_1 v_0 u_1 + \alpha_0 \alpha_1' u_0 v_1 + \alpha_0 \alpha_1 u_0 u_1 + \alpha_2 u_2 + \alpha_2' v_2 \end{aligned}$$

It must be noted that the proposed signal representation is equivalent to the one proposed in [43]. The use of the variables α_i is here introduced to explicitly denote delay fault effects and to enable the use of boolean differential calculus.

4.2 Path Delay Fault testing sensitization conditions

In case a transition is propagated through a path containing the block b with output out, a logic error is sampled at the path's end if:

$$\sigma_b = w_{out} \oplus v_{out} = 1. \tag{4.2}$$

σ_b describes all the configurations of U_b, V_b and α_b that produce the sampling of a wrong logic value at the path's end in case the value of s_{out} is propagated through a sensitized path.

In the particular case of single path delay faults, the condition that $s_k \in S_b$ is the on-path input require that a path delay fault is on the path ($\alpha_k = 1$ and a logic error is sampled at path end ($\sigma_b = 1$), so the following relationship can be imposed:

$$\sigma_{b,k} = \sigma_b \alpha_k = 1.$$

The kind of transition propagated to s_k can be imposed as the conjunction of $\sigma_{b,k}$ with the the suitable values of u_k and v_k. For instance, a rising transition can be described as $u'_k v_k$, thus leading to the following condition:

$$\sigma_{b,k}^{rise} = \alpha_k u'_k v_k \sigma_b = 1, \tag{4.3}$$

that describes all the possible kind of sensitization conditions for the side inputs of the considered block (robust, non-robust and functional) in case of a rising transition of s_k. As it will be shown, the kind of sensitization conditions depends on the values to be assigned to the variables α of the off-paths to satisfy equation 4.3.

4.2.1 Robust Conditions

The possible values of the side inputs that ensure *robust test conditions* for the considered block need to be computed. This test paradigm is simply enforced by making fault detection independent of any delay in the circuit other than that of the tested path. In this case, it can be simply imposed that $\sigma_{b,k}$ must be independent of the values of α_j, $\forall j \neq k$. This condition ($\rho_{b,k}$) can be expressed in closed form using the *boolean differences* with respect to all the possible non-empty subsets of $\alpha_{b,k} = \alpha_b \backslash \{\alpha_k\}$. In particular, it is:

$$\rho_{b,k} = \Bigg(\sum_{\substack{j \neq k \\ j=0}}^{n-1} \frac{\partial \sigma_{b,k}}{\partial \alpha_j} + \sum_{\substack{i \neq k \\ i=0}}^{n-2} \sum_{\substack{j \neq k \\ j=i+1}}^{n-1} \frac{\partial^2 \sigma_{b,k}}{\partial \alpha_i \partial \alpha_j} + \sum_{\substack{i \neq k \\ i=0}}^{n-3} \sum_{\substack{j \neq k \\ j=i+1}}^{n-2} \sum_{\substack{l \neq k \\ l=j+1}}^{n-1} \frac{\partial^3 \sigma_{b,k}}{\partial \alpha_i \partial \alpha_j \partial \alpha_l} + \ldots$$

$$\ldots + \frac{\partial^n \sigma_{b,k}}{\partial \alpha_0 .. \partial \alpha_{k-1} \partial \alpha_{k+1} .. \partial \alpha_{n-1}} \Bigg)' \quad (4.4)$$

where $\partial f / \partial x$ is the boolean difference of f with respect to x ($\partial f / \partial x = f|_{x=0} \oplus f|_{x=1}$).

In the remainder of this work, the disjunction of all the boolean differences of the generic function g with respect to all non-empty subsets of boolean variables in the generic set A will be denoted to as $\mathcal{D}(g, A)$. Therefore, equation 4.4 can be rewritten as:

$$\rho_{b,k} = \mathcal{D}(\sigma_{b,k}, \alpha_{b,k})'. \quad (4.5)$$

The set of input configurations of the logic block must satisfy both the condition of error detection in case of path delay fault on the on-path ($\sigma_{b,k}$) and the robust propagation conditions (given by $\rho_{b,k}$). So the relationship holds:

$$\eta_{b,k} = \rho_{b,k} \sigma_{b,k} = 1. \quad (4.6)$$

$\eta_{b,k}$ includes all the possible robust sensitization conditions (that are independent of the variables in $\alpha_{b,k}$ set) for the on path input s_k. Note that these conditions allow for transitions on side inputs. This may increase the logic hazards probability, but it allows to explore cases related to multiple input transitions that may have an impact on the timing of CMOS ICs [14, 55].

It is worth mentioning that the closed form formulation of $\rho_{b,k}$ involves the computation of $2^{n-1} - 1$ coefficients. As an alternative, the dependencies on α_i can also be eliminated in an iterative or recursive way:

$$\eta^{i}_{b,k} = \begin{cases} \sigma_{b,k} & \text{if } i = -1 \\ \eta^{i-1}_{b,k} \left(\dfrac{\partial \eta^{i-1}_{b,k}}{\partial \alpha_i} \right)' & \text{if } 0 \leq i \leq n-1 \ \wedge \ i \neq k \end{cases} \tag{4.7}$$

requiring a linear number of computations. The final value of $\eta_{b,k}$ can be computed as:

$$\eta_{b,k} = \begin{cases} \eta^{n-1}_{b,k} & \text{if } k \neq n-1 \\ \eta^{n-2}_{b,k} & \text{if } k = n-1. \end{cases} \tag{4.8}$$

In the example introduced in section 4.1, if s_0 is the on-path input, conditions are:

$$\eta_{b,0} = u_0'.v_0.v_1.u_2'.v_2' + u_0.v_0'.u_1.v_1.v_2'.$$

In this expression, the two product terms correspond to the cases of rising and falling transitions of s_0, respectively. The corresponding pairs of input vectors are shown in table 4.1.

These assignments to the block's side inputs provide *robust conditions* for fault effect propagation, avoiding the possible generation of *functional hazards* at the output of the block. Conversely, they do not provide any protection with respect to *logic hazards* generated inside the block, because the method herein described does not consider blocks' internal implementation.

It must be noted that, in case f_b corresponds to the function of a logic gate, the presented method provides the traditional conditions for robust test generation.

4.2.2 Non-Robust Conditions

The problem of *non-robust tests* in circuits composed of combinational macros has not yet been addressed in details in the literature. However, in case no robust test exists for a given path, it may still give rise to timing errors and it should be tested. To this purpose, non-robust side input assignments that are not included in the robust set are also addressed in this work. So, it is required that:

a) at least one side input has a fault-free delay (thus violating robustness);

b) no side input should be required to be delayed (the violation of this condition would lead to functional sensitization).

For a given pair of macro's input values ($\langle u, v\rangle$), condition a) can be expressed as:

$$\exists\, j \neq k \ :\ \sigma_{b,k}(u,v)|_{\alpha_j=1} = 0$$

while condition b) is:

$$\neg\exists\, j \neq k \ :\ \sigma_{b,k}(u,v)|_{\alpha_j=0} = 0.$$

The conjunction of these conditions ($\mu_{b,k}$) can be expressed in a more compact way as:

$$\mu_{b,k} = \sigma_{b,k}\big|_{\prod_{j\neq k}\alpha'_j} \cdot \left(\sigma_{b,k}\big|_{\prod_{j\neq k}\alpha_j}\right)' = 1 \tag{4.9}$$

where $\sigma_{b,k}\big|_{\prod_{j\neq k}\alpha'_j}$ represent the evaluation of $\sigma_{b,k}$ when $\alpha_j = 0\ \forall j \neq k$ (i.e. the generalized cofactor of $\sigma_{b,k}$ with respect to $\prod_{j\neq k}\alpha'_j$) and $\sigma_{b,k}|_{\prod_{j\neq k}\alpha_j}$ is the evaluation of $\mu_{b,k}$ when $\alpha_j = 1\ \forall j \neq k$.

The set of input test pairs satisfying both the condition of error sampling (with error on the on-path) ($\sigma_{b,k}$) and non-robust minus robust condition ($\eta_{b,k}$) are given by:

$$\theta_{b,k} = \sigma_{b,k}\mu_{b,k} = 1. \tag{4.10}$$

So, $\theta_{b,k}$ groups those test pairs that do not satisfy robust constraints while they still ensure non-robust sensitization conditions.

Therefore, once robust test generation for a given path delay fault fails, non-robust test sequences may be generated.

In the example of section 4.1, it is:

$$\theta_{b,0} = u'_0v_0v_1u_2v'_2\alpha'_2 + u_0v'_0u'_1v_1u'_2v'_2\alpha'_1.$$

As can be seen, the first product term corresponds to a rising transition of u_0 and defines a set of configurations of U and V that detects the fault only if $\alpha_2 = 0$. The second product term describes a test set detecting the fault only if $\alpha_1 = 0$. The corresponding test vectors are shown in table 4.1.

The whole set of non-robust tests can be computed as:

$$\xi_{b,k} = \eta_{b,k} + \theta_{b,k} = 1. \tag{4.11}$$

As an alternative, $\xi_{b,k}$ can be directly computed by using the following formulation that, for each side input, imposes that either the test is independent of the considered side input timing or it requires that it has a

kind of sensitization	$\mathbf{u_0u_1u_2}$	$\mathbf{v_0v_1v_2}$	$\boldsymbol{\alpha_0\alpha_1\alpha_2}$
robust	0-0	110	1--
	11-	010	1--
non-robust minus robust	0-1	110	1-0
	10-	010	10-

Table 4.1: Robust and non-robust test conditions for the block implementing the function $ab + c$

fault-free timing:

$$\xi^i_{b,k} = \begin{cases} \sigma_{b,k} & \text{if } i = -1 \\ \xi^{i-1}_{b,k}\left(\alpha'_i + \left(\dfrac{\partial \xi^{i-1}_{b,k}}{\partial \alpha_i}\right)'\right) & \text{if } 0 \le i \le n-1 \ \wedge \ i \ne k \\ \xi^{i-1}_{b,k} & \text{if } i = k \end{cases} \tag{4.12}$$

requiring the computation of a linear number of coefficients. The values of $\xi_{b,k}$ can be computed as:

$$\xi_{b,k} = \begin{cases} \xi^{n-1}_{b,k} & \text{if } k \ne n-1 \\ \xi^{n-2}_{b,k} & \text{if } k = n-1 \ . \end{cases} \tag{4.13}$$

4.2.3 Quality of non-robust tests

In the case of non-robust tests, the number and the timing of the side inputs which do not satisfy robust constraints have a strong impact on the probability that a non-robust test is invalidated in spite of a path delay larger than the available slack [49, 15]. In particular, non-robust tests featuring the longest path segments which are robustly testable are preferable to others, as shown in [49]. Also, side inputs that do not satisfy robust test constraints should stabilize to their final value as soon as possible, as shown in [15].

Since the presented method makes use of the unbounded delay hypothesis, the only way to improve the *quality* of non-robust tests is to minimize the number of side inputs that do not satisfy robust constraints.

From this point of view, equation 4.11 does not pose any constraints on the quality of non-robust tests, which will depend on random choices performed by SAT solver. However, the problem can be approached by using a pseudo boolean formulation.

As known, *pseudo boolean solvers* [1] can minimize linear functions in the form $\sum_i a_i b_i$ where $a_i \in N$ and $b_i \in \{0, 1\}$, according to boolean constraints that are typically expressed as CNFs. This property can be exploited in order to minimize the number of side inputs that do not satisfy robust constraints.

In case a solution of equation 4.11 requires a fault-free transition of a side input s_i, the corresponding variable α_i is set to 0. Conversely, if this constraint is not present, α_i may be set to any value. Therefore, if $\mathcal{Q}$ is the set including all the side inputs of blocks along the path, the cost function to be minimized is:

$$\Psi = \sum_{\forall i \in \mathcal{Q}} (1 - \alpha_i)$$

where the sum is the arithmetic one.

In case the solver achieves $\Psi = 0$, a robust test is found, otherwise, the minimization process will reduce the number of side inputs that do not satisfy robust constraints, thus providing a high quality non-robust test. This formulation of the high quality path delay fault testing as a pseudo boolean problem may take advantage from any performance improvement of pseudo boolean solvers.

4.2.4 Functional sensitization

The test pairs that satisfy equation 4.2 (i.e. the fault sampling conditions) but does not satisfy non-robust conditions are referred to provide only *functional sensitization* conditions [51] and they can be can be computed as:

$$\nu_{b,k} = \sigma_{b,k}(\eta_{b,k} + \xi_{b,k})' \ = 1.$$

The test pairs defined by $\nu_{b,k}$ may result in an error only if the transition of one or more side inputs violate timing constraints, that is are able to discover only multiple faults.

4.3 Accounting for Hazards

4.3.1 Hazards on side inputs

In order to keep the description of the method as simple as possible, in the previous analysis it was not accounted for possible *hazards* on the side inputs of b.

Before to describe how the method accounts for hazards, a brief remark of the kind of hazards existing is required. A module output may present two kinds of *static hazards* [21]: function and logic. *Function hazards* depend on the timing of block's inputs and the block's logic function, while *logic hazards* are implementation dependent. Since the presented method is independent of the implementation of logic blocks, only function hazards will be considered.

The *signal model* described in section 4.1 can be modified in order to describe the possible presence of hazards that are characterized by the boolean variables h_i and χ_i. The first variable denotes the possible presence of a hazard (of course, all primary inputs have $h_i = 0$), while the second one describes the uncertainty on the value that is sampled in case the hazard is active when the sampling instant occurs. This latter event occurs only if a timing violation is in order, because, at the sampling instant, the signal has not yet stabilized to its final value. Therefore, the value of a signal s_i can be written as:

$$w_i = (\alpha_i' v_i + \alpha_i(h_i' u_i + h_i \chi_i)) \tag{4.14}$$

For the considered logic block, let $H_b = (h_0, h_1, \ldots, h_{n-1})$ and $\chi_b = (\chi_0, \chi_1, \ldots, \chi_{n-1})$, be the ordered sets of such variables. *Robust test conditions* can be expressed by imposing that, for all inputs but the on-path one, fault detection is independent of the value of α_j and, for all inputs, fault detection is independent of χ_j.

Therefore, equation 4.6 can be rewritten as:

$$\rho_{b,k} = \Bigg(\sum_{\substack{j\neq k\\ j=0}}^{n-1} \frac{\partial\sigma}{\partial\alpha_j} + \sum_{j=0}^{n-1} \frac{\partial\sigma}{\partial\chi_j} + \sum_{\substack{i\neq k\\ i=0}}^{n-2} \sum_{\substack{j\neq k\\ j=i+1}}^{n-1} \frac{\partial^2\sigma}{\partial\alpha_i\partial\alpha_j} + \\ + \sum_{\substack{i\neq k\\ i=0}}^{n-1} \sum_{j=0}^{n-1} \frac{\partial^2\sigma}{\partial\alpha_i\partial\chi_j} + \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{\partial^2\sigma}{\partial\chi_i\partial\chi_j} + \dots \\ \dots + \frac{\partial^{2n-1}\sigma}{\partial\alpha_0 \dots \partial\alpha_{k-1}\partial_{k+1} \dots \partial\alpha_{n-1}\partial\chi_0\partial\chi_1 \dots \partial\chi_{n-1}}\Bigg)' \quad (4.15)$$

that, in a more compact way, is:

$$\rho_{b,k} = \mathcal{D}(\sigma, \alpha_{b,k} \cup \chi_b)'.$$

Since the different effects of hazards are not of interest for the method, a relevant simplification can be made by considering $\chi_i = X \ \forall i$, where X is a boolean variable representing an unknown value. In this way, equation 4.15 becomes:

$$\rho_{b,k} = \Bigg(\sum_{\substack{j\neq k\\ j=0}}^{n-1} \frac{\partial\sigma}{\partial\alpha_j} + \frac{\partial\sigma}{\partial X} + \sum_{\substack{i\neq k\\ i=0}}^{n-2} \sum_{\substack{j\neq k\\ j=i+1}}^{n-1} \frac{\partial^2\sigma}{\partial\alpha_i\partial\alpha_j} + \sum_{\substack{i\neq k\\ i=0}}^{n-1} \frac{\partial^2\sigma}{\partial\alpha_i\partial X} + \dots \\ \dots + \frac{\partial^{n}\sigma}{\partial\alpha_0 \dots \partial\alpha_{k-1}\partial\alpha_{k+1} \dots \partial\alpha_{n-1}\partial X}\Bigg)' \quad (4.16)$$

or, in a more compact formulation:

$$\rho_{b,k} = \mathcal{D}(\sigma, \alpha_k \cup X)' \; .$$

Equation 4.6 can be used to calculate $\eta_{b,k}$ and also in this case an iterative or recursive computation can be used. Considering a simple example ($f_b = v_0 v_1$), where s_0 is supposed to be the on-path input, it is obtained:

$$\eta_{b,c} = u_0' u_1 v_1 h_0' + u_0 u_1' h_0' h_1' v_0 v_1$$

. The first product corresponds to the case of a rising transition. In order to satisfy it, the on-path transition (i.e. that on s_0) must be hazard free and the side input must simply be at high logic value with the second test vector. Note that the constraints on the on-path input signal are automatically satisfied by considering the logic feeding such a signal. The second product term, instead, corresponds to a falling transition. In this case, both the inputs should be hazard free, while the side input should remain stable at the high logic value. Again, these conditions correspond to the usual constraints used in the robust testing of an AND gate.

4.3.2 Evaluation of hazards within the circuit

As shown in the previous section, the robust test conditions may include constraints on the variables (h_i) denoting the possible presence of hazards on path's side inputs. Test generation should provide hazard-free transitions when required by equation 4.16, therefore, the circuit should be characterized from the point of view of hazards.

Within the *timing arcs delay model*, a hazard may be present at the output of a functional block because of:

a) the generation of a hazard caused by hazard-free transitions at the block's inputs;

b) the propagation of an input hazard.

Note again that (logic) hazards generated because of the internal timing of the considered block are neglected.

In the actual circuit, the hazards generated in case a) depend on the input vectors applied to the considered module c, the timing of the input signals of c and the delays that are related to such input signals. Since an unbounded delay model is used, the characterization of all the possible hazards occurring at the output of c is required.

To this purpose, the signal representation introduced in section 4.3 will be used, but in this case w_i does not represent the sampled value of s_i; it simply represents any value after u_i and before v_i.

Under such hypothesis, γ_c describes the possible presence of a static hazard at the output of c:

$$\gamma_c = (u_c \oplus w_c)(w_c \oplus v_c) = 1 \tag{4.17}$$

where, u_c, w_c and v_c can be computed as a function of the values characterizing the n inputs of c:

$$\begin{aligned} u_c &= f_c(u_0, u_1, \ldots, u_{n-1}), \\ w_c &= f_c(w_0, w_1, \ldots, w_{n-1}), \\ v_c &= f_c(v_0, v_1, \ldots, v_{n-1}). \end{aligned}$$

Note that the condition in equation 4.17 can be satisfied only if $u_c = v_c$.

It must be noted that, in this approach, *dynamic hazards* are not considered, because they would be relevant only if constraints on side inputs specify both u_i and v_i with $u_i \neq v_i$. This, however, contradicts robustness hypothesis because in such a case the value sampled at the output of the considered block would depend on the side input's timing. In any case, the proposed method can be easily extended to handle dynamic hazards.

In order to compute w_c, a block's input signal s_i is characterized by u_i, v_i and by:

$$w_i = (\alpha_i' v_i + \alpha_i(h_i' u_i + h_i X)),$$

thus accounting for both the generation of a function hazard at the output of c and the propagation of input hazards described by the variables $h_i \in H_c$.

In equation 4.17, γ_c' describes the possible conditions that, depending also on the macro's inputs timing as defined by the values of the variables in α_c, cannot give rise to the possible presence of hazards. In the context of robust testing, the input values that ensure a hazard-free output independently of their relative timing are required. Therefore, the difference operator ($\mathcal{D}$) is used again to provide the configurations of U_c, V_c and H_c which, independently of the actual ICs timing (a part from the timing internal to c), cannot give rise to the generation of a hazard. The characteristic function ($n_c(U_c, V_c, \alpha, X)$) of this set of configurations can be computed as:

$$n_c(U_c, V_c, \alpha, X) = \gamma_c' \; \mathcal{D}(\gamma_c', \alpha \cup \{X\})'. \tag{4.18}$$

Consistently with its definition, the function h_c= n_c' describes the input configuration that may produce an output hazard.

As an example, considering the function $f_c = v_0 v_1 + v_2$ again, it is:

$$\begin{aligned} n_{out} = {} & v_0 v_1 u_0 u_1 h_0' h_1' + v_0 v_1 u_0 u_2' h_0' h_1' h_2' + v_0' v_2' u_0' h_0' h_2' + \\ & + v_0' u_0' u_2' h_0' h_2' + v_1 u_0' u_2' h_0' h_1' h_2' + v_1' v_2' u_1' h_1' h_2' + v_1' u_1' u_2' h_1' h_2' + \\ & + v_2 u_2 h_2' + v_2' u_0 u_1 h_0' h_1' h_2' + u_0 u_1 u_2 h_0' h_1' h_2' + u_0' u_1' u_2' h_0' h_1' h_2', \end{aligned}$$

that denotes all the possible hazard-free conditions.

The use of the function n_c depends on the method used for delay test generation. In a context completely based on BDDs, n_c can be expressed for each c as a function of PIs and it can be substituted to the variable h'_c in the function describing the robust sensitization conditions for a path. This may be not possible for some circuits also because the BDD representing hazards may be more complex than that describing the circuit's logic function.

In this method, instead, BDDs are used to compute the functions n_c of each IC's macro. Then the (potential) hazard activity of the whole circuit is computed as a CNF by translating the relationship $h'_c = n_c(U, V, \alpha, X)$ of each macro to a CNF.

This representation of hazards' activity in the circuit will be used to justify the constraints imposed by robust conditions on the variables h_i of side inputs, assuming that PIs have $h_i = 0$.

4.4 Instance of fault model

The test generation technique described in the previous sections may support different kinds of delay fault models for macro based circuits. In this section, it will be instantiated by using a fault model [30] that is consistent with the timing arc delay model, illustrated in figure 4.2.

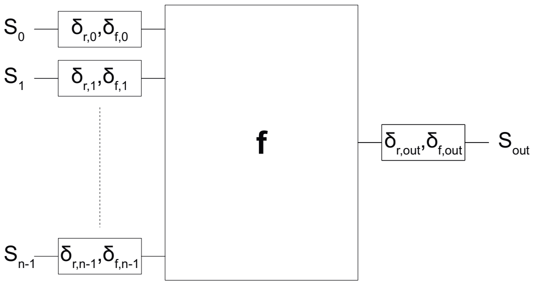


Figure 4.2: An example of timing arc delay model that accounts for the different kinds of transitions on the inputs and the output of a macro

It must be noted that the delay model in figure 4.1 was only used to simplify the discussion of the proposed approach. The model in figure 4.2 accounts for the kind (rise (r) or fall (f)) of transitions of the inputs and of the output of a macro. It is well suited to describe the delays of complex CMOS cells in a context where:

a) macros are not expected to feature possibly complex paths requiring the use of exhaustive test approaches [43, 64];

b) the delays of the interconnects between such cells are still expected to represent a relevant fraction of the actual path's delay.

As regards the values of delay parameters, in this work this model has been adjusted using the unbounded hypothesis.

In this context, the timing of a path can be verified by sensitizing it for all the possible combinations of rise and fall delays of its on-path signals. This fault model will be thereafter referred to as *pattern path fault model* where a pattern π_p of a path p containing m_p macros is here defined as one of all the possible combinations of rising (r) and falling (f) transitions $\{r, f\}^{m_p+1}$. Such a model is consistent with the constraints imposed by the logic functions of on-path macros.

In particular, the number of pattern delay faults related to a path p depends on the functions of the on-path macros. In fact, if the relationship between the on-path input and the output is unate for all the macros along the path, only two patterns are possible. On the contrary, if all these relationships are binate there are 2^{m_p+1} patterns. Note that this model requires a specific test pair for each pattern fault.

It must be also noted that the meaning of a pattern delay fault is well defined in the case of robust tests, while the same does not hold in the case of non-robust tests. In fact, if the side inputs of a macro does not satisfy robust conditions, the macro's output presents the same initial and final value. In such cases, the values r and f are referred to the direction of the last transition of the possible output hazard. If *out* is the macro output and $v_{out} = 0(1)$, then the transition will be denoted by $f(r)$.

As an example, figure 4.3 shows a small circuit composed of very simple combinational macros whose functions are shown in table 4.2. Bold lines denote a path instance $p = (s_0, s_7, s_{10}, s_{11})$ which involves the macros M_0, M_3 and M_4.

The module M_0 is non inverting, so that only the cases of a rising or a falling transition that propagate from s_0 to s_7 with no inversion has to be

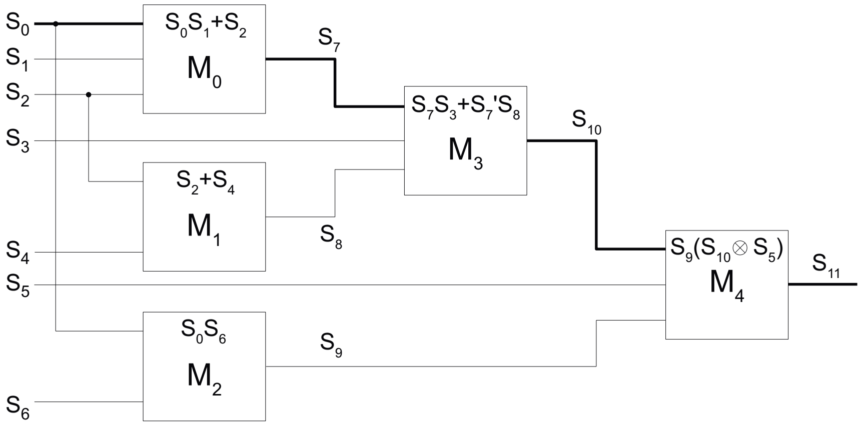


Figure 4.3: Example of macro based combinational circuit

macro	**equation**
M_0	$s_7 = s_0s_1 + s_2$
M_1	$s_8 = s_2 + s_4$
M_2	$s_9 = s_0s_6$
M_3	$s_{10} = s_3s_7 + s_8s_7'$
M_4	$s_{11} = s_{10}s_9$

Table 4.2: Equations of the macros in figure 4.3

considered. These two possibilities that are in order for the arc (s_0, s_7) will be denoted as $\langle rr\rangle$ and $\langle ff\rangle$, respectively. Conversely, M_3 and M_4, instead, let to propagate the on-path signal in both a non-inverting and an inverting way. Therefore, all the cases ($\langle rr\rangle$, $\langle rf\rangle$, $\langle fr\rangle$ and $\langle ff\rangle$) for both the arcs (s_7, s_{10}) and (s_{10}, s_{11}) have to be considered. By merging such information in a consistent way, the set of pattern delay faults illustrated in table 4.3 is obtained.

test no.	$\mathbf{s_0s_7s_{10}s_{11}}$
1	$\langle rrrf \rangle$
2	$\langle rrrr \rangle$
3	$\langle rrfr \rangle$
4	$\langle rrff \rangle$
5	$\langle ffrf \rangle$
6	$\langle ffrr \rangle$
7	$\langle fffr \rangle$
8	$\langle ffff \rangle$

Table 4.3: Pattern delay faults of the physical path p in figure 4.3

Chapter 5

Implementation of our approach

In order to verify the feasibility of the proposed test generation approach, an ATPG algorithm and a software tool that implement it were developed. The tool generate robust and non-robust test pairs for path delay faults. To do so, the algorithm uses *BDD* based *boolean differential calculus* to compute *sensitization conditions* for the on-path macros of the block-based circuit under test. Also, it uses *boolean satisfiability* to justify such conditions by means of suitable primary inputs' assignments.

Before to present the algorithm implementation in more details, a brief presentation of the instruments used to obtain it are given in the following sections.

5.1 Instruments

5.1.1 Boolean Differences

The concept of *boolean differences* was introduced by Shannon in 1938 [48], but it was formalized and used for the first time by Reed [45] and Muller [41] in 1954. Subsequently, many other works continued to deepen and extend such a concept, principally in order to solve the fault detection problem in logic circuits (in particular for stuck-at faults) [59].

In the following, the mathematical matters will not be discussed, because this is not the aim of this work. Instead, some of the characteristics of the boolean differences that permit us to use theme for the purpose of discovering robust (or non-robust) path delay faults will be presented.

Given a boolean function $F(X)$, where $X = (x_1, x_2, \ldots, x_i, \ldots, x_n)$ and x_i are boolean variables, the *boolean difference* of $F(X)$ with respect to a variable x_i can be indicated as $\partial F/\partial x_i$ and corresponds to:

$$F|_{x_i=0} \oplus F|_{x_i=1}$$

where $\oplus$ is the *exclusive OR* symbol and $F|_{x_i=1}$ and $F|_{x_i=0}$ are the positive and negative *cofactors* of $F(X)$ with respect to x_i. So the formula can be written as:

$$\frac{\partial F}{\partial x_i} = f(x_0, x_1, \ldots, 0, \ldots, x_{n-1}) \oplus f(x_0, x_1, \ldots, 1, \ldots, x_{n-1}). \tag{5.1}$$

As can be seen from previous formula, if the function $F(X)$ is independent of x_i, then the two cofactors are equals between them, so the result of the exclusive OR will be zero. Therefore boolean differences can express the *dependence* of a function respect to a variable (or more variables).

As discussed in section 2.1 the *robust testability* of a path delay fault is related to the independence of a delay fault on the on-path, respect to the delay faults on the side paths. As stated in section 4, in the present work the presence of a delay fault on a signal is represented in a symbolic way (by means of a boolean variable, called α). Therefore, the robustness conditions can be express in a functional way, applying the boolean differences method on the on-path macros' functions with respect to the α variables of the off-paths.

5.2 Binary Decision Diagrams

Binary Decision Diagrams are *rooted*, *directed*, *acyclic graphs*, that are able to represent boolean functions as a chain of *if-then-else* constructs. Such a data structure is based on the concept of *Shannon expansion* [48], that states, between others things, that a boolean function can be splited in two parts, fixing a value for a variable. Iterating such a transformation on the function, an if-then-else structure is obtained, called *Binary Decision Tree (BDT)*. A *BDD* is obtained by converting the BDT in a graph and removing some redundancy.

In a more pragmatic way, a BDD is a graph (directed and acyclic), with a single initial node, called *root*, with two terminal nodes, labeled as *logic zero* and *logic one*, and several internal nodes, representing the boolean variables of the function. Every node, except the terminal ones, have two arcs representing the possible variable assignment values (i.e. the logic zero and the logic one). Figure 5.1 shows a representation of the function $a*b+c$ by means of a BDD structure.

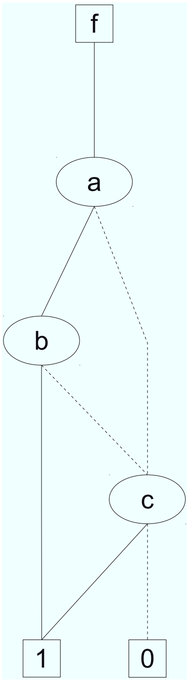


Figure 5.1: BDD example for the function $a * b + c$

The BDD structure can be changed to obtain a new BDD that describe the same function, but whose structure is optimized; often this is obtained by removing some redundancy. A BDD on which no further optimization can be carried out, is called to be *reduced*. Also, a BDD on which the variables on every path from root to leafs has the same order, is called to be *ordered*, also called *OBDD*. If a BDD is *reduced* and *ordered*, is called *ROBDD*. A *ROBDD* is a *canonical representation* of a boolean function, that is, the ROBDD representation of a function is unique, given an order of the variables. BDDs where initially introduced in [33] and [2], but in [8] the *OBDDs* and the *ROBDDs* were defined.

In order to represent and manipulate BDD data structures, several instruments are available. In the algorithm described later, the *CUDD (Colorado University Decision Diagram)* package [53] was used, so in the next section a brief presentation of this package will be given.

5.2.1 Colorado University Decision Diagram package

In order to use BDDs in the algorithm presented, *Colorado University Decision Diagram* package [53] was used. *CUDD* is a library of functions that permits to create and manipulate various type of decision diagram structures; it was developed by Fabio Somenzi of Colorado University.

CUDD package can be used as a black box, or by one of the interfaces available and written in various programming languages, or it can be used importing the library in the project [54]. This latter manner was used in this instance.

A significant characteristic of BDDs is the order of the variables, because a big gain in terms of dimension of the structure and/or efficiency during manipulation can be obtained, using a proper order of the variables. CUDD package permits to reorder the BDDs by means of a large set of dynamic reordering algorithms. Some of them are variations of existing techniques, while others have been developed specifically for the package [54]. Reordering can be obtained automatically every time a BDD reach a given number of node, or it can be forced by a call. In both the modality all the methods can be used. Also, it is possible to impose an arbitrary order of the variables, by a permutation of them [54].

Figure 5.2 shows a BDD representation of a simple function with no reordering of variables (in A) and after applying a simple reordering algorithm (B). As can be seen in figure 5.2, BDD in A has 28 nodes, while that one in B has been halved to 14 nodes, obtaining a big advantage in terms of dimension of the structure, hence in physical space allocation and probably in management costs. It must be noted that the result obtained in figure 5.2 B could not be the best possible, it was obtained applying one of the algorithms available from the CUDD package only with the purpose of show an example of the gain that can be obtained in terms of node numbers after a reordering operation on a BDD.

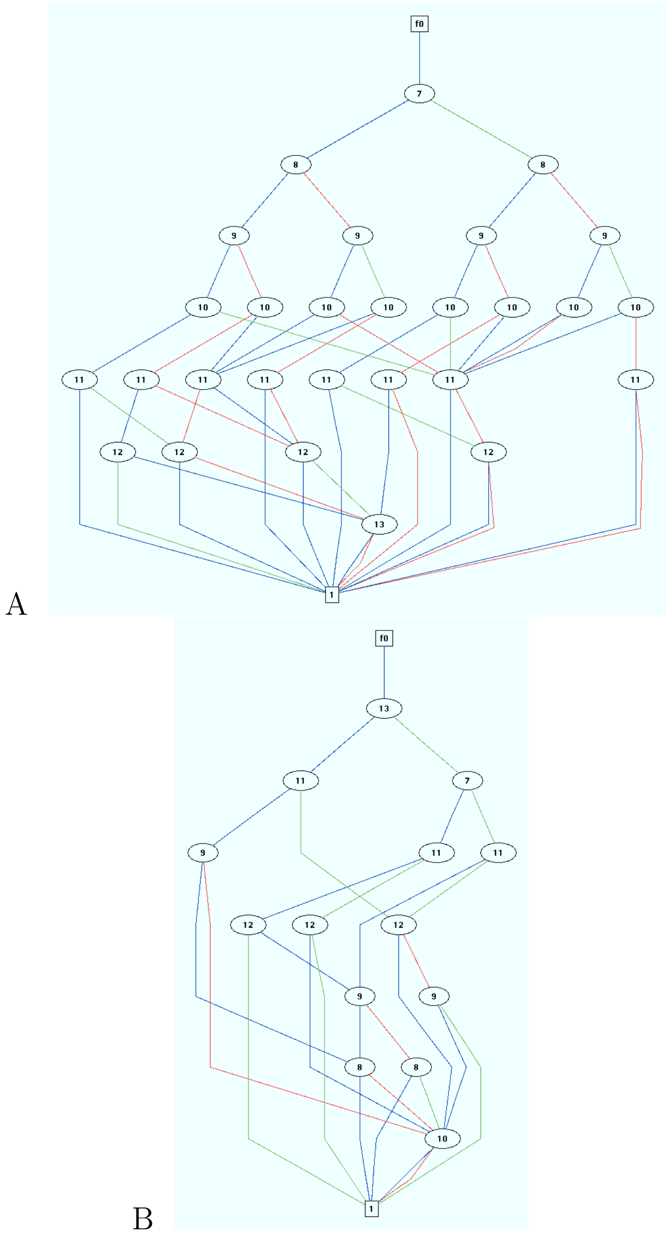


Figure 5.2: BDD example before (A) and after (B) reordering

5.3 Boolean Satisfiability

Given a boolean function, the *Boolean Satisfiability (SAT)* problem [1] permits to determine if exist a valid assignment for the variables, such that the function is *satisfiable*; if so, it is possible to obtain such values. If the assignments do not exist, the function is said to be *unsatisfiable*.

In order to use the satisfiability, a *SAT solver* is required; this is a software tool, that implements one of the many satisfiability algorithms available. Several solver algorithms and SAT solvers have been developed; between the most known, there are *GRASP* [37], *WalkSAT* [38], *MiniSat* [22] and *Chaff* [40]. The big effort in SAT solvers development is due [40] to the fact that SAT is widely used in some branch of the *Electronic Design Automation (EDA)*, such as *automatic test generation* [56] and *logic synthesis* [31], but also in *Artificial Intelligence (AI)*, for example in *automatic theorem proving*.

The most part of solvers employs combinations of the *Davis-Putnam (DP)* backtrack search strategy [19] and heuristic local searches techniques. When using heuristics, it is not guaranteed the *completeness* of the elaboration, that is, it is not guarantee that the solver will find the satisfiability assignments or the unsatisfiability [40].

In this work *zChaff*, an implementation of the *Chaff* algorithm [40] is used. Chaff is a *complete* solver, since it is based almost exclusively on the DP search algorithm [40].

Almost all the SAT solvers work on functions described in *Conjunctive Normal Form (CNF)*. Such a formulation consists of a conjunction of disjuncted boolean values. Every value is called *literal*, it is an instance of a function variable and it can be in asserted or negated form. A *clause* is the logical OR of literals, and finally the logical AND of all the clauses represents the function.

Conjunctive normal form is used to describe functions on which to apply satisfiability because every function can be expressed in CNF and every subset of clause can be tested for satisfiability independent from each others; a function is satisfiable if all the clauses are satisfiable themselves.

5.4 Test generation algorithm

The proposed test generation approach uses BDD based boolean differential calculus to compute sensitization conditions for on-path macros, while it uses boolean satisfiability to justify such conditions by means of suitable PIs' assignments.

The proposed algorithm reads a *BLIF (Berkeley Logic Interchange For-*

mat) [57] description of the combinational circuit and, for each macro b in the circuit, it uses a BDD package called *CUDD (Colorado University Decision Diagram)* [53] to perform the following operations:

- it maps the sets U_b, V_b, $H_b \cup \{X\}$ and α_b that characterize the macro's inputs to a set of independent BDD variables;
- accordingly to the macro's function f_b and to equation 4.18, it computes the BDDs corresponding to the functions $f_b(U_b)$, $f_b(V_b)$ and $n_b(U_b, V_b, H_b)$ that characterize the output signal of the macro.

Once this operation has been performed, the CNF describing the consistent operations of the macro b is computed by:

a) adding the variables u_b, v_b and h_b to the BDD system;

b) computing the BDDs corresponding to the following expressions: $u_b = f_b(U_b)$, $v_b = f_b(V_b)$ and $h_b = n'_b(U_b, V_b, H_b)$;

c) mapping the (local) BDD variable indexes to global CNF signal indexes;

d) translating the BDDs computed in b) to a CNF.

By repeating this operation for each macro of the circuit, a *conjunctive normal form (CNF)* is achieved, describing the consistent operations of the whole circuit when a test pair is applied. Such a CNF is stored for further use.

Then, a path list is generated that contains all or a fraction of the circuit's physical paths. In this regards, it must be noted that the selection of target paths is a critical step in delay test generation [49], but this matter is beyond the scope of this work. For each path, all the consistent pattern delay faults are generated.

Once computed the CNF characterizing the whole circuit and the fault list, the proposed algorithm starts to process target faults (in this case, pattern delay faults).

In the *robust* case, for a given pattern delay fault, the algorithm first builds the BDD describing w_i for each output signal s_i along the path. Then, it applies differential operators to obtaining the function $\sigma_{b,k}$ (where k is the on-path input of the considered macro) that must be also consistent with the kind of transition imposed by the target pattern delay fault on s_i. Then, it iterates as indicated in equation 4.7 to compute the BDD corresponding to $\eta_{b,k}$ which, in the robust case, is not a function of variables in $\alpha_{b,k}$.

Finally, the BDDs corresponding to the sensitization conditions of on-path macros are translated to CNFs, joined to the CNF describing the whole circuit and passed to the SAT solver.

In the *non-robust* case, $\xi_{b,k}$ is computed for each on-path input, while the kind of transitions imposed by the pattern delay fault in this case regards only the macros' outputs with the test vector v, thus letting possible hazards to propagate along the path.

5.4.1 Implementation

In order to verify the effectiveness of the proposed approach and algorithm, a software tool was developed. As said, such a tool works as ATPG for path delay faults in circuits composed by functional macros, but it is scalable in both the directions; in fact it is also possible to generate test pairs for both total functional circuits (i.e. constituted by a single block) and gate level described circuits.

Initially, the tool reads the circuit, described in *BLIF* format [57], so it is possible to know the topological characteristics of the circuit, such as the number of the blocks and the number and the structure of the paths (the paths are noted in a text file). Using these information, a set of CNF clauses is created, describing the conditions for the propagation of the signals throughout the entire circuit.

For every path, all the blocks are analyzed, so, for every block of the path, a *BDD* describing its function is generated. Boolean differences are applied on such BDDs, obtaining the *robust* (or *non-robust*) *sensitization conditions*. Starting from the BDD representation of the conditions, a *CNF* is generated; the CNFs related to every block of the path are joined together (in conjunction) to obtain a single CNF. Also, the global propagation conditions are added to the CNF and, when searching for robust tests, additional conditions to avoid hazards are added.

The zChaff SAT solver operates on the total CNF and returns all the assignments for the PIs' variables that justify the CNF function. Such values are the test pairs for the path under test of the circuit.

The software tool is able to change its behavior according to several options available; a list of the most important features will be briefly explained below. Above all, it is possible to generate *robust* or *non-robust* test pairs, also it is possible to generate *all the tests* discoverable or only *one test pair for every pattern* of the path. Moreover, it is possible to search the test pairs in every path of the circuit or only in *specific paths*, given in a text file. It is also possible to work on a percentage of the entire set of the paths, chosen randomly from the entire set.

There are also several ways to customize some of the internal details of the tool; principally, it is possible to customize the test generation algorithm, also it is possible to choose the results desired (only statistics, also the test vectors value, also the graphical representation of BDDs...) and some of the formatting of the results. Finally it is possible to change some of the technical details for BDD management, such as the garbage collection or extraction of values from BDDs as cube or prime.

With the default options, the tool generates two statistics files as results. The first one collects all the principal information divided in four classes:

1. statistics of call
2. circuit statistics
3. test vector generation statistics

   (these are the actual results)
4. computational statistics.

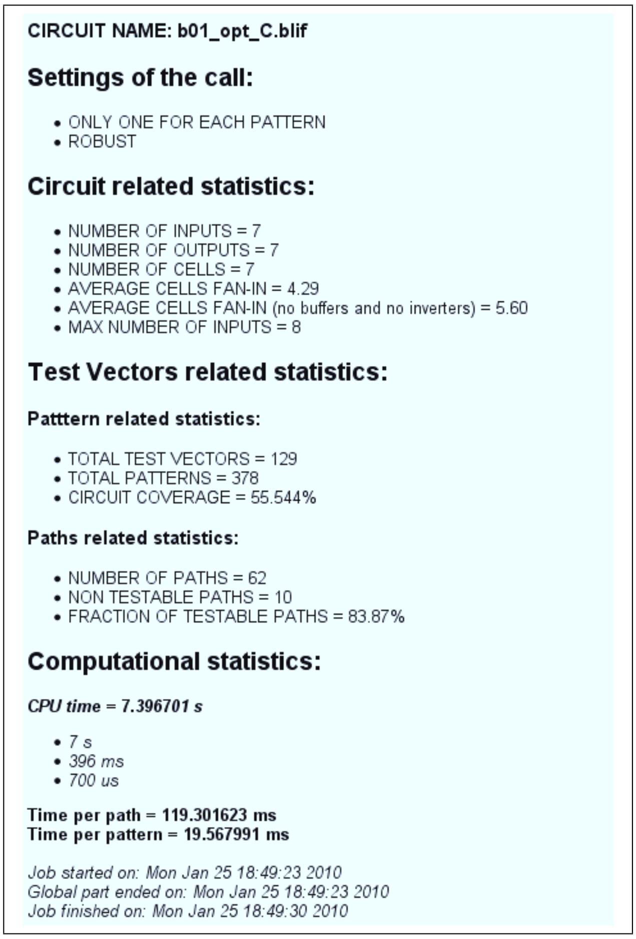
Figure 5.4 shows an example of statistical results obtained by the research of robust test pairs for *b01*, a simple full-scan circuit from ITC'99 benchmarks [16], whose blif representation is shown in figure 5.3.

The other statistics result is a XY file, whose points give the percentage of paths whose coverage is less than X; it permits to construct a *probability density function* of the coverage. Figure 5.5 shows a graph of the probability density function for robust test pair generated for *b01* circuit.

```
.model b01_opt_C.blif
.inputs a b c d e f g
.outputs h i j k l m n
.names a b d e f j
110-- 1
1-10- 1
-110- 1
--101 1
--010 1
.names a b d e f j m k
-11-1-- 1
-0-01-- 1
---101- 1
--10-0- 1
---01-1 1
1----00 1
.names a b d f j k m n l
-11--1-- 1
--1-10-- 1
--0-00-- 1
-1---10- 1
1--1--00 1
...
```

```
...
.names a b d e f m
010-- 1
100-- 1
1110- 1
0010- 1
111-1 1
001-1 1
01-10 1
10-10 1
.names d e f n
011 1
.names g h
1 1
.names c i
1 1
.end
```

Figure 5.3: Blif description of b01 full-scan macro-based circuit



CIRCUIT NAME: b01_opt_C.blif

Settings of the call:

- ONLY ONE FOR EACH PATTERN
- ROBUST

Circuit related statistics:

- NUMBER OF INPUTS = 7
- NUMBER OF OUTPUTS = 7
- NUMBER OF CELLS = 7
- AVERAGE CELLS FAN-IN = 4.29
- AVERAGE CELLS FAN-IN (no buffers and no inverters) = 5.60
- MAX NUMBER OF INPUTS = 8

Test Vectors related statistics:

Patttern related statistics:

- TOTAL TEST VECTORS = 129
- TOTAL PATTERNS = 378
- CIRCUIT COVERAGE = 55.544%

Paths related statistics:

- NUMBER OF PATHS = 62
- NON TESTABLE PATHS = 10
- FRACTION OF TESTABLE PATHS = 83.87%

Computational statistics:

CPU time = 7.396701 s

- *7 s*
- *396 ms*
- *700 us*

Time per path = 119.301623 ms
Time per pattern = 19.567991 ms

Job started on: Mon Jan 25 18:49:23 2010
Global part ended on: Mon Jan 25 18:49:23 2010
Job finished on: Mon Jan 25 18:49:30 2010

Figure 5.4: Example of the TPG output: statistical results of robust generation for *b01* circuit

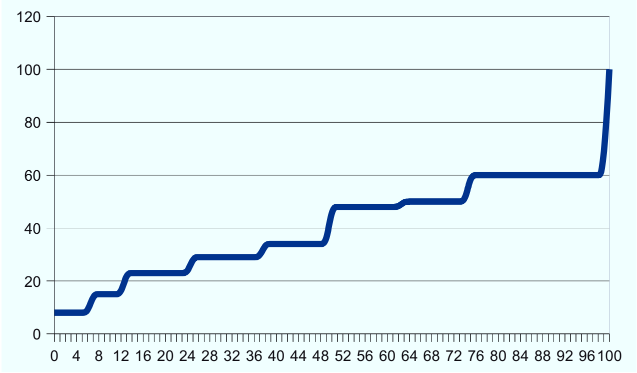


Figure 5.5: Probability density function of the robust coverage for *b01* circuit

Chapter 6

Experimental Measurements

In order to assess the feasibility of the proposed approach, a set of combinational benchmarks was considered. The set of benchmarks is composed of some circuits from the ITC'99 [16] (full-scan version) and ISCAS'85 combinatorial sets [7].

Since all the considered benchmarks are originally described as gate level netlists, some gate level subnetwork was collapsed in more complex macros featuring a number of inputs in the range $4 - 10$, using the `reduce_depth` command of *SIS* [47]. Then, local (i.e. without considering global don't cares) minimization was performed, using the command `simplify`.

In this regards, it must be noted that these circuits are not representative of optimized macro-based circuits. Conversely, they are still useful to the purpose because:

a) they are more complex than optimized circuits;

b) they feature several different macros (while for instance circuits based on bricks use a small number of macros);

c) local optimization may result in macros' input configurations which are not controllable or not observable thus providing overheads to test generation.

6.1 Results

The characteristics of the considered benchmarks are shown in table 6.1 providing the number of PIs, POs, cells and the average fan-in of the circuit's macros. In this regards, note that benchmarks still contain several simple gates (including inverters) that decrease this quantity. In fact, the average fan-in over all benchmarks is 3.7, but becomes 4.5 when not considering buffers and inverters. The last two columns of table 6.1 show the number of physical paths and the number of pattern faults for each benchmark, respectively. Note that in the case of the benchmark c6288, only a sample of paths was considered, because of the huge number of resulting pattern delay faults.

bench	**PIs**	**POs**	**cells**	**avg.**	**paths**	**patterns**
name	**no.**	**no.**	**no.**	**fan-in**	**no.**	**no.**
b01	7	7	7	4.29	62	378
b02	5	5	5	3.80	42	134
b03	35	34	43	3.70	990	3236
b04	36	7	96	3.85	9671	57042
b05	35	70	175	4.02	183451	2439776
b06	11	15	16	2.69	148	418
b07	50	57	184	3.91	15875	41846
b08	30	25	37	4.57	571	1936
b09	29	29	52	5.08	767	2366
b10	28	23	55	4.29	714	1762
c432	36	7	96	2.89	71950	143900
c499	41	32	74	4.43	7648	113280
c880	60	26	142	3.96	4708	16094
c1355	41	32	120	3.07	9440	139904
c1908	33	25	186	3.01	38751	502396
c2670	233	140	408	2.46	22466	117376
c3540	50	22	447	2.97	655228	15995316
c5315	178	123	307	3.21	33343	331168
c6288	32	32	484	4.80	10000	1393742
c7552	207	108	1030	3.02	68616	561222

Table 6.1: Characteristics of the considered benchmarks set

Table 6.2 and table 6.3 show test generation results in the robust and non-robust cases, respectively. In particular, table 6.2 shows the percentage of robustly detected pattern delay faults (C_r), the percentage of paths that generate at least a robustly testable pattern delay fault (P_r), and the number of generated test pairs (with no kind of compaction).

bench **name**	$\mathbf{C_r}$ (%)	$\mathbf{P_r}$ (%)	**test pairs** **no.**	**CPU** **time**
b01	55.54	83.87	129	10.4
b02	56.25	78.57	65	3.0
b03	66.36	87.17	1854	39.7
b04	28.93	45.35	11128	224.0
b05	3.34	9.03	38478	76.6
b06	47.12	58.78	181	3.7
b07	22.88	30.26	9192	67.9
b08	40.52	50.26	617	497.4
b09	72.22	77.44	1676	21.0
b10	50.75	64.29	861	29.0
c432	8.02	13.97	11551	82.6
c499	48.53	51.46	54848	42.2
c880	88.96	89.51	14170	49.7
c1355	39.33	41.39	54848	40.4
c1908	31.54	36.26	127386	26.3
c2670	24.21	24.91	16914	64.1
c3540	2.97	9.22	215604	26.5
c5315	51.83	67.23	132256	159.0
c6288	19.88	41.24	215864	217.0
c7552	39.51	62.68	134742	184.0

Table 6.2: Results for robust test generation

Table 6.3 shows the percentage of non-robustly detected pattern delay faults (C_{nr}), the percentage of paths that generate at least a non-robustly testable pattern delay fault (P_{nr}), and the number of generated test pairs (with no kind of compaction). As can be seen, similarly to the gate level case, some benchmark present very low levels of robust coverage.

The last two columns of table 6.2 and table 6.3 show the average CPU time (per pattern) in the robust and in the non-robust cases, respectively. The data have been collected by using a Intel Celeron CPU working at 2 GHz. The computational costs have been shown to be dominated by the time spent by the SAT solver to prove that a pattern delay fault is untestable. This can be seen by comparing the CPU time used for robust and non-robust test generation for the benchmarks C432 and C880. Although the benchmark C880 is larger than the C432, the average CPU

bench name	**C_{nr} (%)**	**P_{nr} (%)**	**test pairs no.**	**CPU time**
b01	60.68	91.94	147	19.9
b02	57.44	80.95	66	2.9
b03	67.66	88.38	1882	52.5
b04	34.85	57.04	13899	188.3
b05	4.89	12.50	55257	31.2
b06	48.98	62.16	189	3.3
b07	25.21	32.94	10318	54.6
b08	42.51	55.17	656	34.6
b09	72.75	77.44	1692	10.2
b10	52.32	67.93	889	19.8
c432	8.74	15.41	12588	87.2
c499	48.75	51.46	54912	216.2
c880	89.05	89.51	14182	40.0
c1355	46.10	54.92	139904	175.1
c1908	33.15	36.78	136327	20.0
c2670	28.02	37.54	23523	24.8
c3540	5.39	19.43	655228	57.9
c5315	57.19	75.58	152893	37.1
c6288	26.95	49.32	298741	194.0
c7552	44.29	64.77	146031	59.0

Table 6.3: Results for non-robust test generation

time per pattern fault is smaller. This because the C432 bench has several untestable delay faults.

In this regards, it must be noted that, with respect to the data in tables 6.2 and 6.3, consistent savings can be achieved because several paths exist (48.55% in average) that do not contain any non-robustly testable pattern. This occurs because it is impossible to generate a test vector v that satisfies static sensitization conditions. Therefore, if such paths contain several patterns it is convenient to first attempt to generate a test vector satisfying static sensitization conditions. If this operation fails, no robut or non-robust tests can be generated.

Chapter 7

Conclusions and future works

In this thesis the main work carried out in my PhD period was described. The aim of such an activity was the development of an ATPG methodology for delay faults in macro based circuits whose implementation is not known.

The result of the work was a framework to be used in delay fault test generation, that target robust as well non-robust test conditions for different kinds of path delay faults.

The proposed algorithm can deal with circuits that pose problems to BDD only based methods; this is obtained using BDDs at macro level, and satisfiability at circuit level. In addition, the used signal representation can also support methods that improve the quality of non-robust tests. Results show the feasibility of the proposed approach for a set of combinational circuits.

Many improvement may be done both to the algorithm and to the framework that implements it. In results shown in the section 6.1, it can be seen that, for several circuits, CPU time value is high. It must be emphasized that computations are performed starting from scratch for each path, and this imply that sensitization conditions may be computed several times for the same side input of a block.

In this regards, considering the simple circuit in figure 7.1, the paths $S_0 - S_7 - S_{10} - S_{11}$ and $S_4 - S_8 - S_{10} - S_{11}$ share M_3 and M_4 macros. In particular, conditions for sensitization from S_{10} input to S_{11} output of macro M_4 are the same when considering both the paths. The software developed, in this case, compute every time the same conditions, giving an overhead in terms of CPU time. This choice was done in order to keep the single path computation separated from each other, for the purpose to use any method (independent of the ATPG algorithm) to choose the paths on which to apply it.

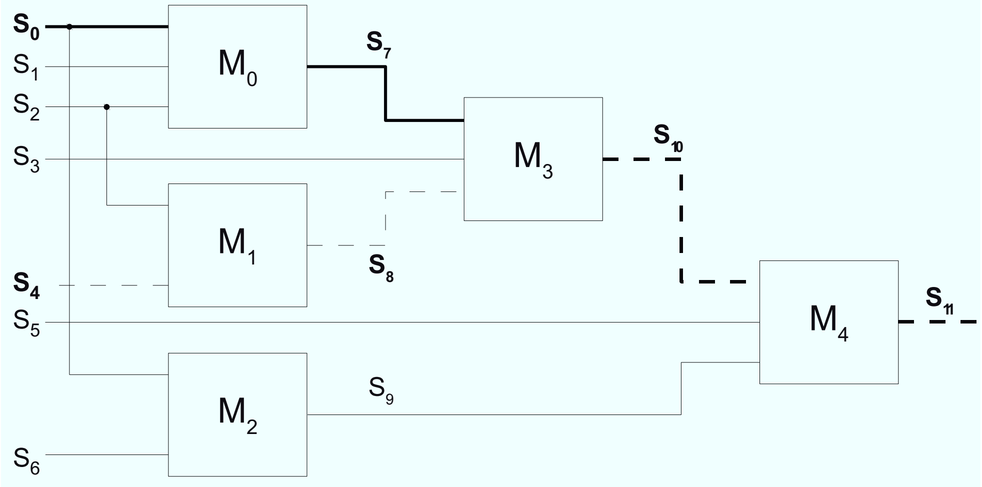


Figure 7.1: Paths with shared blocks

To avoid this problem, improvements are possible in the algorithm. In fact, basing the fault search not on paths, but on blocks or group of adjacent blocks, the duplicated computations of conditions may be avoided. In this manner, it would be possible also to exploit others delay fault models, such as *segment delay fault*, proposed in [26] and [49]. Such a model is a restriction of the path delay fault one, but it considers segments (i.e. subpaths) instead of paths; from PI to segment start and from segment end to PO, no constraints are considered.

It could be also possible to enhance non-robust tests quality, by minimizing the number of constraints that make a fault non-robust, as explained in section 4.2. In fact, using a pseudo boolean approach to minimize the number of constraints violating robustness, it can be possible to obtain robust tests if the minimized value is zero, and non-robust tests else. Such conditions may be considered high quality non-robust tests, because the number of violating constraints is the minimum possible.

A possible further work may be the evaluation of different kind of synthesized circuits. As the approach considers circuits composed of macros whose implementation is not known, it would be interesting to evaluate different implementations of circuits in order to compare the resulting values, in terms of robust and non-robust coverage and performance of the algorithm. This can be obtained both synthesizing circuits using different kind of standard library, and using a custom approach (as done in this work), but giving different parameters to synthesizer (i.e. block size, maximum and minimum number of inputs, maximum logical depth…). In this way, a tuning of the circuits' behaviors could be obtained.

Finally, the software may be enhanced by adding the fault simulation feature. In fact, as said, the software permits to generate all test pairs for the paths under test. In this manner it was seen that, for every testable pattern, there are a lot of test pairs, so it is possible that a single test pair may reveal more than a path delay fault. Therefore, it could be possible to obtain a test vector compaction by means of fault simulation. This could be possible because, in the proposed algorithm, the off-path inputs are not forced to remain stable, as in the approach proposed in [64], where only one path at a time can be sensitized.

Bibliography

[1] H. Van Maaren A. Biere, M. Heule and T. Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009. [cited at p. 30, 31, 41, 55]

[2] S. B. Akers. Binary decision diagrams. *IEEE_J_C*, (6):509–516, Jun 1978. [cited at p. 53]

[3] K. Baker and et. al. Defect-based delay testing of resistive vias-contacts. pages 467–476, 1999. [cited at p. 30]

[4] R.D. Blanton and J.P. Hayes. Properties of the input pattern fault model. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 372. IEEE Computer Society, 1997. [cited at p. 82]

[5] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. Design Automation Conference*, pages 338–342, 2–6, Jun 2003. [cited at p. 27]

[6] K. A. Bowman, S. G. Duvall, and J. D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE_J_JSSC*, 37(2):183–190, Feb 2002. [cited at p. 27]

[7] F Brglez and H Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *ISCAS85: IEEE International Symposium on Circuits and Systems*, 1985. [cited at p. 31, 63]

[8] R. E. Bryant. Graph - based algorithms for boolean function manipulation. *IEEE_J_C*, 35(8):677–691, Aug 1986. [cited at p. 29, 53]

[9] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE_J_C*, 40(2):205–213, Feb 1991. [cited at p. 30]

[10] M Bushnell and V Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI circuits.* Kluwer Academic Publishers, 2000. [cited at p. 5]

[11] Yu Cao and L. T. Clark. Mapping statistical process variations toward circuit performance variability: an analytical modeling approach. In *Proc. 42nd Design Automation Conference*, pages 658–663, 13–17, Jun 2005. [cited at p. 27]

[12] S. Chakravarty. On the capability to detect delay tests to detect bridges and opens. In *Asian Test Symposium*, pages 314–319, 1997. [cited at p. 30]

[13] Chih-Ang Chen and S. K. Gupta. A satisfiability-based test generator for path delay faults in combinational circuits. In *Proc. rd Design Automation*, pages 209–214, 3–7, Jun 1996. [cited at p. 30]

[14] Liang-Chi Chen, S. K. Gupta, and M. A. Breuer. A new gate delay model for simultaneous switching and its applications. In *Proc. Design Automation Conference*, pages 289–294, 2001. [cited at p. 37]

[15] Kwang-Ting Cheng, A. Krstic, and Hsi-Chuan Chen. Generation of high quality tests for robustly untestable path delay faults. 45(12):1379–1392, Dec 1996. [cited at p. 40]

[16] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc 99 benchmarks and first atpg results. *IEEE Design & Test of Computers*, 17(3):44–53, Jul–Sept 2000. [cited at p. 31, 59, 63, 89]

[17] A. J. Daga, L. Mize, S. Sripada, C. Wolff, and Qiuyang Wu. Automated timing model generation. In *Proc. 39th Design Automation Conference*, pages 146–151, 10–14, Jun 2002. [cited at p. 30, 34]

[18] S. DasGupta, R.G. Walther, and T.W. Williams. An enhancement of LSSD and some application of LSSD in reliability, availability and serviceability. In *Proc. of Int. Fault Tolerant Comp. Symp.*, pages 32 – 34, 1981. [cited at p. 33]

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. [cited at p. 56]

[20] S. Eggersgluss, G. Fey, and R. Drechsler. Sat-based atpg for path delay faults in sequential circuits. In *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2007*, pages 3671–3674, 27–30, May 2007. [cited at p. 30]

[21] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. In *Proc. Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 111–120, 11–13, Nov 1964. [cited at p. 42]

[22] Niklas En and Niklas Srensson. An extensible sat-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. [cited at p. 56, 84]

[23] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE_J_C*, C–32(12):1137–1144, Dec. 1983. [cited at p. 10]

[24] Dimitris Gizopoulos, Antonis M. Paschalis, Yervant Zorian, and Mihalis Psarakis. An effective bist scheme for arithmetic logic units. In *Proceedings of the IEEE International Test Conference*, pages 868–877, Washington, DC, USA, 1997. IEEE Computer Society. [cited at p. 82]

[25] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE_J_C*, C–30(3):215–222, March 1981. [cited at p. 10]

[26] K. Heragu, J. H. Patel, and V. D. Agrawal. Segment delay faults: a new fault model. In *VTS '96: Proceedings of the 14th IEEE VLSI Test Symposium*, page 32. IEEE Computer Society, 1996. [cited at p. 68]

[27] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *IEEE_J_PROC*, 89(4):490–504, Apr 2001. [cited at p. 30]

[28] V. Kheterpal, V. Rovner, T. G. Hersan, D. Motiani, Y. Takegawa, A. J. Strojwas, and L. Pileggi. Design methodology for ic manufacturability based on regular logic-bricks. In *Proc. 42nd Design Automation Conference*, pages 353–358, 13–17, Jun 2005. [cited at p. 21, 23, 24, 27, 81]

[29] Hyungwon Kim and John P. Hayes. Realization-independent atpg for designs with unimplemented blocks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(2):290–306, 2001. [cited at p. 82, 83]

[30] A. Krasniewski. Testing fpga delay faults in the system environment is very different from ordinary delay fault testing. In *Proc. Seventh International On-Line Testing Workshop*, pages 37–40, 9–11, Jul 2001. [cited at p. 22, 23, 31, 46]

[31] Wolfgang Kunz and Dominik Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, 1997. [cited at p. 56]

[32] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Proc. Meeting the Tests of Time. International Test Conference*, pages 795–801, 29–31, Aug. 1989. [cited at p. 10]

[33] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959. [cited at p. 53]

[34] J.C.M. Li, Tseng Chao-Wen, and E.J. McCluskey. Testing for resistive opens and stuck opens. pages 1049–1058, 2001. [cited at p. 30]

[35] Chin Jen Lin and S. M. Reddy. On delay fault testing in logic circuits. *IEEE_J_CAD*, 6(5):694–703, Sept 1987. [cited at p. 16, 28, 34]

[36] Shun-Yen Lu, Ming-Ting Hsieh, and Jing-Jia Liou. An efficient sat-based path delay fault atpg with an unified sensitization model. In *Proc. IEEE International Test Conference ITC 2007*, pages 1–7, 21–26, Oct 2007. [cited at p. 18]

[37] J. P. Marques-Silva and K. A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE_J_C*, 48(5):506–521, May 1999. [cited at p. 56]

[38] David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. In *AAAI/IAAI*, pages 321–326, 1997. [cited at p. 56]

[39] M. K. Michael and S. Tragoudas. Generation of hazard identification functions. In *Proc. Fourth International Symposium on Quality Electronic Design*, pages 419–424, 24–26, Mar 2003. [cited at p. 28]

[40] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proc. Design Automation Conference*, pages 530–535, 2001. [cited at p. 30, 31, 56]

[41] D E Muller. Application of boolean algebra to switching circuit design and to error detection. *IRE Transactions on Electronic Computation*, (3):6–12, 1954. [cited at p. 51]

[42] L. Pileggi, H. Schmit, A. J. Strojwas, P. Gopalakrishnan, V. Kheterpal, A. Koorapaty, C. Patel, V. Rovner, and K. Y. Tong. Exploring regular fabrics to optimize the performance-cost trade-off. In *Proc. Design Automation Conference*, pages 782–787, 2–6, Jun 2003. [cited at p. 20]

[43] I. Pomeranz and S. M. Reddy. On testing delay faults in macro-based combinational circuits. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 332–339, 6–10, Nov 1994. [cited at p. 27, 28, 29, 34, 36, 47]

[44] S M Reddy, C J Lin, and S Patil. An automatic test pattern generator for the detection of path delay faults. In *Proceedings IEEE/ACM International Conference on Computer-Aided Design*, pages 284–287, 1987. [cited at p. 17]

[45] I. S. Reed. A class of multiple-error-correcting codes and the decoding scheme. *IRE Professional Group on Information Theory*, 4(4):38–49, Sep 1954. [cited at p. 51]

[46] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10(4):278–291, July 1966. [cited at p. 10]

[47] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and Sangiovanni A. Vincentelli. Sis: A system for sequential circuit synthesis. Technical report, University of California, Berkeley, 1992. [cited at p. 63, 90]

[48] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, Dec 1938. [cited at p. 51, 52]

[49] M. Sharma and J. H. Patel. Testing of critical paths for delay faults. In *Proc. International Test Conference*, pages 634–641, 30–1, Oct–Nov 2001. [cited at p. 40, 57, 68]

[50] Ozgur Sinanoglu and Alex Orailoglu. Rt-level fault simulation based on symbolic propagation. In *VTS '01: Proceedings of the 19th IEEE VLSI Test Symposium*, page 240, 2001. [cited at p. 84]

[51] Mukund Sivaraman and Andrzej J. Strojwas. *A Unified Approach for Timing Verification and Delay Fault Testing.* Kluwer Academic Publishers, Norwell, MA, USA, 1998. [cited at p. 16, 17, 27, 29, 33, 41]

[52] Gordon L. Smith. Model for delay faults based upon paths. In *Digest of Papers - International Test Conference*, pages 342–349, 1985. [cited at p. 12, 13, 27]

[53] Fabio Somenzi. Cudd: Cu decision diagram package release 2.2.0, 1998. [cited at p. 30, 54, 57]

[54] Fabio Somenzi. http://vlsi.colorado.edu/ fabio/cudd/node3.html, 2009. [cited at p. 54]

[55] J. Sridharan and T. Chen. Gate delay modeling with multiple input switching for static (statistical) timing analysis. In *Proc. th International Conference on VLSI Design Held jointly with 5th International Conference on Embedded Systems and Design*, page 6pp., 3–7, Jan 2006. [cited at p. 37]

[56] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. 15(9):1167–1176, Sep 1996. [cited at p. 56]

[57] Berkeley Logic Synthesis and Verification Group. Berkeley logic interchange format. Technical report. [cited at p. 57, 58]

[58] B. Taylor and L. Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *Proc. 44th ACMIEEE Design Automation Conference DAC '07*, pages 344–349, 4–8, Jun 2007. [cited at p. 23]

[59] Andr Thayse. *Boolean calculus of differences*. Springer-Verlag (Berlin, Heidelberg, New York), 1981. [cited at p. 51]

[60] Kim Yaw Tong, V. Rovner, L. T. Pileggi, and V. Kheterpal. Design methodology of regular logic bricks for robust integrated circuits. In *Proc. International Conference on Computer Design ICCD 2006*, pages 162–167, 1–4, Oct 2007. [cited at p. 21, 81]

[61] J. A. Waicukauski, E. Lindbloom, B. Rosen, and V. Iyengar. Transition fault simulation by parallel pattern single fault propagation. In *in Proceedings of IEEE International Test Conference*, pages 542–549, 1986. [cited at p. 13]

[62] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. [cited at p. 6, 14, 15]

[63] Sitaran Yadavalli and Sudhakar M. Reddy. Symsim: symbolic fault simulation of data-flow data-path designs at the register-transfer level. In *ITC*, pages 606–615, 1999. [cited at p. 84]

[64] Joonhwan Yi and J. P. Hayes. High-level delay test generation for modular circuits. *IEEE_J_CAD*, 25(3):576–590, Mar 2006. [cited at p. 27, 28, 29, 30, 31, 47, 69]

[65] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel. A hybrid asic and fpga architecture. In *Proc. IEEEACM International Conference on Computer Aided Design ICCAD 2002*, pages 187–194, 10–14, Nov 2002. [cited at p. 19]

Appendices

Appendix A

High quality testing for circuits composed of small combinational macros

A.1 Introduction

Another significant work, that I deal with during my PhD period, targets specifically one of the kind of macro based circuits, those that use *logic bricks* as basic building block [28]. In particular, a *test generation and fault simulation method* for this kind of circuits has been developed.

As explained in section 2.4, *logic bricks* exploit a regular layout structure, that mitigate the lithography reliability problems that may affect standard CMOS libraries when the feature size is scaled toward nanometer technologies [28].

For a given application, in [60] a methodology is introduced identifying a set of basic logic functions whose layout can be conveniently fitted into a small number of geometrically optimized regular structures. The transistor level structure of a logic brick may include either pass-transistor logic (PTL) [28] or conventional CMOS gates. Moreover, the proposed logic bricks are supposed to be configurable by exploiting suitable vias [28].

In order to satisfy the test quality requirements of nanoscale CMOS circuits, it should be noted that the extension of switch and gate level fault models to logic bricks may be not immediate because the tools mapping the logic brick's function on the physical fabrics may hide some implementation details to the designer.

In this regard, *test generation for logic modules* whose realization is not known is a widely addressed topic in test automation [29]. The reason for this is twofold: first, these tests can be used before of the availability of a gate level circuit implementation; second, realization independent tests can be used for circuits whose realization is not known because of intellectual property (IP) restrictions. These techniques mainly target circuits implemented using standard cell or full-custom CMOS design styles. Their typical main target is to generate test sets providing a high stuck-at fault coverage on the widest possible range of gate level implementations. Different fault models, such as the bridging one, have been considered as a possible target for realization independent test generation, as in [24].

In this regards, consider a functional block b implementing a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$. Any combinational fault model for the module b can be described as $F \subset G$, where G is the set of the possible functions $g : \{0,1\}^n \rightarrow \{0,1\}^m$ and F represents the set of the possible faulty functions that can be related to b because of the presence of a physical defect. In [4], this model has been referenced as *multiple input pattern* fault model and it has been used to abstract physical defects to the behavioral level.

In the most general *cell fault model* [24], it is $F = G \setminus f$ and, as a consequence, all the possible input vectors should be applied to the block under test and, for each of them, any possible configuration of output values should be exposed to POs. In order to reduce the intrinsic complexity of the cell fault model, several fault models featuring $F \subset G \setminus f$ have been developed. These operations are typically based on relevant properties of f.

When considering a multi-output combinational module, most of the techniques [4] consider a single module output at a time. Hayes and Kim [29], instead, consider the problems related to multiple errors at module outputs when the modules are embedded inside a combinational circuit. In particular, the possible presence of multiple errors at the module's outputs is accounted by using a test generation heuristic that first attempts to propagate the error on a target output in a way that is independent of other module outputs and then employs an 11-values algebra to account for the possible reconvergence of module's output signals and the possible masking of erroneous values.

The case of logic bricks is rather different from that of large combinational macros because their very compact structure and the use of nanoscale technologies are very likely to make defects to produce multiple errors at their outputs. These may be due to defects affecting sub-circuits which are shared by more than one module output, or to faults such as bridgings that connect sub-circuits independent each other under fault-free conditions.

The lack of knowledge about the internal structure of logic bricks can be approached by the different test generation approaches, that account for functional modules whose structure is unknown [29], while the problems related to the presence of multiple errors at the brick's output has not yet addressed in details.

To this purpose, a *fault simulation and test generation method* was studied and developed. It target an exhaustive verification (cell model) of the module functionality by accounting in an *exact* way for the problems related to the physical defects that may give rise to multiple errors. In particular, such a method attempts to verify that, for each inputs' configuration of module under test, the brick's outputs have the correct value. Of course, this will be possible only under the controllability and observability constraints imposed by the surrounding logic.

The choice of using an *exhaustive verification* not only depends on the possibly unknown structure of the logic brick, but also on the increased variability of circuit parameters and on noise. Even if the transistor level structure of the module and the defect location are known, a circuit level analysis of the faulty circuit may be only partially meaningful. In fact, the actual behavior of a faulty module will depend on its parameters which may vary across the die and from die-to-die. In addition, the noise that the module and its neighborhood are experiencing during the application of a test may result in different behaviors of the same faulty circuit.

Anyway, the proposed method can be easily extended to handle different fault models where only a fraction of the input and output configurations are considered.

This method uses a *fault simulation technique* that, for a given module and input vector, in a *single step* accounts for all the possible defect induced behaviors (i.e. the set of all the possible faulty configurations) of the module outputs. To this purpose, each module output is considered as an independent boolean variable and the possibly different fault effects are propagated in a symbolic way: each signal in the transitive fan-out of the considered module outputs is computed as a function of such variables. In this way, the circuit POs are computed as a function of the module outputs and the behaviors of such signals that may be detected can be easily computed by comparing such values with the POs' fault-free values.

Symbolic fault effects propagation can be performed in different ways, for instance a BDD based approach to account for the uncertainties in bridging faults propagation can be used, as in []. In this work, instead, the limitations on the number of outputs which are expected in a logic brick have been exploited, to propose an efficient technique which prevents from the complexity related to the use of *Binary Decision Diagrams (BDDs)*. Symbolic techniques have also been applied to the fault simulation of RTL designs [63, 50].

In particular, any function of the module outputs is represented by means of its truth table as it is encoded by the bits of a word of the simulation host. With such a choice, any boolean operator of the host machine can be applied to such a kind of operands, thus computing the related composed function.

Note that, when using a 64-bit architecture, a maximum number of 6 outputs modules can be computed (of course this limitation can be easily extended, but simulation can no longer be performed in a single step). In addition, the proposed approach to fault simulation can be easily extended to HDLs.

The test generation method, instead, uses a *boolean satisfiability (SAT) solver* [22] to compute test vectors for target behaviors (i.e. a module input configuration and a set of wrong output values). The test generation process is repeated until a module has been exhaustively tested.

A.2 Fault modeling and simulation

The *cell fault model* supposes that a defect affecting the block under test (c) changes its function f_c in an unpredictable way, thus requiring an exhaustive testing of the module that should be verified for each possible input configuration and for each possible output configuration. Fault simulation

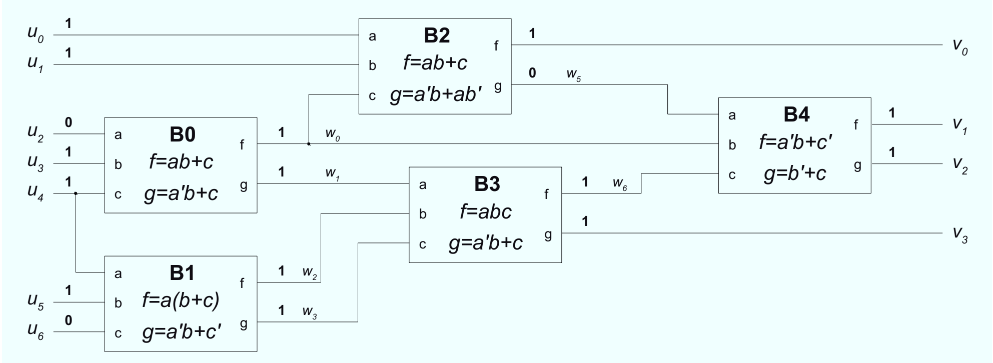


Figure A.1: Fault simulation step example

should efficiently account for this kind of behavior.

Example 1. *As an example, consider the combinational circuit illustrated in figure A.1. The figure shows also the fault-free signals' values when the input vector*

$$(u_0, u_1, u_2, u_3, u_4, u_5, u_6) = 1101110$$

is applied to such a circuit.

In case the block under test is $\boldsymbol{b}_0$*, its input configuration is* $(abc) = 011$ *and the fault-free values of modules' outputs are* $(fg) = 11 \Rightarrow (w_0w_1) = 11$*. In the considered circuit, the detection of a defect affecting* b_0 *depends on the faulty value of* w_0w_1*:*

- *if* $w_0w_1 = 01$*, then an error propagates to the PO* v_1*, thus resulting in fault detection;*
- *if* $w_0w_1 = 10$*, then an error propagates to the PO* v_2*, thus resulting in fault detection;*
- *if* $w_0w_1 = 00$*, then the fault is not detected.*

Therefore, the input configuration $(abc) = 011$ *is still not completely verified by the considered test vector. An additional test vector would be required to expose defects resulting in* $(w_0w_1) = 00$ *when such a local input configuration is applied. This is possible by applying the test vector* 1101111 *that results in an error at the output* v_3

The *exhaustive verification* of module's functionality is possible only if the inputs of the module are *freely controllable* and its outputs are *freely* *observable.* This property, however, may be not verified because of observability and controllability don't cares.

From the point of view of fault simulation, the example 1 suggests that, for the current test vector, several simulation runs may be required to analyze the effects of defects affecting module outputs in an unknown way.

To account in a single step for all the possible behaviors of the module's outputs, a *symbolic approach* was used, that relates a boolean variable to each output of the considered module and, for each signal in the transitive fan-out of the module's outputs, computes a function of such variables.

A.2.1 Fault simulation algorithm

In order to describe the proposed *fault simulation algorithm*, some definitions for a combinational circuit composed of multi-output functional modules must be introduced:

- let $U = \{u_0, u_1, ..., u_{n-1}\}$ be the set of circuit PIs;
- let $V = \{v_0, v_1, ..., v_{m-1}\}$ be the set of POs;
- let $W = \{w_0, w_1, ..., w_k\}$ be the set of internal signals;
- let $S = U \cup V \cup W$ be the set of all circuit signals.

Let also n_b and m_b be the number of inputs and outputs of a module b implementing the function $f_b : \{0,1\}^{n_b} \rightarrow \{0,1\}^{m_b}$, respectively.

When a test vector t belonging to a sequence T is applied to the circuit, $\varphi(s_i, t) \in \{0,1\}$ is the fault-free logic value of the signal $s_i \in S$.

In the proposed method, a boolean variable $\alpha_k \in A = \{\alpha_0, \alpha_1, ..., \alpha_{m_c-1}\}$ is related to each output of the block under test c, and a boolean function $\sigma(s_i, t) : \{0,1\}^{m_c} \rightarrow \{0,1\}$ defined on the support A is related to each signal s_i belonging to the transitive fan-out of the outputs of c.

First, fault-free simulation computes $\varphi(s_i, t)$ for each signal in the circuit. Then, if the signal s_j corresponds to the k-th output of c, symbolic simulation initializes $\sigma(s_j, t)$ to α_k. These values are propagated throughout the circuit by using the modules' functions, to compute the expression of each signal in the transitive fan-out of c as a function of the variables belonging to A.

Once symbolic propagation terminates, the POs of the circuit are a function of the variables $\alpha_k \in A$, and the configurations of these variables that can be detected at the p-th PO w_p are given by:

$$\varphi(w_p, t) \oplus \sigma(w_p, t) = 1. \tag{A.1}$$

Finally, the configurations of the variables in A (i.e. the output values of c) that are detected by t are those satisfying:

$$\delta(c, t) = \bigvee_{p=0}^{p=m-1} \sigma(w_p, t) = 1 \tag{A.2}$$

Note that $\delta(c, t)$ is related to the current input configuration $\xi(c, t)$ of c, and it will be denoted as $\delta(c, \xi(c, t))$.

At the end of simulation, the output configurations of c that can be detected by the whole test sequence T when the input configuration ξ_0 is applied to c are given by:

$$\Delta(c, \xi_0) = \bigvee_{\forall t \in T \,|\, \xi(c,t)=\xi_0} \delta(c, \xi(c, t)) = 1. \tag{A.3}$$

Note that the maximum size of the on-set of $\Delta(c, \xi_0)$ is $2^m_c - 1$, this because the configuration of the variables in A corresponding to the fault-free output of c ($f_c(\xi_0)$) cannot belong to $\Delta(c, \xi_0)$.

Example 2. *Consider again the example 1. In such an example, let α_0 and α_1 be the boolean variables related to the outputs w_0 and w_1 of the block under test. As illustrated in figure A.2, as a result of symbolic fault effects propagation, the functions σ related to signals in the fan-out of $\boldsymbol{b}_0$ are:*

$$\begin{array}{ll} \sigma(w_0) = \alpha_0 & \sigma(w_6) = \alpha_1 \\ \sigma(w_1) = \alpha_1 & \sigma(v_3) = 1 \\ \sigma(v_0) = 1 & \sigma(v_1) = \alpha_0 + \alpha_1' \\ \sigma(w_5) = 0 & \sigma(v_2) = \alpha_0' + \alpha_1 \end{array}$$

Therefore, $\delta(v_2) = 1 \oplus \alpha_0' + \alpha_1 = \alpha_0\alpha_1'$. Since $\varphi(w_0) = \varphi(w_1) = 1$, the fault is detected if an error is present on w_0, but not on w_1.

As regards the implementation of symbolic simulation, possibly different alternatives are possible. For instance, in [] symbolic simulation was

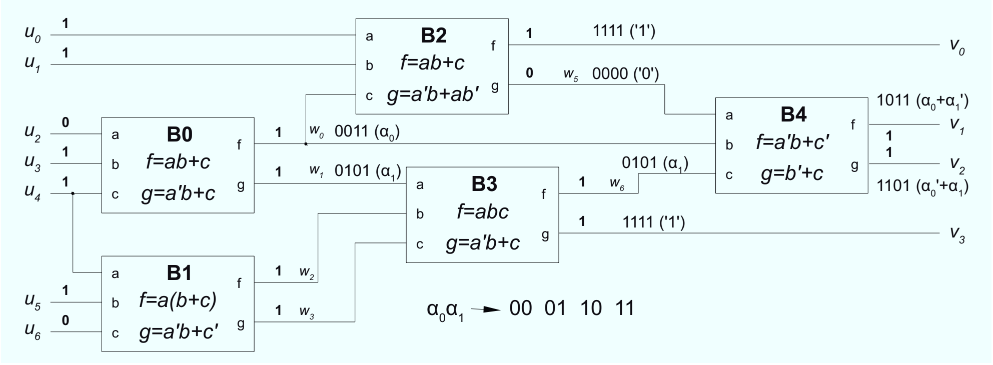


Figure A.2: Fault simulation algorithm example

performed using Binary Decision Diagrams (BDDs). Here, the properties of logic bricks that are expected to feature some limitation on the number of module's outputs are exploited, to propose an alternate technique that avoids some of the implementation complexity related to the use of BDDs.

In this approach, the functions σ describing the possible behaviors in the presence of a fault are described by their truth tables which are mapped on the words of the simulations' host. In the practice, the i-th bit of a host word holds the value of the i-th row of the truth table of σ (for a given configuration $(\hat{\alpha}_0, \hat{\alpha}_1, ..., \hat{\alpha}_{m_c-1})$, it is $i = \sum_{j=0}^{m_c-1} \hat{\alpha}_j 2^j$).

In case blocks' functions are represented using boolean expressions, the function σ related to the output of a logic block can be computed using bitwise logic operators of high level languages (AND, OR, NOT). As a less efficient alternative, one can use the logic vectors data types and operators which are commonly available in HDLs.

A.2.2 Coverage metrics

For a given macro c featuring n_c inputs and m_c outputs, one can define a *fault coverage* as the fraction of possible pairs of input fault-free configurations (ξ) and faulty output configurations that are detected:

$$C_c = \frac{1}{2^{n_c}} \sum_{\forall \xi} \frac{|\Delta(c,\xi)|}{2^{m_c} - 1} \tag{A.4}$$

where, $|\Delta(c,\xi)|$ is the size of the ON set of $\Delta(c,\xi)$.

In order to account for possibly large differences in the complexity of the used macros, the fault coverage of the whole circuit is expressed as a sum:

$$C = \frac{\sum_{\forall c} w(c) C_c}{\sum_{\forall c} w(c)} \tag{A.5}$$

where the coverage of each cell is weighted by a coefficient $w(c)$ accounting for the functional complexity of c that is expected to be proportional to its hardware complexity. In this work, the literal count of f_c is used.

A.3 Test generation

In the test generation procedure, a pseudo random test sequence is applied to the circuit until it does not provide coverage increments for a given number of test vectors. Then a deterministic approach, based on Boolean Satisfiability (SAT), is used.

In particular, such a method selects an undetected behavior (i.e. a pair of input configuration and output value) and try to generate a test pattern that generate such a behavior. The mechanism is better explained in example 3.

Example 3. *Figure A.3 shows a schema of the deterministic test generation step. In this situation, a test vector for the* $i-th$ *module is searched, when the local input vector* 01001 *is applied and the output configurations* 101 *and* 110 *have been detected during fault simulation. A test vector applied at PIs provides a fault coverage increment if the result of* $COND_B$ *is true, that is if the three following conditions hold:*

i. M_i *outputs values are not a behavior already detected;*

ii. the behavior generated by test vector 01001 *is not the fault free one;*

iii. vector 01001 *feeds the macro* M_i *(*$COND_A$*).*

Iterating the step explained in example 3, an exhaustive test generation for all modules is obtained, thus detecting all the (detectable) faults.

A.4 Results

In order to evaluate the proposed approach, the full-scan version of the ITC'99 benchmarks were used in their optimized version, as provided by [16].

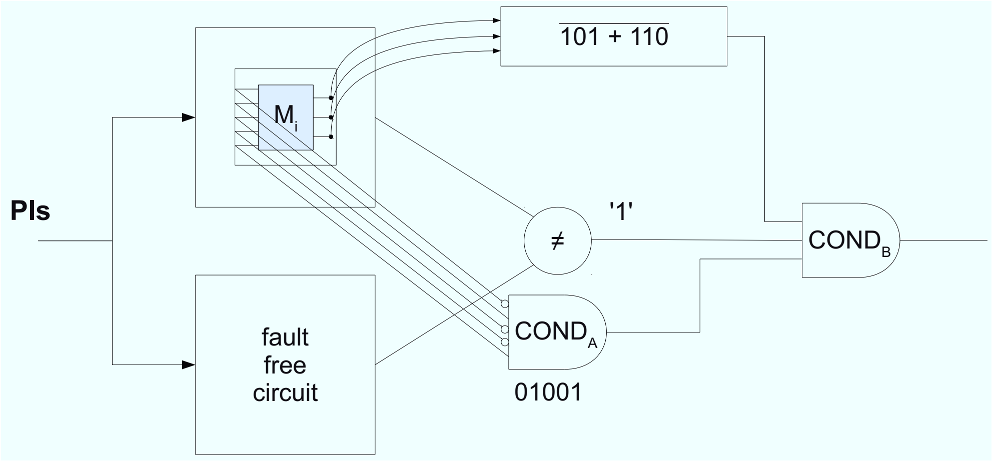


Figure A.3: Deterministic test generation step example

Since these circuits are described at the gate level, the `reduce_depth` command of SIS [47] was used, that allows to cluster more nodes in a single output one. Moreover, since the achieved nodes are single output ones, a very simple approach that joins blocks sharing some input and cube to obtain multi-output blocks was used. Of course the complexity and cost of these circuits is not representative of logic bricks based circuits designed using dedicated tools. However, they are still representative of the computational problems encountered by fault simulation and test generation tools. Table A.1 shows the characteristics of the circuits obtained after such transformations.

Table A.2, instead, shows the test generation algorithm results for the considered benchmarks. As can be seen, the first column (i.e. the tests number) shows two values in sum between them. These values are those resulting from the pseudo random approach and the deterministic one, respectively.

With regard to CPU times, they are quite linearly related to circuits' sizes (i.e. the number of cells).

bench name	**input no.**	**output no.**	**cells no.**	**average fan-in**	**average fan-out**	**average literals**
b03	35	34	41	4.26	1.60	7.85
b04	77	74	163	3.95	1.77	7.61
b05	35	70	156	3.85	1.69	7.61
b06	11	15	18	2.33	1.50	3.55
b07	50	57	125	4.25	1.75	8.22
b08	30	25	33	5.03	1.81	9.12
b09	29	29	47	4.48	1.68	9.91
b10	28	23	37	4.89	1.51	7.18
b11	38	37	97	5.77	1.77	13.96
b12	126	127	250	4.74	1.52	8.90
b13	62	63	51	4.86	1.84	12.62
b14	276	298	2311	3.75	1.25	5.66

Table A.1: Characteristics of the benchmarks set

bench name	**test no.**	**coverage value**	**coverage value**	**CPU time**
b03	641+290	51.88	53.22	519
b04	2865+7874	50.41	54.85	36219
b05	894+626	44.84	48.81	3955
b06	224+0	72.07	72.07	7
b07	760+445	36.61	40.25	6425
b08	225+169	55.00	56.69	307
b09	2496+873	50.82	57.25	2576
b10	243+0	48.86	48.86	3495
b11	2658+287	39.48	40.62	45894
b12	923+560	57.65	62.96	6259
b13	2091+335	65.71	66.28	1370
b14	7130+4683	48.86	54.69	1259657

Table A.2: Results after the application of the algorithm

List of Symbols and Abbreviations

Abbreviation	Description	Definition
s_i	i-th signal	page 34
u_i	fault-free value of s_i with the first test vector	page 34
v_i	fault-free value of s_i with the second test vector	page 34
α_i	variable denoting the presence of a delay fault on s_i	page 35
w_i	effectual value of s_i	page 35
σ_b	error function for block b	page 36
$\rho_{b,k}$	robust constraints for input k of block b	page 37
$\eta_{b,k}$	robust tests for input k of b	page 37
$\mu_{b,k}$	non-robust (minus robust) constraints for input k of block b	page 39
$\theta_{b,k}$	non-robust (minus robust) tests for input k of block b	page 39
$\xi_{b,k}$	non-robust tests for input k of block b	page 39
ψ	cost function for tests quality	page 41
$\nu_{b,k}$	functional sensitization tests for input k of block b	page 41
h_i	signal denoting the possible presence of a hazard on signal s_i	page 42
χ_i	uncertainty on the value sampled in case of hazard on signal s_i	page 42
γ_c	static hazard description function of block c	page 44

Abbreviation	Description	Definition
n_c	input configuration that cannot give rise to a hazard on block c	page 45
h_c	input configuration that may produce an hazard (n'_c) on block c	page 45
C_r	percentage of robustly detected pattern delay faults	page 64
P_r	percentage of paths with at least a robustly testable pattern delay fault	page 65
C_{nr}	percentage of non-robustly detected pattern delay faults	page 65
P_{nr}	percentage of paths with at least a non-robustly testable pattern delay fault	page 65

List of Figures

List of Tables

Index