Politecnico di Torino

POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Evaluating the cell-aware fault coverage of functional test programs

---

Developed in collaboration with:
STMICROELECTRONICS srl

Advisor
Matteo Sonza Reorda
Co-Advisor
Michelangelo Grosso
Co-Advisor
Riccardo Cantoro

Candidate
Iacopo Guglielminetti

A.A.2022

Contents

1 Introduction

Nowadays the complexity of integrated circuits (IC) continues to increase, as well as the demand for them in a wide range of markets, like the automotive, the IoT and consumer electronics. In the automotive industry, system on chip (SoC) and ICs are used to control sensors systems like parking systems, brakes, engines and more. So these components are used in safety-critical situations. Electronic companies have to face very high safety standards and constraints in order to be sure that the ICs and SoCs do not allow the overall system to interrupt its normal activities or cause damage to the final user. To assure the quality of the produced ICs, semiconductors companies have to test them before they will be sold. The scope of this test is to decrease the percentage of devices that due to some production defects, do not behave as expected and reach the application field. In the last years, many new testing methods were developed. This thesis focuses on the development of a working testing flow able to test ICs for increasing their reliability.

1.1 Objective and Motivations

The objective of this thesis is to support a test flow by means of fault simulation, i.e., the evaluation of the effectiveness a set of test stimuli for testing ICs with the cell-aware test (CAT) approach. In particular, this flow is based on with functional test, which consists in the application of test stimuli at the inputs of the unit under test and observing the output results in mission-like circuit configuration. Starting from a general netlist of an IC, the flow is able to create a list of CAT faults that represents the real defect of the design, fault simulates the list with the inputs functional patterns and then produces a fault report. The overview of the developed work is reported in the figure1.1. In particular, the selected fault simulator used was Z01X, a tool developed by Synopsys. Nowadays fault simulators that work with CAT and ICs with design for testability (DfT) already exist. In fact, using commercial tools such as TetraMax (TMAX) automatic test pattern generator (ATPG by Synopsys) is possible to test ICs with scan chains as DfT logic. The main motivation behind this thesis is the development of a complete testing flow that works with functional test programs. In fact, during the years, like reported in the paper [3], it was demonstrated that functional tests with lower fault coverage can complement scan testing and contribute to reaching a minor defect level where the defect, i.e., the percentage of faulty devices that pass the post-production tests.

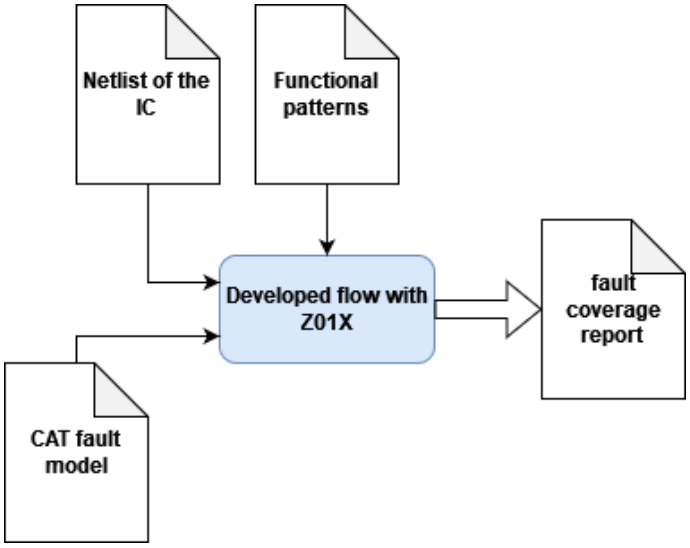


Figure 1.1: Overall description of the developed testing flow

To validate the flow different IC netlists with different complexity have been used. The more complex one was the openMSP430, a 16 bit microcontroller core which specification were written in Verilog.
The motivation behind the choice of this processor is that the number of CAT faults present in it is adapt to make the used tools produce results in a reasonable amount of time.

1.2 Explanation of the problem

Assessing the correct behaviour of an IC or a Soc on-site requires functional test patterns, also at the end of the production phase in addition to Design for Testability (DfT) testing technique. In the automotive sector, self-test of the electronic component is used. For example, a car before key-on performs a series of tests on the ICs. With the development of functional tests with CAT faults, a more precise diagnosis can be performed by the system. Moreover, as reported in the paper [1], it was demonstrated that a large number of faulty ICs labelled as good after the production phase are due to intra-cells defects, which are the main target of cell-aware testing.

1.3 Brief summary

The main works and activities of this thesis could be summarized as follow:

- The study of the cell-aware fault model;
- The study of a general flow for CAT;

- The developing of a fully working flow for CAT testing using Z01X;
- The comparison of Z01X results with the expected ones on a small circuit to understand how much the tools results are reliable;
- The testing of different circuits with CAT faults using functional test patterns;
- The comparison of testing the circuits with functional test patterns and scan test patterns;
- The comparison of fault coverage between the CAT fault model and stuck-at faults (SAF) model on different circuits;
- The functional testing with CAT faults on a real IC.

1.4 Thesis structure

This thesis is divided into chapters witch contains the following information:

- chapter 2 summarizes and report all the necessary information to understand the work developed in this thesis and the produced results. The arguments range from the concept of testing ICs to the faults models used in the thesis, from the different tests generation tecniques to the netlist structure of the unit under test;
- chapter 3 summarizes a general approach for developing a testing flow able to work with CAT fault model and test any possible ICs and SOCs described at gate-level netlist. The three main explained steps are the creation of the CAT model, the developing of a complete CAT fault list and the use of a CAT fault simulator to produce the final results;
- chapter 4 describes in detail all the passages and tools needed for developing a testing flow able to work with the CAT fault model. Starting from the software that creates the CAT fault model for the logic synthesis library, then it is presented the ad-hoc program developed to generate the CAT fault list that can be read by the selected fault simulator, Z01X. At last, is presented how to correctly read the produced results;
- chapter 5 presents the results of the developed CAT flow. The fault simulation was performed on gate-level netlists of circuits with variable complexity. The tests were made with functional test patterns and scan test patterns. Using the same test stimuli the SAF and CAT coverage comparison is presented. At last, the developed CAT flow was used to simulate functional patterns on a CPU of a microcontroller named openMSP430;
- chapter 6 reports conclusions about the results obtained in the chapter 5.

2 Background

This chapter presents all the fundamental theory to understand the work done in this thesis. The first step is to provide a summary on the production flow of an IC, in order to understand at what step the testing task is performed and why. The second section presents the testing general concept and how it is used to distinguish between good and "faulty" devices. Then it is introduced the fault simulation concept and how the results are reported. After the different fault models used in this thesis are analyzed:

- Stuck-at fault model;
- Transition delay fault model;
- CAT fault model.

Then the different tests generation techniques are analyzed, in particular, three of them:

- ATPG;
- Random patterns;
- functional patterns.

Then the main difference between the CAT, SAF and transition delay faults are analyzed. The CAT ATPG is presented. At last, are reported the structure of a gate-level netlist and the DfT techniques used in this thesis, as well as the cell description at transistor level.

2.1 ASIC design and production flow

The abbreviation ASIC stands for Application Specific Integrated Circuits. The scope of these semiconductor devices is to provide a specific solution for a specific problem. They are different from Field Programmable Gate Arrays (FPGA), where the hardware inside can be programmed many times to offer solutions for different applications. The ASIC flow remains the same despite what the ICs will do once produced and what technology is used to create them. Figure 2.1 report the main steps of this flow.

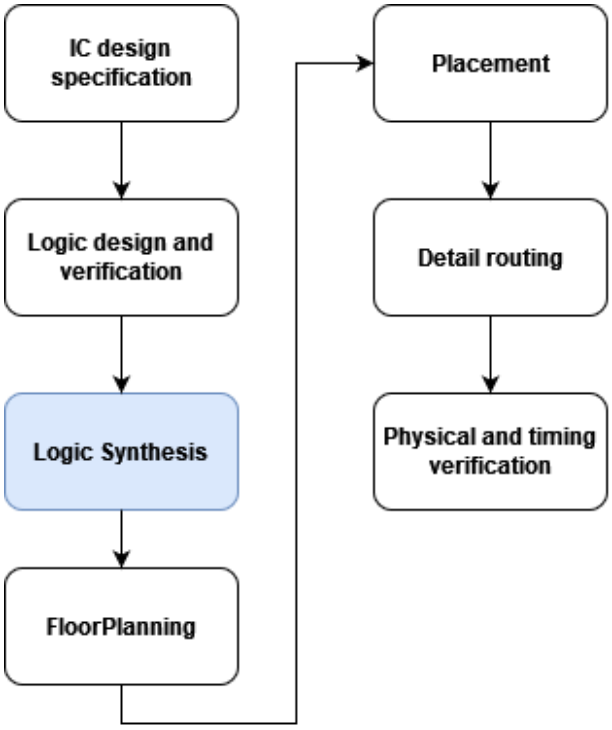


Figure 2.1: ASIC design flow

The first step is the IC design specification. When a company wants to launch on the market a new product, the first phase consists into understand what the possible customers will need and what they expect from the product. All the features that this new product will satisfy are summarised in a high-level product specifications list. For example, the new ASIC that a company want to develop could be a new processor. A list of possible high-level specifications could be the number of cores, the data parallelism, the interface used to interact with other ICs, the possibility of floating point operation, etc... Then the specifications are split into functional blocks where the relationships between them are specified, for example, the data exchange. Then the specifications are described in a high level programming language like system C, also useful for hardware and software partitioning.
The second step is the Logic design and verification, where the frontend developer of the ASIC flow transforms into hardware description language (HDL) the specifications from the previous step. The main HDLs used for this step are VHDL, Verilog and System Verilog. They can model and "describe" the data and control flow of ICs like programming languages. In the figure 2.2 is reported an example of VHDL code.

```
entity CU is
--  Port ( );
    generic(NBIT: integer := 32;
            NBITSignal: integer := 68;
            NBITFUNC: integer := 11;
            NBITOPCODE: integer := 6;
            NBITREGS: integer := 15;
            NBITTYPE: integer := 4);
    port(
    IR: in std_logic_vector(NBIT-1 downto 0);
    reset: in std_logic;
    enable: in std_logic;
    clk: in std_logic;
    jumpTaken: in std_logic;

    stallIFStage: out std_logic;
    resetIFStage: out std_logic;
    enableIFStage: out std_logic;
    RDSelector: out std_logic;
```

Figure 2.2: VHDL code example

The HDLs are used to pass from the specification level to the register transfer level (RTL). The third step is the Logic synthesis. Starting from the RTL description of an ICs, using a synthesis library which contains the information about the area, timing and power of the cells inside it, it creates a logic gate-level netlist description of the IC. The cells are the main building block used by the Logic synthesis. They perform basic combinational operations such as nand, or, xnor and they implement memory elements, like the flip-flops. At this step the tools that perform the synthesis take into account the constraints about the timing, area and power and create an suitable gate-level netlist. An example of synthesis process is reported in the figure 3.2, where two cells from a logic synthesis library named AND2_X1 and OR2_X1 are used to translate the starting RTL code written in Verilog.

```
module synthesisExample(A,B,C,result);
input A,B,C;
output result;

 assign result= (A & B) | C ;
endmodule
```

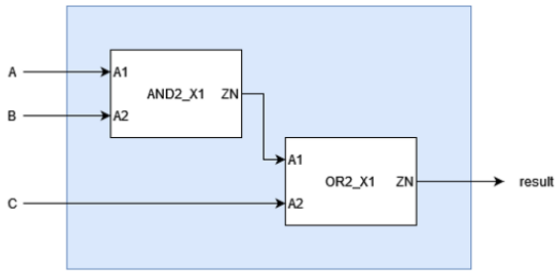


Figure 2.3: Synthesis process example

The fourth step is the FloorPlanning, where the area of the die in which the ICs will be placed is divided into physical partitions that take into account the possibility of future growth and the area requirements from the logic synthesis step. The fifth step is the Placement, where all the cells are planned to be placed in a legal position on the die. Also, this passage tries to minimize the global wire length, global route congestion and reduce the communication timing. The FloorPlanning and the Placement are just the preliminary steps of the Detail routing, where the metal layers and the die are modelled to perform the expected function of the ICs, creating the less possible DRC (Design Rule Check) violations. In the figure 2.4 is reported an example of physical design generated

after the ASIC flow with the tool Innovus, by Cadence.

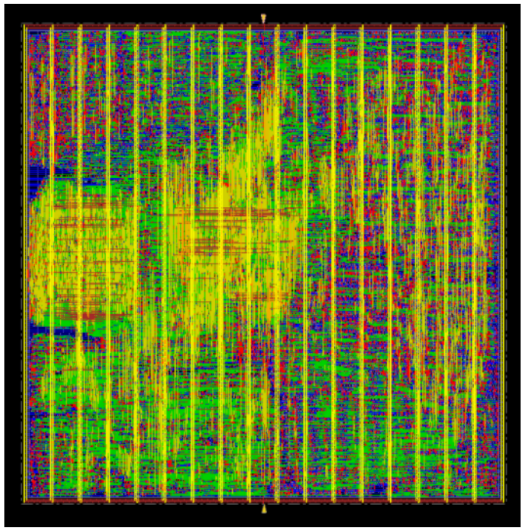


Figure 2.4: Example of physical design created by Innovus

All the test stimuli developed in this thesis act on gate-level netlists. After the design flow of an ASIC is complete, the following step is manufacturing it. The ASIC production flow is made of different steps.
The production process of an IC starts from a wafer i.e., a smooth and thin slice of semiconductor. The most used type of semiconductor is crystalline silicon. To the wafer is applied the front-end-of-line (FEOL) process. This step aims at creating the single electronic elements that will be inside the final IC. The fundamental electronic part created in this step is the transistor. The transistors are modelized on the surface of the wafer die with many processes, like the doping of the semiconductor in which n-type or p-type regions are created on the wafer, which modifies the conductivity of the semiconductor. Another important process is the etching, used to remove the superfluous part on the wafer. After the FEOL is complete, on the wafer slice are present the final transistors with unconnected pins. The second main step is the back-end-of-line (BEOL), where the transistors are connected together with many levels of metal layers. The interconnections scheme of course is based on the design flow results of the ASIC that need to be produced. After the BEOL is complete, the resulting IC on the wafer is tested with special functional test patterns applied on the circuits with special probes. There are many types of this test. Then every single netlist on the wafers is separated from the others. At last, a supporting case that isolates the semiconductors product from the environment is created, this process is named packaging. The functional tests created during the design flow of an ASIC are applied at the end-of-manufacturing phase.

2.2 Introduction to testing

Nowadays, for a large range of electronic products, dependability is the most important parameter for the quality of a product. The misbehaviour of an IC is a short or long period of time in which the system shows an unexpected behaviour that does not respect its specification. There are 2 main categories of misbehaviours, static and dynamic. If the unexpected behaviour consists in wrong results on the outputs, the misbehaviour is static otherwise if the unexpected behaviour consists in correct results of the outputs but the timing of them is wrong, the misbehaviour is dynamic. For example, if given some input the outputs of the electronic components is not the expected one, then static misbehaviour is present. The misbehaviours decrease the dependability of a system, but what causes them? Failures that derive from defects are the causes of misbehaviours. A fault is the logic model of a defect in the system that belong to the hardware. A fault does not necessary produce a failure in the system, in fact, it needs to be activated by precise input conditions. After the activation, the fault produces an internal error in the system and, only if possible, it propagates the effects to the outside creating an observable failure. Now that the concept of fault is fully presented, the following step is to give a definition to dependability, that "is the science of studying faults and developing techniques to implement dependable systems". How could tests help in increasing the dependability of ICs? Faults could arise in every moment of the life cycle of a system, in particular, there are 4 main phases:

- The specification phase, where the features of the product are transformed into specifications;
- The design phase, where the specifications are transformed into descriptions of the system;
- The production phase, where the physical description of the design is transformed into a real product;
- The operation phase, where the final product interacts with the user and the surrounding environment.

Testing can be performed at the end-of-manufacturing phase i.e., the end of production phase or during the operation phase, for example by running self-tests performed by the system itself. This means that testing an IC allows to reduce the number of faulty products after the production phase. In fact, during this step Physical defects like transistor defects, open and shorts in the metal layers of the ICs could occur. The basic procedure to apply tests on the unit under test (UUT) is reported in the figure 2.5.

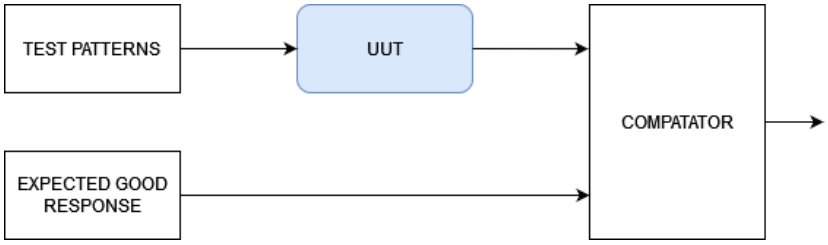


Figure 2.5: Procedure example to apply test of a unit under test

After the tests are applied, the UUTs that produce static or dynamic misbehaviour are discarded and labelled as faulty. It exists very different test modalities and typologies to apply to the UUTs, that have different costs in term of time and complexity. To determine if the different tests performed were useful to increase the dependability of the system, the metric usually taken into account to measure the product quality is the defect level (DL). The DL is the number, expressed in percentage, of faulty devices that are able to pass the end-of-manufacturing tests. These faulty systems are then sold as good. They are usually few, in fact, are expressed in part per million (ppm). The first method to compute the DL of a product is to collect all the faulty machines that return from the different purchasers and count them, but usually not all of them return to the producer, so a formula was proposed by Williams and Brown in 1981 and it states: DL= $1-Y^{1-T}$. T is the fault coverage reached by the end-of-manufacturing tests i.e., the sum of the detected faults over the total number of faults present in the UUT. Y is the Yield i.e., the percentage of good systems over all the existing ones. The validity of the DL formula strictly depends on the fault model adopted for the tests performed. Summing up, testing is mainly performed at the end-of-manufacturing phase but not only, but it also consists in applying stimuli to the UUT and detecting faulty products that do not behave as expected due to physical defects in the production process. The tests stimuli are developed with different methodologies and the fault coverages that they guarantee are calculated using fault simulators. These elements will be presented in the following section.

2.3 The fault simulation concept

A fault simulator is the main tool needed for the evaluation of test set. The three fundamental inputs needed are the gate-level netlist of the UUT, the list of test patterns created by other tools like ATPG, LFSRs and the list of faults from a fault generator. The flow for fault simulation is reported in the figure 2.6.

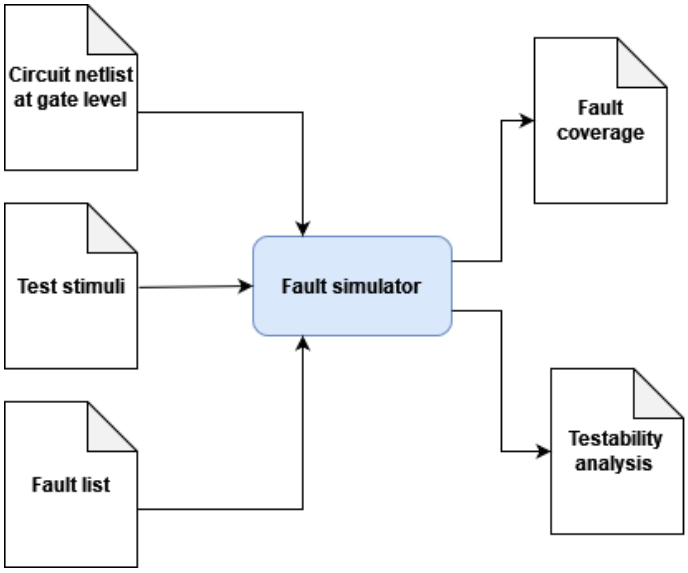


Figure 2.6: Example of fault simulator flow

The main results of fault simulator are:

- Fault Coverage computation, this is the main purpose of a fault simulator. The list of faults in input is analysed and a final report is produced, in which they list the faults with their status, for example, they could be detected by the input patterns simulated or not;
- Testability analysis, depending on the tool used, many analyses could be performed on the UUT. For example, the input gate-netlist is usually split into modules that represent different parts of the ICs. The fault simulator could list these modules and for every one of them show the fault coverage, in order to understand the areas of the UUT that are difficult to test with the given inputs patterns;
- Fault dictionary, that consists in a list of faults, test vectors and the behaviour of the simulations of each fault in the UUT.

The general behaviour of a fault simulator consists in two main steps. The first is to simulate test stimuli of the UUT without faults and check if the expected values of the inputs patterns are correct. Then a fault is taken from the faults list and it is injected into the UUT, so a faulty machine is created. The inputs stimuli are simulated on it, to see if it is possible to observe a misbehaviour on the outputs and then according to the results the faults are classified. This last step is repeated for every fault in the fault list. The complexity depends on the number of faults in the fault list and on the size of the UUT. For example, if the number of faults is equal to m, and the number of cells present in the gate-level netlist is n, the total complexity will be O(m*n). Different techniques were developed to improve the performance of such flow, which are currently available in commercial tools.

2.4 Faults simulation results classification

The most important output produced by fault simulation is the fault coverage report. Here every fault in the fault list is divided into categories depending on the simulations results. These cathegories are somewhat different depending on the specific tool used. For this thesis, two fault simulators were used, TMAX and Z01X. To understand the results in chapter 5 I report the most important faults types in the TMAX user guide [4] and Z01X user guide [5] presents in the faults coverage reports. Regarding TMAX:

- DT (Detected): a fault is classified as detected if during the fault simulation misbehaviour could be detected on the outputs of the UUT or by implication. This class is divided into more subclasses, the most important ones are DS (Detected by Simulation) and DI Detected by Implication;
- ND (Not detected): it indicates that the test patterns were not able to cover this fault; the 2 main reasons could be that the simulation inputs patterns never produces the excitation conditions for the faults i.e., it is a NC (not controlled) fault, or that the effect of the fault are not observable on UUT ouputs i.e., it is a NO (Not observed) fault;
- UD (Undetectable): the "undetectable" fault can not be detected by the simulation by all the possible inputs. They are not considered for the test coverage because they can not produce misbehavior on the outputs of the UUT;
- AU (ATPG Untestable): these faults can neither be detected under the current ATPG conditions nor proved redundant;
- PT (Possibly Detected): the faulty machine of the fault simulator will simulate an X (undetermined value) on the outputs in response to the presence of the fault.

Regarding Z01X:

- DD (Dropped Detected): the fault was detected by the test stimuli;
- PT (Potential Detect): the fault can not be classified with certainty as detected. More precisely, these faults produce on faulty machines an unknown status;
- IA (Illegal Access): the fault propagates to an unallocated array/class so it was dropped from the simulation;
- NS (Not Strobed): the fault has a path to an observable point, such as a primary output, but that location was never strobed. Given some test stimuli, during the fault simulation the fault simulator checks the output values after a specific amount of time is elapsed, the strobe period. For example, if a strobe period of 20ns was defined, every time this interval is elapsed the fault simulator checks the output values;
- NO (Not Observed): during simulation, the fault does not causes mismatching activity, in the faulty machine, observable on the outputs of the UUT;

- NC (Not Controlled): the fault does not toggle during simulation;
- NA (Not Attempted): Before the starting of a fault simulation, all the faults are Not attempted. If after the simulation some faults are still not attempted, something went wrong;
- ND (Not Detected): in the faulty machine, a not detected fault during simulation never produces a mismatch on the outputs in comparison to a good machine;
- NI (Not injected).

2.5 Fault models

The misbehaviour at the end-of-manufacturing stage of an IC is produced by a real physical defect present in the circuit. To enable an analytical approach to test generation and to provide a means to express the quality of the test, the abstract concept of fault model is used, which has been demonstrated to have some correlation with the defect effects on the circuit. To decrease the DL of an IC the fault model (or the fault models) used for testing need to be more accurate possible to describe the real defects. Over the years many were proposed, this thesis focuses on CAT fault model but it is also compared with the SAF model and Transition delay faults model (TDF).

2.5.1 Stuck-at fault model

The SAF is a standard and popular model to test ICs and SoCs. Many faults simulators and fault generators are able to work with this fault model because it is easy to represent and use. The idea behind this fault model is that a defect inside the gate-level netlist of a circuit could cause the interconnections between the cells to assume a only fixed value that can not change anymore. The two possible values in which the interconnections could "stuck" are 0 (stuck-at-0, sa0) and 1 (stuck-at-1, sa1). There is an empiric correlation between the coverage of the SAFs and the real defect coverage, but it is not complete. The model deals with the cells in the logic-level netlist as black boxes, that accelerates the generation during simulation of SAF, decreasing the complexity of the model but also the precision of it at model the real defect. The total number of stuck-at faults is equal to 2n, where n is the number of interconnections between the cells ports. Figure 2.7 reports how to place correctly the SAF in a gate-level netlist.

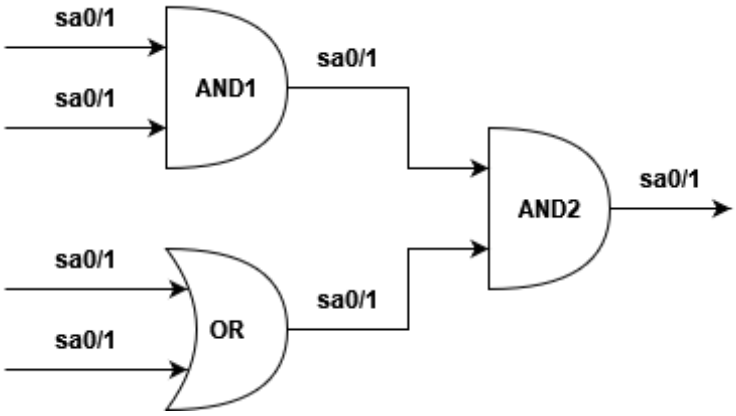


Figure 2.7: Example of SAFs in a gate-level netlist

To detect a SAF the suitable input patterns need to be generated. For example, if at the output of a cell there is a sa0, in input a pattern able to produce at the output a 1 is needed. Then the misbehaviour due to the fault needs to be propagated to the output. For example in the circuit in the figure 2.8 to be able to detect the sa1 on the circuit output port we need to produce a zero at the AND1 cell and then propagate the erroneous 1 to the available output.

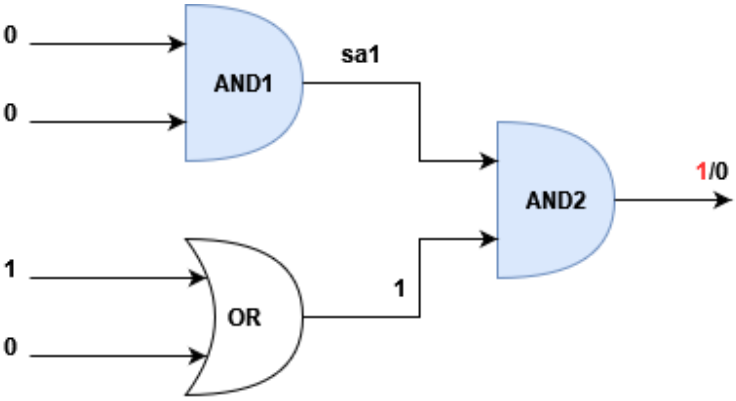


Figure 2.8: Example of sa1 detection

The normal output given the input reported in the figure is 0. With the sa1 present, the output change from 0 to 1 (so the result of the circuit with the fault present is reported in red). It is not needed to test every single stuck-at in a circuit. Luckily, there exist some local equivalence principles that allow the fault simulator to test only one of the stuck-at in a certain group. If it is covered, so are the others and vice-versa. An example is shown in figure 2.10.

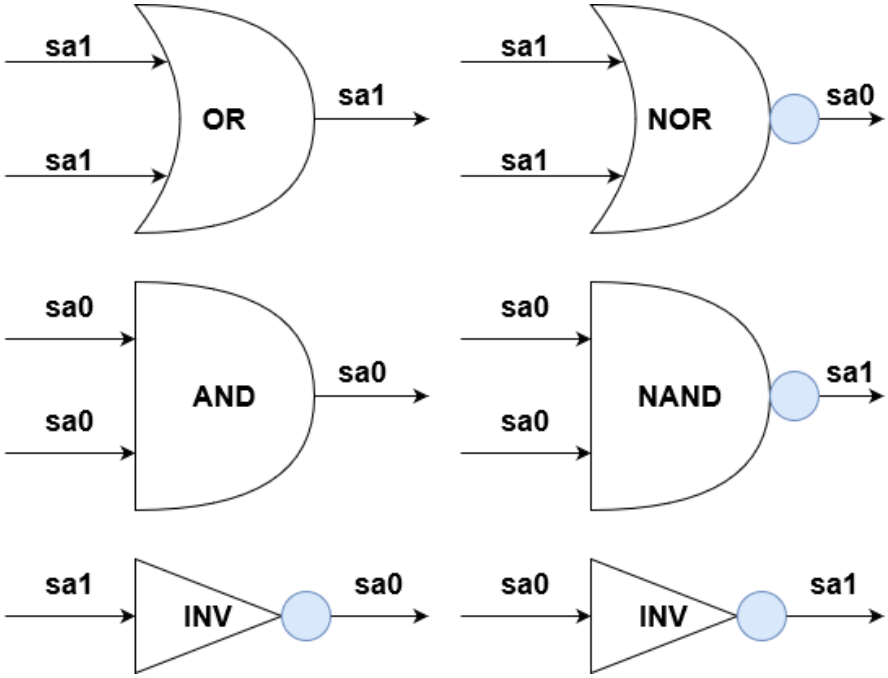


Figure 2.9: Equivalence rules of SAF

Summing up, SAF is a standard model for testing ICs and SoCs but is not able to correctly model intra-cell defects because the cells inside the gate-level netlist are threated as black boxes.

2.5.2 Transition delay fault model

The working frequencies of the ICs are increasing constantly, so the defects that characterize the timing (delay defects) are now very common. To detect them, the tests need to be made at speed, in other words, the clock speed used when applying the test needs to be the equal to the nominal speed. The delay defects may depend on the process variation i.e., the variations that affect transistors when they are produced; with the decreasing transistor sizes, this phenomenon becomes more frequent. In the figure 2.10 is reported an example of general delay defect. In the input of flip-flop4, in the precedent clock cycle, the value was 0. In the current clock cycle, the combinational logic between the flip-flops produces a 1, but unfortunately the propagation of the results is too slow due to a delay defect, so at the next clock cycle the flip-flop4 reads 0 instead of 1.

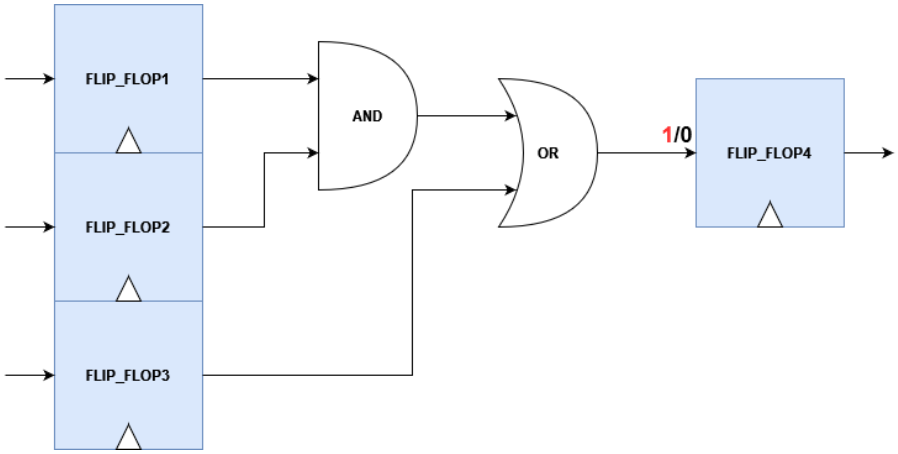


Figure 2.10: Example of delay defects

Delay defects are usually described with two fault models, transition delay and path delay. TDF models the defects of a gate that prevents the output of it to pass from 0 to 1 (slow-to-rise) or from 1 to 0 (slow-to-fall). The total number of TDF is equal to the number of cells in the circuit netlist and the number of their inputs. To detect a TDF, two patterns are necessary. The first sets the desired value on the output port of the gate, the second creates the transition. In the figure 2.11 is reported an example of slow-to-rise TDF.

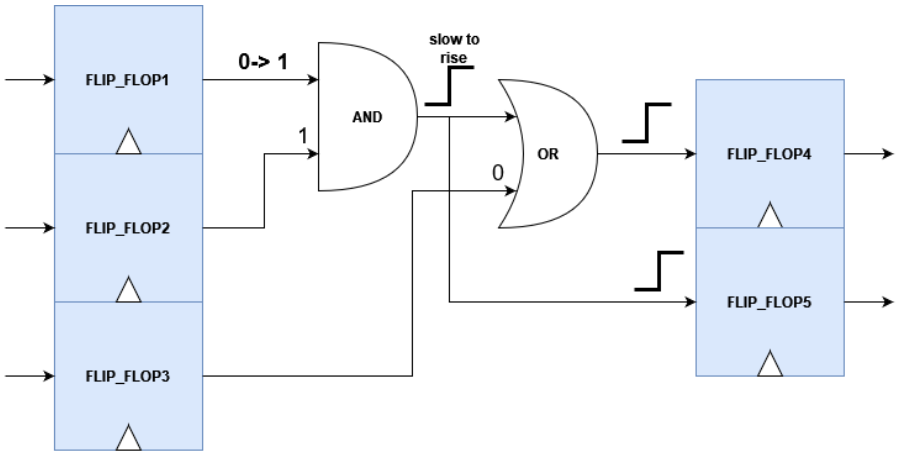


Figure 2.11: Example of TDF

Like SAF, this model treats cells in the netlist of the IC like black boxes.

2.5.3 Fault modelling with the CAT approach

The CAT approach is based on the analysis of the internal structure of every cell in the library used for the logic synthesis in order to correctly represent all the possible

intra-cells defects and understand what input patterns are needed to test them. The main objective of the resulting fault model is to overcome the limitation of SAF and TDF fault models.
Let us start with the basic concepts of CAT. The electronic model of a library cell, despite the simple boolean function performed, could be very complex, with resistors, capacitors and transistors connected together in different ways.
In the figure 2.12 is shown the LTspice model of cell AND2_X1, present in the open-source technology library Silvaco Nangate 45nm (nangate) [17].

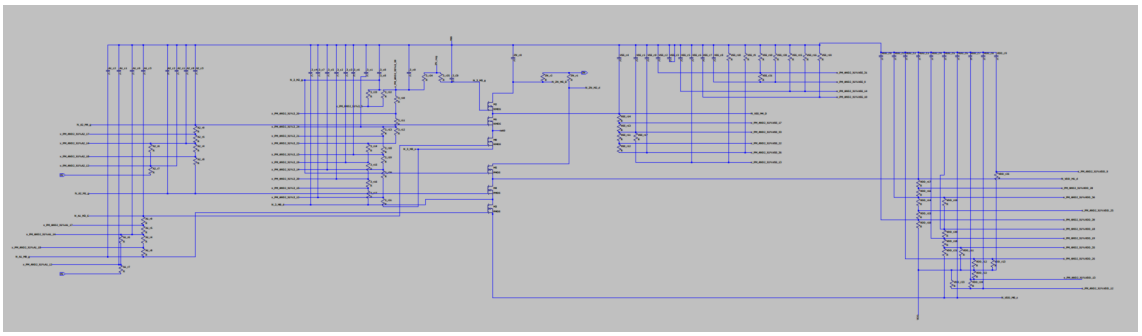


Figure 2.12: Example of cell netlist at transistor level

To easily present the CAT fault model, a simple cell scheme easier to understand was developed. It also contains the typical intra-cell defects that are modelized by CAT. As reported in the figure 2.13, the cell is made by 2 input ports A and B, 1 output port ZN and 3 transistors, 2 NMOS and 1 PMOS. There are 6 main categories of intra-cell defects that can be modelized by CAT according to the paper [13], and they are:

- short, i.e., any short defects in the cell, like between objects in the same metal layer or different layers;
- open, i.e., any intra-cell open defects, such as an open in vias or in the metal layer;
- transistor open, i.e., any cell defects that could switch a transistor off;
- transistor short, i.e., any cell defects that could switch a transistor on;
- port open, i.e, a disconnected ports;
- port short, i.e, any short between the port and VDD, VSS or another port.

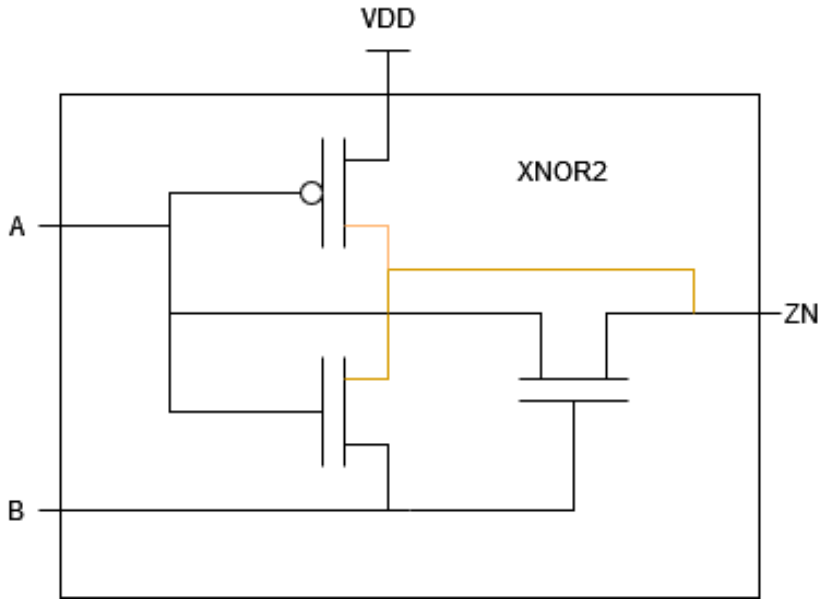


Figure 2.13: Simple model of a XNOR2 cell

In order to analyze every fault of this category, a representative case for every one of them was created and then inserted into the XNOR2. The figure 2.14 show the cell with the parasitic components, where a maximum (or minimum) value of the main parameter represents a possible defect (e.g., an infinite capacitance becomes a short).

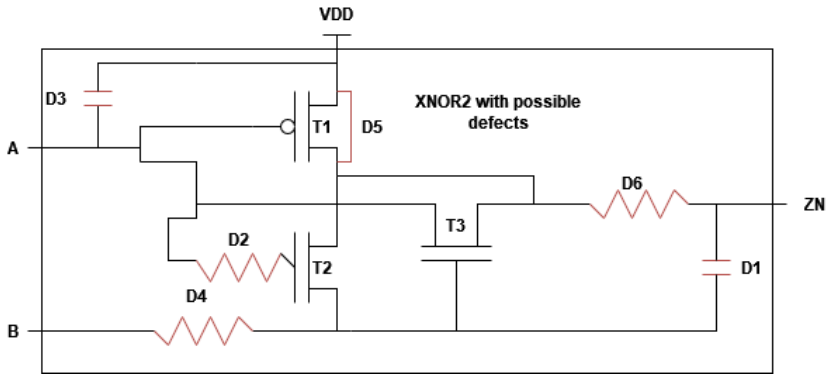


Figure 2.14: Simple model of a XNOR2 cell with defects

The list of the defects is:

- D1, that represents a short;
- D2, that represents a transistor open;
- D3, that represents a port short;

- D4, that represents a port open ;
- D5, that represents a transistor short;
- D6, that represents an open.

The inputs and the output of the XNOR2 cell can assume only two values, 0 and 1. The defects in the netlist may cause the output to assume a different value from the expected one, depending on the input values provided. To create a characterization for the defects present in the cell, a possible method is to simulate the behaviour of the cell without the defects and record the results. Then inject one by one the defects in the cell, and see what inputs patterns are able to produce a difference in the output between the faulty and the good machine. Regarding the XNOR2 example, the possible input patterns are 4. For every defect present in the cell, if a pattern is able to detect it, this information is saved. There are several ways to represent CAT fault model, one of them could be to summarize the results of the simulation in a table. For every fault, a zero indicates that the corresponding input pattern is not able to cover the defect, and a 1 indicates that the corresponding input pattern is able to cover the defect. Regarding the XNOR2 example, the detection table of the defects is reported below. Looking at the table, it

A	B	ZN	D1	D2	D3	D4	D5	D6
0	0	1	1	0	1	0	0	1
0	1	0	1	0	1	1	1	0
1	0	0	0	0	0	0	1	0
1	1	1	0	1	0	1	0	1

Table 2.1: First version of the Detection table of the cell.

may occur that some defects, intrinsically different, could be detected in the same way. In fact, D3 and D1 are the same from the point of view of CAT fault model. So we can summarize the previous table in the one below.

A	B	ZN	D1	D2	D4	D5	D6
0	0	1	1	0	0	0	1
0	1	0	1	0	1	1	0
1	0	0	0	0	0	1	0
1	1	1	0	1	1	0	1

Table 2.2: Detection table of the cell.

D3 is equivalent to D1, this means that when D1 is detected, also D3 is detected. After this step, the CAT fault model is complete and fault simulators, ATPGs or a fault list generators can use this information to work with CAT faults. The objective of CAT testing is complement and improve the tests based on the SAF and TDF fault models. The main differences between them are reported in a specific section.

2.6 Test generation techniques

Test generation techniques are fundamental for testing. An ideal test is effective and short. In fact, they need to be able to find the larger number of defect present in the UUT in the smaller time possible. The test time factor is important and does not need to be underestimated. For example, to test an IC, 2 different test stimuli were developed, the first reached a fault coverage of 91 % with a test time t, and the second reached a fault coverage of 93 % with a test time 3t. If the target feature of the produced ICs is the quality, the second test is selected; if the the target feature of the produced ICs is the quantity, the first stimuli are selected to test the UUT. The more time is spent to test the UUT, the more time the other units that need to be tested have to wait, this causes a decrease in production efficiency and wastes an important amount of money. The end-of-manufacturing tests are performed by automatic test equipment (ATE), an automatic machine able to apply the test stimuli and compare the simulation result with the expected one.
Over the years, different techniques have been developed to create test stimuli. In particular, for this thesis 2 main methodologies have been used, the ATPG and the random test patterns generator. Also functional stimuli may be used for testing, which make the target system work in mission-like conditions.

2.6.1 ATPG

The complexity of the ICs grows exponentially over the years, making it impossible to create test patterns by hand, because it would be too difficult and time consuming. To resolve this problem automatic test pattern generators were developed. The general ATPG flow is reported in the figure 2.15.

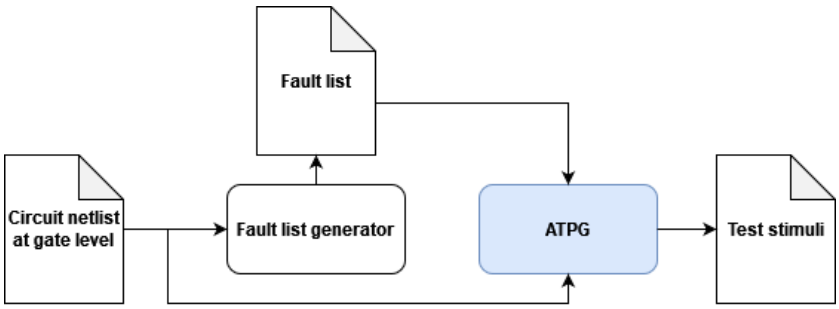


Figure 2.15: General ATPG flow

The main 2 fundamental inputs are the gate-level netlist of the ICs under test and the fault list that refer to it. The fault list generator analyses the netlist of the circuit. The output fault list does not necessarily refer to a single fault model, they could also be constituted by the list of defects from different fault models. The fault list generator once it creates a full and complete list of defects, it also removes the untestable faults from it. The term untestable indicates the faults that are not reachable by the inputs

or not connected to the outputs, so they are unuseful for testing purpose.
The fault list generator also performs fault list collapsing. Depending on the fault model selected, some faults are equivalent. Two main rules exist to understand if two faults are equivalent, the first is the test equivalence and the second is functional equivalence:

- For the testing purpose, two faults α and β are equivalent if any test pattern that test α also test β and viceversa;
- From the functional point of view, let F_c the correct behavior of an IC and let F_α the faulty behavior of the IC with a fault α present; the fault β is functional equivalent to α if the faulty behavior of the IC with this faults present are the same i.e., $F_\alpha = F_\beta$.

The fault list generator takes into account these 2 rules and creates a list of prime faults and equivalent faults. Only the prime faults are sent to the ATPG, in fact, if the ATPG creates the test pattern to cover the fault α and it is the prime fault of the other defects β and γ, so also this last two are covered by the pattern. Fault list collapsing speeds-up the computational time of ATPGs by decreasing the number of faults to simulate. After the jobs of the fault list generator are complete, the ATPG can start. It selects a subset of faults from the fault list, usually, these faults have common properties that make them easy to test together, and the patterns that could possibly detect them are usually the same. This step is important for decreasing the time used by the tool. Then the ATPG creates a series of patterns that could cover the selected faults; these patterns will be used by an internal fault simulator that checks if effectively the input patterns created are able to produce a misbehaviour on the outputs of the UUT, in which the selected subset of faults is present. The faults are simulated one by one. In the next step, fault dropping is performed. At the beginning of the fault simulation step, all the faults are labelled as not tested. If the test patterns created were able to cover the faults, then they are labelled as detected. If the previous steps and the simulation prove that some faults are untestable, they are dropped from the fault list. At last, if regarding some faults some computational thresholds are reached, these faults are labelled as aborted and are dropped from the simulation. After the fault dropping is performed, the ATPG checks if any faults remain to be tested: if they are finished, it saves the test patterns that were able to detect some faults in a list, and also it produces a report for the user in which it lists what faults were covered by the patterns; if they are not finished, it starts again the procedure to create the test patterns. In the figure 2.16 all the ATPG main steps are reported.

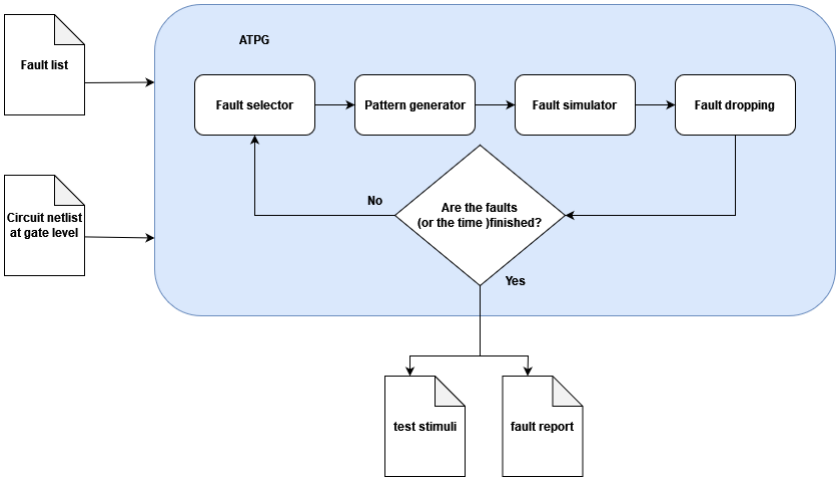


Figure 2.16: Detailed ATPG flow

To reduce the total number of test patterns needed to test the detected faults, some "test set compaction" techniques where developed. The most used ones are 2:

- Exploiting the don't care value;
- Reverse order fault simulation.

When the ATPG creates a test pattern to cover one fault, some inputs value applied to the IC netlist remain unspecified because they are not necessary to detect the fault. The ATPG could try to assign to these inputs specific values to see if the number of faults detected by the pattern increases. The second techniques consist into taking the generated test patterns and performing fault simulation of them in a reverse other. The test patterns that do not detect any faults are deleted. The complexity of the ATPG grows exponentially with the netlist dimension of the UUT. The algorithms that the ATPG uses to create the test stimuli depend on many factors, for example by the fault model used to describe the faults to test. In section 2.8 for example, is reported the procedure for the CAT faults.

2.6.2 Random patterns

Applying random patterns to the UUTs is one of the most simple methods for testing it. This method is faster, less time and resource expensive than other ATPG methods. Random patterns are often used in combination with Built-In Self-Test technique (BIST), whose purpose is to improve the test quality, reduce the test time and avoid the usage of ATEs. For the testing purpose, the patterns created for this kind of test generation technique need to appear random from the point of view of the UUTs, and deterministic from the point of view of the generator. This will allow repeating the same tests over time. These types of patterns are called Pseudo-random vectors, which are also characterized by a low temporal correlation between the patterns and uniform sampling in the

possible space of combination. The pseudo-random patterns could be created by ad-hoc hardware named Linear Feedback Shift Registers (LFSR). These components perform the polynomial division between an external "seed" and the characteristic Primitive polynomial implemented by the LFSR. The polynomial is characterised but a degree n. For example, given the polynomial x^2+x+1 the respective degree is 2. A polynomial is identified as primitive if it can perform polynomial division with the polynomials with the form x^m+1 of degree m, with m= 2^m-1, but it can not perform the division with the polynomials of any degree minor of m. The characteristic Primitive polynomial of an LFSR is made with simple flip-flops, multiplexers and xnors, so it is simple to fabricate. An example of LFSR is reported in figure 2.17.

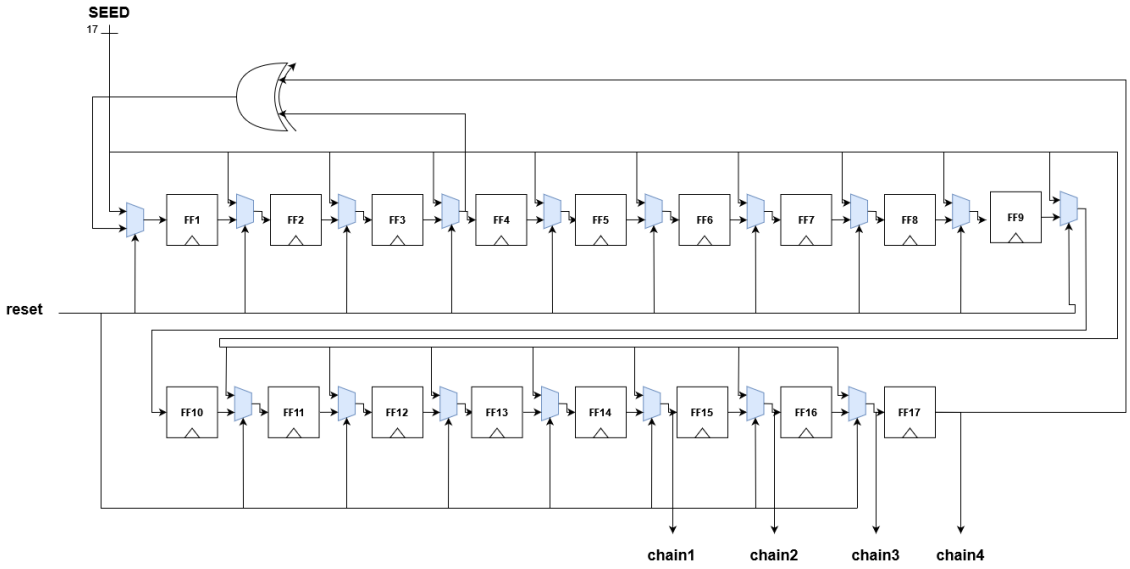


Figure 2.17: Detailed LFSR scheme

In this particular version, the outputs are 4 pseudo-random generator, which values are the same but shifted in time, this means that this LFSR in total could feed 4 input of the UUT. It is possible to break the correlations between the outputs of the LFRS by inserting different xnor between them. The number of pseudo random patterns that an LFSR can produce is 2^n-1, so the LFSR reported in the precedent figure, can produce a random sequence of 131,072 values.

2.6.3 Functional patterns

The term functional tests is intended for all the tests developed using only the functional information about the module under test, so the test stimuli format is equal to the functional stimuli. This kind of test is developed to test the function of ICs. As reported in the paper [3], it was demonstrated that the structural test and the high fault coverage obtained by the tests performed with scan chains (a kind of DfT) may be insufficient to obtain the required DL. Instead, the use of functional tests and scan tests together can

obtain a lower DL. Functional patterns are used at end-of-manufacturing testing and in in-field tests. The last typologies are particularly important because usually tests can not be performed using an ATE and structural information of the UUT are not available. An example of functional test methodologies is Software-Based Self-Test (SBST), that are a set of tests developed for SoC and processors and is used in in-field test and end-of-manufacturing test. The functional patterns are generated with different methods and then are loaded into the available memories throw the peripherals. Then they are applied and the results are collected, to check for possible misbehaviours. Figure 2.18 reports an example of SoC tested with SBST.

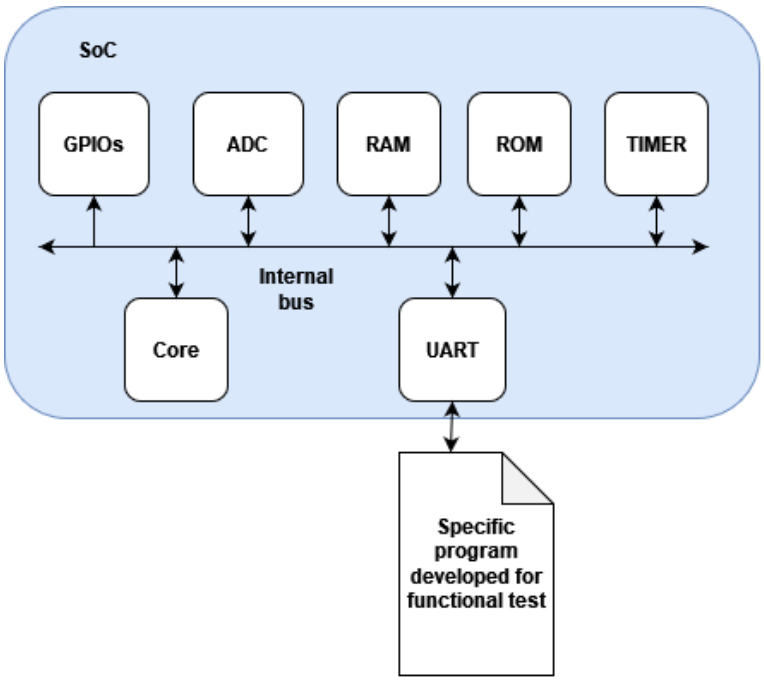


Figure 2.18: Example of SBST with a SoC

2.7 Main differences between CAT, TDF and SAF

CAT aims at producing a fault list more closely related with the real defects inside the logic cells so the CAT faults are representative of the defects present inside the cells of a logic synthesis library used to produce the gate-level netlist of the UUT. These defects are usually more numerous than the SAFs and TDFs, because the number of intra-cell defects inside the logic synthesis cells is bigger than SAF and TDF. To understand the difference between CAT faults and SAFs and TDFs, two possible CAT detection tables regarding two cells AND and INV, which are an AND gate with two inputs and an inverter, are reported in the table 2.3. Also, regarding the AND cell, in the tables 2.4 the SAFs and TDFs are reported with the patterns that are able to detect them. Slow-

to-rise is shown with the acronym STR and slow-to-fall is shown with the acronym STF. Regarding the inputs A nad B of the AND, the term R stand for rise and indicates that the corresponding port pass from assume the low state to assume the high state. The term F stand for fall and indicates that the corresponding port pass from assume the high state to assume the low state. Regarding the output ZN, R and F are the effects of the inputs changes.

A	B	ZN	D4	D5	D6	D7
0	0	0	0	0	0	1
0	1	0	1	0	0	1
1	0	0	0	1	0	1
1	1	1	0	1	1	1

A	ZN	D1	D2	D3
0	1	1	0	1
1	0	0	1	1

Table 2.3: CAT model of an AND on the left. CAT model of an INV on the right.

A	B	ZN	sa1	sa0
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

A	B	ZN	STR	STF	D8	D9	D10
0	R	0	0	0	0	0	1
R	0	0	0	0	0	0	1
0	F	0	0	0	0	0	1
R	1	R	1	0	1	0	0
F	0	0	0	0	0	0	1
1	R	R	1	0	1	0	0
F	1	F	0	1	1	0	1
1	F	F	0	1	1	1	1

Table 2.4: On the left, comparison between CAT and SAF. On the right, comparison between CAT and TDF.

Looking at the table of the inverter INV, it is possible to see that the defects D1 and D2 are equivalent to sa0 and sa1 respectively. Instead, D3 can be found when the output assume both the values zero and one. Regarding the AND cell, D4 is detectable only with 1 pattern and it can be seen as a special case of a sa1. In fact, sa1 on the AND can be detected by 3 patterns, instead sa0 by only 1. It is possible to deduce that CAT faults can be detected with a lower, equal or higher number of input patterns in comparison to SAFs, as shown by the defects D4, D1 and D3. The example shows that is possible to obtain a CAT fault equivalent to a SAF. It could happen that some CAT faults are not comparable to SAF. In fact, the effect of the defect in the cell can produce dynamic misbehaviour and not a static one. For example, the cell with the defect inside it, can reach the expected value after a certain amount of time. This kind of CAT can be compared with TDF. In the last table, D8 and D9 represent this kind of defect. These CAT faults are not strictly equal to DTF. For example, looking at D8 it could

be detected by the conditions needed by STR but also STF. Like for SAF, CAT faults could be detected with lower, equal or higher number of input patterns in comparison to TDFs.

2.8 CAT ATPG

ATPGs need to be able to find the correct test stimuli that cover the biggest number of faults regarding a fault model. To achieve these goals, ATPGs use different algorithms for every fault model. In the figure 2.16 the general algorithm was reported. According to the CAT fault model presented in the previous section, the main steps of the CAT ATPG flow can be summarized as follow. The CAT ATPG is applied to the XNOR2 presented in the precedent section, and information about the CAT fault model was summarized in the tables 2.2 reported in the section 2.5.3.

- Cycle1: the fault selector reads the table and understands that D1, D4, D5 are detectable with only one pattern, the second one in which the inputs values are A=0 and B=1, and the value of the output is ZN=0. So these faults are selected. Then the pattern generator creates the corresponding pattern. In the next step, the fault simulator takes one fault at a time and creates a faulty machine. After every simulation, it checks if the fault produces a mismatch at the output. According to that D1, D4 and D5 are all detected. The fault dropping deletes this faults from the simulation queue;

- Cycle2: the remaining faults to cover are D2 and D6. According to the table, they can be both detected by the last pattern, so they are both selected. Then the pattern generator creates the last pattern where A=1, B=1 and ZN=1. In the next step, the fault simulator simulates this new pattern on the created faulty machines and finds that both faults D2 and D6 are detected. So the fault dropping drops them from the simulation queue. Now the CAT faults are finished and the CAT ATPG had found that only 2 patterns are necessary to cover all the faults.

2.9 IC netlist at gate-level

The ATPGs, fault simulators and faults generators usually work on the gate-level netlist of the UUT. This means that starting from the RTL of the IC, it was synthesized using a standard cell library. The resulting netlist is a connection of cells of the library that together perform the same function of the circuit at the RTL level. The tools able to perform the logic synthesis usually need the HDL file that describes every cell present in the library, in order to select what of them uses to create the gate-level netlist. The library usually also includes files that describe the internal structure at the transistor-level of every cell, the models used for the transistors and files that describe the timing and the power needed by every cell. Regarding the CAT fault model, it is fundamental to have access to the files that describe the internal structure of every cell, like the one reported in the figure 2.14, otherwise it will not be possible to create the correct CAT

fault description of every cell. These constrains are not present in SAF model and TDF model, so they could work using cells like black boxes.

2.10 Cell description at transistor-level

CAT to correctly represent the intra-cell faults inside a cell needs the transistor-level netlist model of every cell inside the synthesis library. On this netlists some simulations are performed to characterize the internal defect of the cells. Depending from the tools used (so it is strictly connected to the implementation), the simulations can be different. At gate-level, the inputs and the outputs of the gates, i.e., the cells, can assume only 2 values, 0 and 1 that represents the two fundamental logic states. At transistor-level, at every logic state is associated to a voltage. To the state 1 is associated the higher voltage and to the state 0 the lower voltage. Every IC or SoC has a power supply like all the elements inside them. This concept is reported in the figure 2.19.

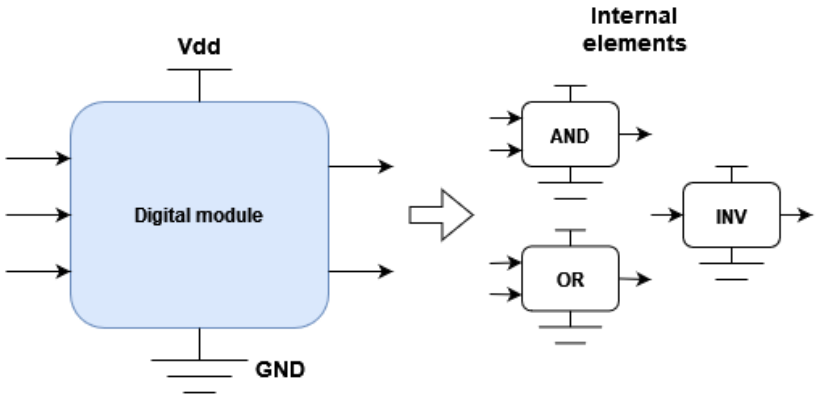


Figure 2.19: Power supply to digital modules

To work correctly, the input and output voltages need to be inside a precise interval. Depending on the technology used by the transistors inside the cells, the interval for correctly reading the inputs and the outputs could vary. The cells inside a logic synthesis refer to the same technology, so the inputs and outputs are compatible. Let V_{OL} be the output voltage for the low state and V_{OH} the output voltage for the high state. This two values in order to be correctly read by the inputs to which are connected, need to be between two values, respectively V_{OHMIN} and V_{OHMAX} regarding the high state, V_{OLMIN} and V_{OLMAX} regarding the low state. The input margins are not equal to the output bound because of the possible noise between the transmitter and the receiver. Let V_{IL} be the input voltage for the low state and V_{IH} the input voltage for the high state. This two values in order to correctly read the inputs to which are connected, need to be between two values, respectively V_{IHMIN} and V_{IHMAX} regarding the high state, V_{ILMIN} and V_{ILMAX} regarding the low state. V_T is the voltage threshold, it depends from the technology used but also to some environment factor, for example the temperature, this is why the minimum voltages of the high state do not overlap with

the maximum voltages of the low state. The representation of the correct voltage values need to make the digital components work are reported in the figure 2.20.

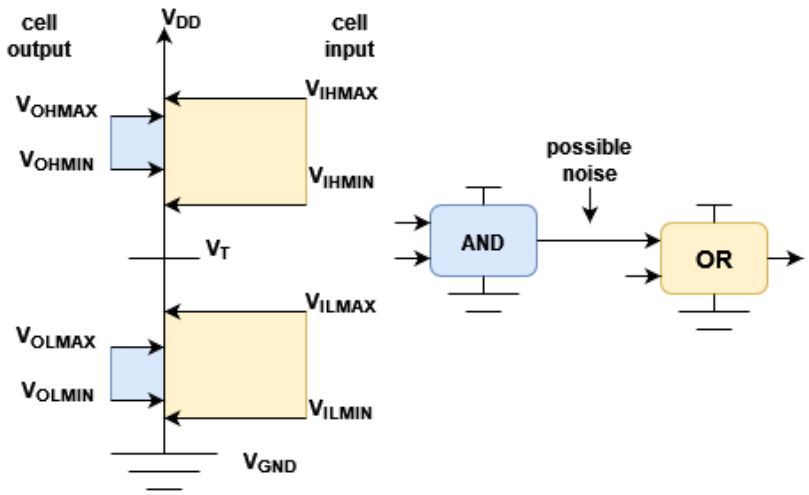


Figure 2.20: Voltages bounds for inputs and outputs

During a normal execution of test stimuli, if these bounds are not respected due to intra-cells faults, a misbehavior is produced. The misbehavior could be static or dynamic.

2.11 Design for Testability

Design for testability consists into the modification of the netlist of an IC by adding internal logic that makes the tests easier to perform. The main reasons for the adoption of DfT techniques are: reducing the test cost, reducing the test time and increasing the quality of the test.
One of the most commonly employed DfT techniques is scan testing. ATPG need to generate a test pattern able to control and observe the internal faults. This could be easily done on pure combinational circuits, but netlist with also sequential logic (like finite-state machine) could increase exponentially the effort needed by the ATPGs to create the test stimuli. The normal processors are commonly constituted by pipelines. This means that the input instructions to control them are split between internal stages inside them. A processor with a five-stage pipeline behaves in the following way: the first stage of the pipeline using the inputs produces results that are saved in the corresponding flips-flops. Then the results of the first stage are taken from the second one, and combining them with the inputs produces the second results that are saved in the corresponding flip-flops. This process allows every stage to work at the same time and the last unit produces the final result. If the ATPG wants to target a fault at the first stage of the pipeline, observing it will be very difficult because the fault effect needs to propagate throws the other stage of the pipeline. For a fault in the last stage, it occurs the opposite solution, in fact, it is easy to observe but difficult to excite. In the figure

2.23 is shown an example of faults that is difficult to observe. PIs stand for primary inputs, and POs stand for primary outputs.

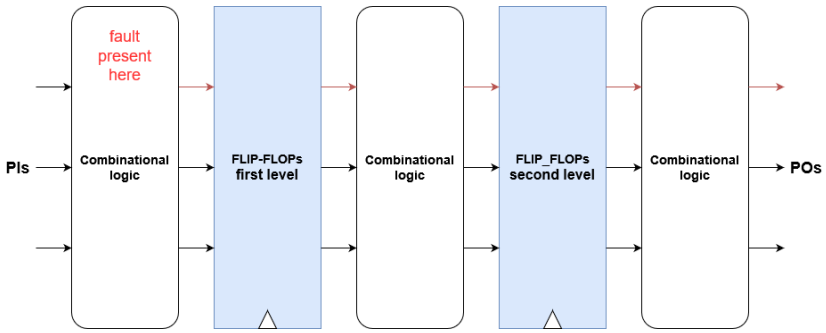


Figure 2.21: Example of hard-observable fault

The DfT method developed to avoid this problem and make the testability of the sequential circuit easier is the scan chain technique. This method offers more control to the internal states of the netlist and also increases the number of outputs, useful to observe the internal states. The basic concept behind the scan chain consist in create two modalities of work of the netlist, which are named "scan-mode" and "normal-mode". The flip-flops in the netlist are substituted with a scan flip-flops able to work with scan chains. The synthesis library usually has specific model of scan flip-flops, with the structure reported in the figure 2.22.

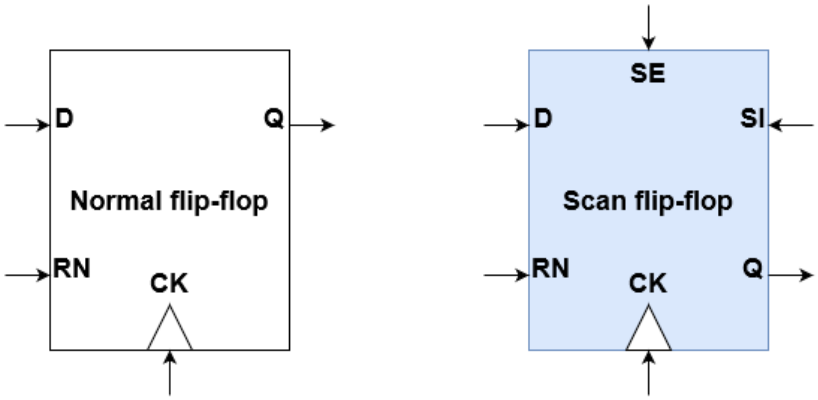


Figure 2.22: Scan flip-flop in comparison with a normal one

During the scan mode, the inputs for the scan flip-flops come from the SI port and not from the normal input D. If the scan flip-flops are connected together, during the scan mode they create a shift register by connecting the output Q of the previous register to the input SI of the following one. This allows loading in the flip-flops the desired values without passing throw the combinational logic before them. The input SE selects the working mode of the scan flip-flops, for example when is equal to 1 the scan mode is selected, and when 0 is the normal mode. During normal mode the input of the scan

register returns to be the normal one, for example D. According to these modes, the scan chain procedure to test the UUT is the following:

- The first step consists in selecting the scan mode, here for every clock cycle specific values are loaded into the scan-flip-flops. The normal inputs of the circuit do not need input values;
- The second step consists in the activation of the normal mode, here the scan flip-flops turn to behave normally. Test stimuli are applied to the normal inputs of the netlist. This step is made in one or more clock cycle. The normal outputs pins produce results after the elaboration of the input data and the values inside the scan registers;
- The third step consists in selecting the scan mode and scanning out the values produced during the precedent normal mode. At the same time, the values that need to be upload in the scan flip-flops are shifted in. The scan mode and normal mode are repeated until all the desired tests are made.

The normal flip-flops inside a netlist are substituted by a scan register usually by the same tools that perform the logic synthesis of the gate-level netlist. The number of scan chains in the netlist and the percentage of normal flip-flops substituted by the scan flip-flops is a choice of the tester engineer that wants to test the IC.
An example of scan chains techniques applied to a circuit is reported in figure 2.23.

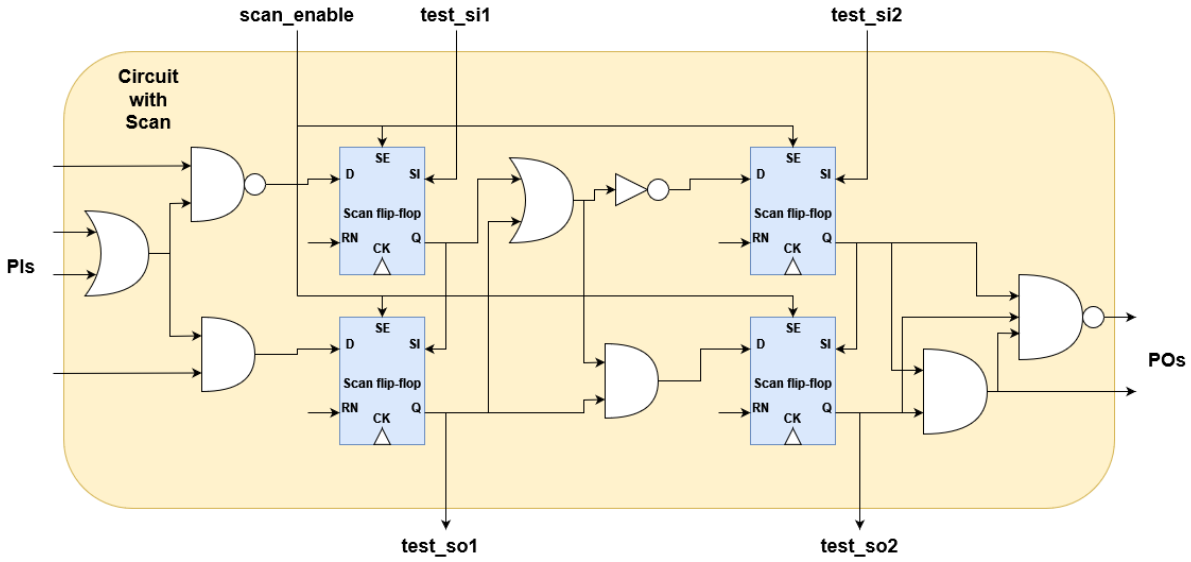


Figure 2.23: Example of netlist with scan chains

In this particular example, there were inserted 2 scan chains of the same length as 2 flip-flops. During the scan mode, the values are uploaded in the circuit using the 2 inputs of the chains, test_s1 and test_si2 and the precedent values from the precedent

normal mode are shifted out using the ports test_so1 and test_so2. During the normal mode, stimuli are applied to the PIs and the values of the outputs are recorded, to see a possible mismatch. The main risk that scan chains can introduce in an IC is overtesting i.e., marking good products as faulty.
Imagine that an IC implements a finite-state machine (FSM) that could assume three internal statuses. These are coded using 2 flip-flops. The total number of states that the registers could assume is 4. Using scan chains is possible to test all the states, but according to the behaviour of the netlist, only 3 are important. So all the possible faults connected to the unreachable state are less important.

3 Approach

This chapter presents a general approach for developing a testing flow able to work with CAT fault model and test any possible ICs and SOCs described at gate-level netlist. First, the structure of the proposed CAT flow is reported, then a summary of the gate-level netlist of an UTT is presented. After it is reported the approach to create the CAT fault model for every cell in a logic synthesis library. The resulting CAT defects are classified in two categories. The CAT fault list generator is presented, the different standards used to describe the test stimuli and the CAT fault simulator data flow. At last, the fault simulations results format is reported.

3.1 CAT proposed flow

The CAT proposed approach aims at creating a working fault simulation flow that, given a general IC or SoC gate-level netlist, is able to create a CAT fault model starting from the synthesis library used, then creates a CAT fault list and passes it to a fault simulator that applies the test stimuli and produces a final fault coverage. The main steps of the flow can be summarized as follows:

- CAT fault model development: this step starts from the transistor-level netlist of the cells present in the synthesis library. For every cell, the corresponding timing and power information are read. On the netlist of the cell, 2 main simulation types are performed. The first use the good version of the cell, where no defects are present. All the possible input stimuli are applied and the results are recorded. The second type of simulation works the faulty version of the cell netlist, where 1 single fault is injected at a time and all the stimuli applied on the good machine are simulated. The intra-cell faults that produce a mismatch on the outputs of the cell are catalogued as CAT faults and a model for every single cell in the library is produced. In particular, in this CAT fault model for every CAT fault is reported what input patterns are able to test it;

- CAT fault list generation: the CAT fault models created in the previous steps are used together with the UUT gate-level netlist to produce a CAT fault list. Starting from the netlist, every cell present inside it is taken into account. The CAT fault model of the specific cell is read, and the complete list of faults for the cell is created. This step is repeated for every instance of the cells in the gate-level netlist. At the end a fully CAT fault list is created;

- CAT fault simulation: in this last step, the fault simulator reads the gate-level netlist of the UUT, the CAT fault list created by the previous step and the test

stimuli to simulate. First, a simulation on the good machine is performed to check if the expected results of the test stimuli coincide with the simulated one. Then, every single fault in the CAT fault list creates a faulty machine. At last, the test patterns are applied and the results present the outputs are collected. If the injected faults produce a mismatch it is catalogued as detected by the test stimuli, otherwise, it is not covered by them.

In the figure 3.1 it is possible to see the data flow of the proposed CAT approach.

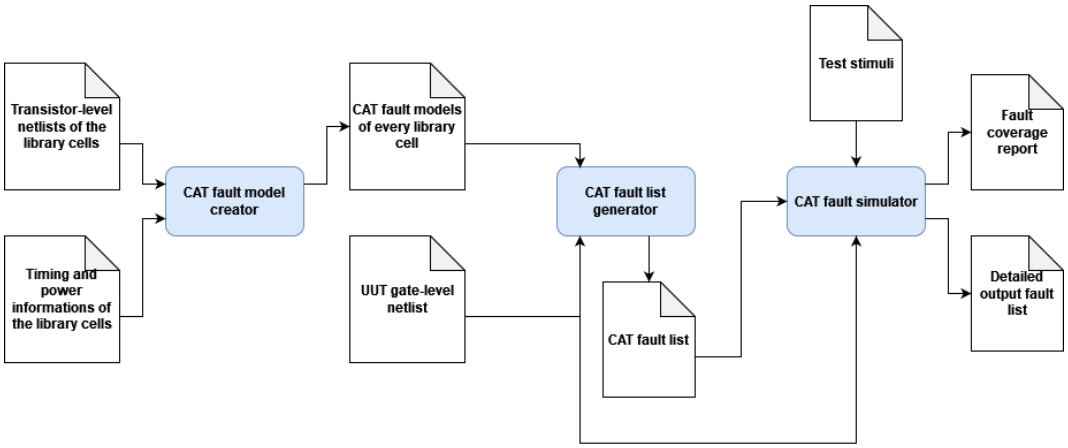


Figure 3.1: Proposed approach for a CAT flow

In the following section the main steps of the flow are reported.

3.2 Gate-level netlist of an IC and CAT faults

The gate-level netlist of an IC consists of interconnections of cells present in a synthesis library. An example is reported in the figure 3.2. Taking 4 different inputs A, B, C and D, it performs the logic AND between them. The output formula is equal to Z= ABCD. The structure of the netlist is very important for the faults that will be analyzed by the faults simulator. This concept can be deduced by comparing the two fault models SAF and CAT.

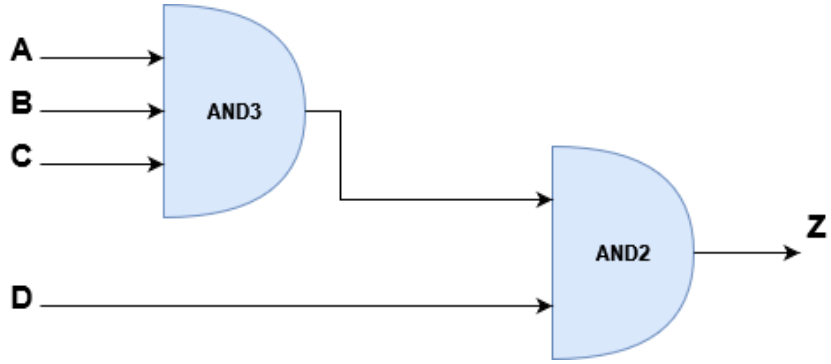


Figure 3.2: Logic synthesis examples

The first model applied on the netlist produce the faults reported in the figure 3.3. In the netlist, the total number of faults is equal to the number of interconnections multiplied by two. So there are 12 stuck-at in the circuit.

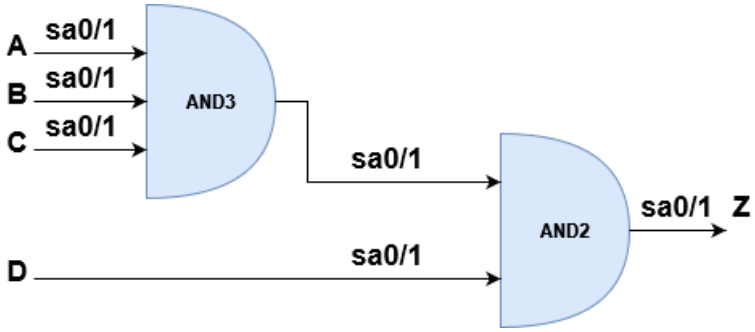


Figure 3.3: SAFs applied to the netlists

Regarding the objective model of this thesis, the CAT fault model, there are several ways in which the intra-cells defects of the cells could be represented. For example, the fault generator can read the internal structure of every cell of the synthesis library in the gate-level description of the UUT and create a list of faults with a name that is unique in all the netlist. There are many ways to implement a CAT flow able to test ICs. The proposed approach refers to a one-time characterization of the cells present in the synthesis library used to produce the gate-level netlist. For every element in the library, a file with the description of the CAT faults is produced. The identifiers that distinguish between the different CAT faults are unique inside a cell. Instead, outside the cells these identifiers are not unique, so they can be distinguished between each other by also referring to the cell name to which they belong. An example of this approach for the CAT faults is reported in the figure 3.4.

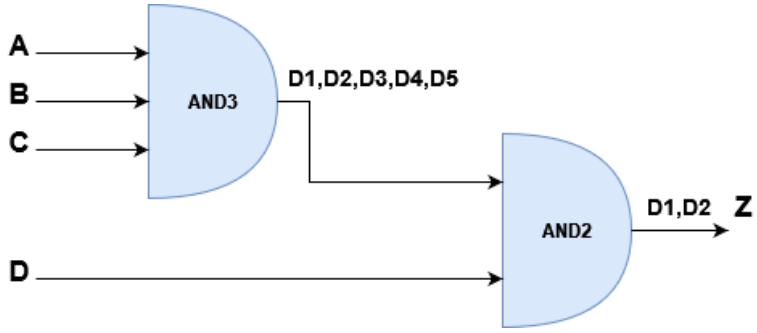


Figure 3.4: CAT faults applied to the netlists

Instead of depending on the number of internal interconnections, the CAT faults depend on the internal structure of the cells. The bigger the number of electronic elements inside a cell, like the transistors, the bigger the number of CAT faults. The AND3 has 3 inputs, so the internal structure is bigger than the AND2 one, implying that the number of CAT faults increases. As mentioned in the section 2.5.3, the CAT fault model does not consider the cells in the netlist as Black boxes, allowing to create of a more precise fault model with respect to SAF and TDF. The necessary analysis to create a CAT fault model for the UUT need to be performed at the transistor-level. The fault generator and the fault simulator and all the tools needed for the testing purpose usually work on a gate-level netlist. To resolve this problem, in the proposed approach, starting from the transistor-level of every cell of the library, a CAT fault model is created and saved in a file; it can be used with gate-level netlist and the usual testing tools. The following section describes how the CAT fault model of every cell is created.

3.3 CAT fault model creator

The following subsection explain how the CAT fault model is created for every cell in the synthesis library. Starting from the transistor-level netlist of a cell, some analyses are performed as reported in the subsection 3.3.1. The resulting CAT faults are split into two categories as explained in the subsection 3.3.2 and the CAT model essential format required for the CAT testing flow is created.

3.3.1 CAT fault analysis

For every cell inside the synthesis library, a CAT fault model is created. This model reports all the CAT faults present in the cell. The necessary inputs to create the CAT fault model are:

- The transistor-level netlist of the cell with all the possible defects that could affect it;
- The timing and power values that the cell respects during the normal operation.

A fault simulation at transistor-level is performed. The first step consists into the creation of all the possible input patterns that the cell can handle. Then a simulation of these patterns on the good circuit is performed. The values of the outputs are recorded and saved. These values are fundamental to understanding if the next simulations will produce misbehaviour. Then in the transistor-level netlist, a single defect is injected. The inputs patterns are applied to the faulty machine and the results are recorded. If they mismatch with the results of the good machine, the information about what patterns were able to produce observable misbehaviours is saved in the CAT fault model of the cell. This procedure is repeated for every defect in the cell. To better understand the fundamental steps to create the CAT fault model of a cell, an example is provided regarding the characterization of a simple cell. The cell is an inverter, that takes the input values of the input port and inverts the state on the output port. This cell could be implemented by using 2 transistors, one PMOS that connects the output port to the voltage supply V_{DD} and an NMOS that connects the output port to the ground V_{GND}. When the input voltage is high, the NMOS is active and connects V_{GND} to V_{OUT}, when the input voltage is low, the PMOS is active and connects V_{DD} to V_{OUT}. Figure 3.5 reports the transistor-level structure of the inverter.

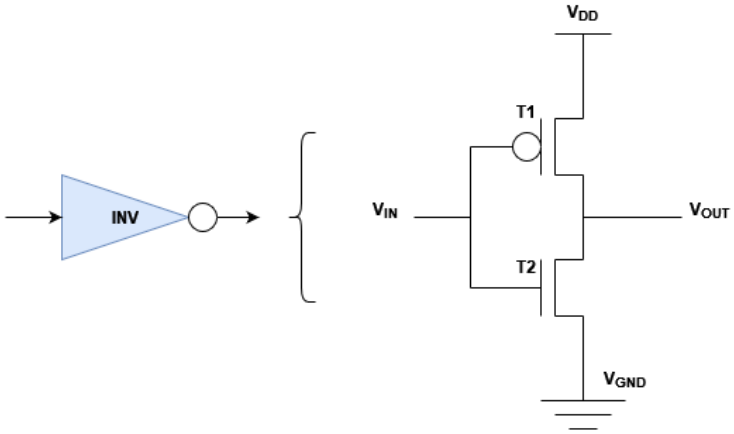


Figure 3.5: Transistor-level netlist of the inverter

The outputs of the inverter are normally produced after an expected delay, and the output values are always the opposite of the inputs values. In the second row of figure 3.6, the correct behavior of the inverter, given some input patterns, is reported. The third row represents dynamic misbehaviour active on the falling edge of the output and the fourth row represents static misbehaviour that does not allow the output to assume the low state.

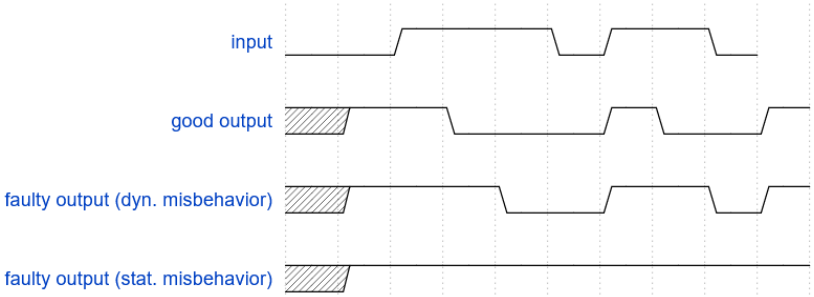


Figure 3.6: Good and faulty output waveforms of the inverter

The defects at transistor-level can be described with parasite resistors and capacitors inserted in different parts of the circuit. The first ones are useful to represent shorts and open inside the netlist. The shorts are described by a resistor with a low resistance (in Ohm) that allows the current to pass inside it. The opens are described by a resistor with high resistance (in Ohm) that allows the current to not pass inside it. The capacitors are useful to describe shorts and opens that create dynamic misbehaviours, this does not mean that static misbehaviours can not be described with capacitors, the same for resistances and dynamic misbehaviours. In the figure 3.7 the resistor R represents a port short defect between the output port V_{OUT} and the supply V_{DD}.

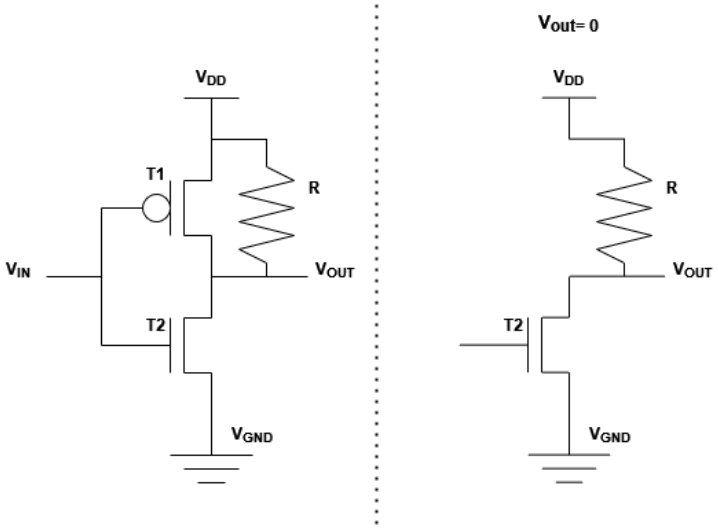


Figure 3.7: On the left, inverter netlist with a port short described with a resistor; on the right, the inverter circuit when the input V_{IN} is 1;

The fault simulator, that has to analyze this faulty machine, simulates the 2 possible input patterns. When to V_{IN} is applied to the low state, the output reaches the high state without delays. When V_{IN} is set to the high state, the output tries to reach the low state but due to the high value of the resistor R V_{OUT} does not behave correctly. The

expected final state is never reached and the final output voltage assumed depends on many factors. In figure 3.8 the black line represents the behaviour of the good machine during the simulation i.e., the inverter without any defect. The red line describes the behaviour of the faulty machine during the simulation i.e., the inverter with the resistor R.

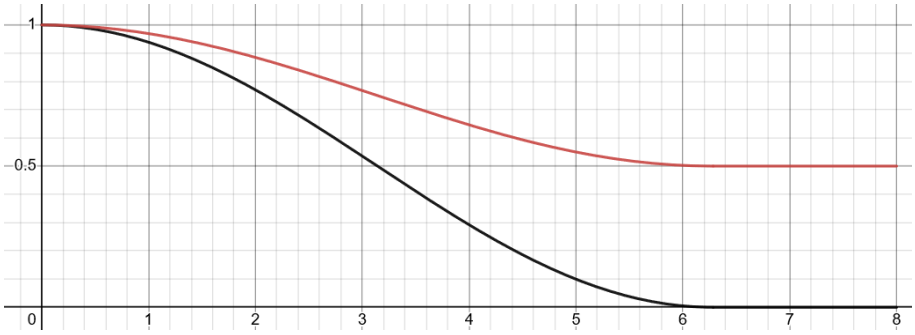


Figure 3.8: Simulation response from the good and the faulty inverter

This defect represents static misbehaviour and in this particular case, this CAT fault effect is equivalent to the one of a sa1. If in the transistor-level netlist the resistor R is substituted with a capacitor C, the behaviour of the faulty machine changes. The figure 3.9 reports the substitution of the resistor R with the capacitor C.

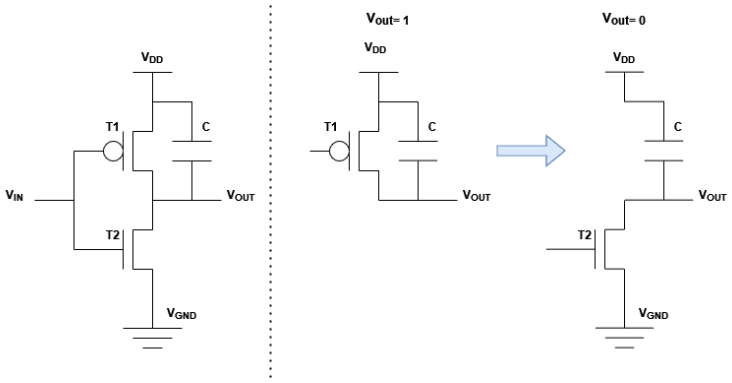


Figure 3.9: On the left, inverter netlist with a port short described with a capacitor; on the right, the inverter circuit when V_{IN} changes from 0 to 1;

The capacitor charges and discharges during the transitions of the output port. This defect is equivalent to dynamic misbehaviour, so in order to detect it, a couple of stimuli need to be simulated in dynamic conditions. By applying the input transition from the low state to the high state, the output response is slowed down by the presence of the capacitor C, with an abnormal high capacity value. In fact, during the transition, the capacitor behaves like a shortcut and allows the supply port V_{DD} to keep charging V_{OUT} while V_{GND} tries to discharge it. In the end, the capacitor behaves like an open and the output finally reaches the low state. In the figure 3.10 with the black line has reported

the behaviour of the good machine during the simulation i.e., the inverter without any defect. The red line describes the behaviour of the faulty machine during the simulation i.e., the inverter with the capacitor C. In this particular case, the CAT fault is equivalent to a slow-to-fall TDF.

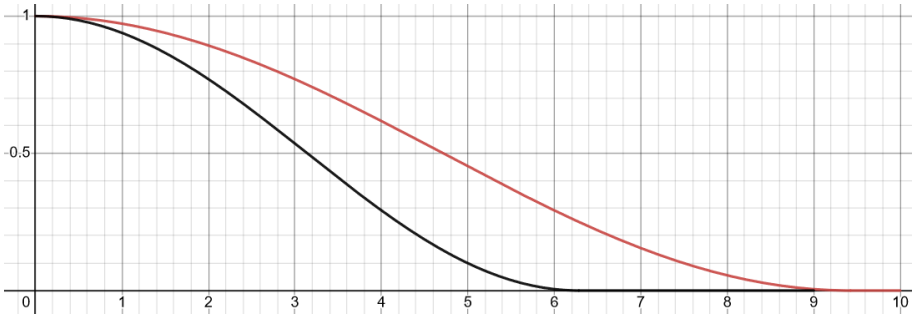


Figure 3.10: Simulation response from the good and the faulty inverter

3.3.2 CAT faults classification

The CAT faults are able to produce two different effects on the outputs of the cells of a synthesis library.
When the defect inside a cell produces static misbehaviour i.e., the desired output state is never reached, the CAT fault can be classified with the term static CAT (CAT_STAT) and can be compared with SAFs, activated with specific input combinations on the cell.
When the defect inside a cell produces dynamic misbehaviour i.e., the desired output state is reached after a large delay in comparison to the expected one, the CAT fault can be classified with the term dynamic CAT (CAT_DYN) and can be compared with TDFs.
The classification between these 2 categories strictly depends on the tool that performs the CAT fault analysis of the cells of the logic synthesis library.
In the previous section, the defect reported in the figure 3.6 is a CAT_STAT, instead the defect reported in the figure 3.9 is a CAT_DYN. Like SAFs, CAT_STATs need only one input pattern to be detected. Like TDFs, CAT_DYN need 2 input patterns where at least one inputs change its status. The data about these 2 categories and the information about the detection of the CAT faults are summarized in a list.

3.3.3 CAT faults essential format

In the proposed approach the information about the CAT fault model refers to every single cell present in the logic synthesis library. These data are saved in files that then are used by the fault simulator that will work with CAT faults.
In the section 2.5.3 was already shown how it is possible to represent and save the CAT faults information. Now, keep in mind the passage to analyze the defect inside a cell present in the section 3.3.1 and the proposed CAT faults classification present in the

section 3.3.2 it is possible to proceed with a CAT faults essential format that summarizes all the fundamental information about the fault model. A fault simulator that works at the gate level does not know the internal structure of the cells, so if a CAT fault is equivalent to a resistor or a capacitor it is not important, as the name of the element in the transistor-level netlist. So every CAT fault is associated an unique name (ID) that will be used in the fault simulations to univocally refer to it. Then, two detection tables need to be created, one for CAT_STAT faults and the second for the CAT_DYN faults. All the other possible information about the CAT faults is not fundamental and strictly depends on the implementation. Starting from a generic netlist of a library cell, the CAT faults essential format is the following. The figure 3.11 reports the transistor-level netlist of a NOR cell with 2 inputs A and B and 1 output ZN.

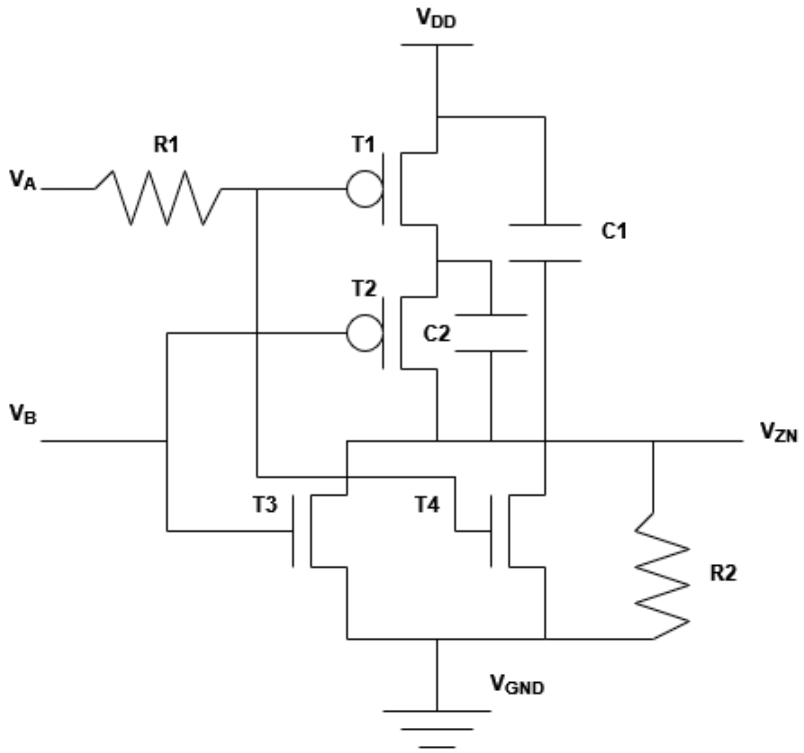


Figure 3.11: Transistor-level netlist of a NOR cell

After performing the CAT faults analysis and CAT faults classification, the following data are created:

- R1 is represented with the ID D1 and it is a CAT_STAT;
- R2 is represented with the ID D2 and it is a CAT_STAT;
- C1 is represented with the ID D3 and it is a CAT_DYN;
- C2 is represented with the ID D4 and it is a CAT_DYN.

The detection tables for the CAT faults are the following.

A	B	ZN	D1	D2
0	0	1	0	1
0	1	0	0	0
1	0	0	1	0
1	1	0	0	0

A	B	ZN	D3	D4
0	R	F	1	1
R	0	F	1	0
0	F	R	0	0
R	1	0	0	0
F	0	R	0	0
1	R	0	0	0
F	1	0	0	1
1	F	0	0	0

Table 3.1: CAT_STAT detection table on the left. CAT_DYN detection table on the right

This model is then used by the fault simulator to perform the simulations on the UUT. As explained in the section 2.5.3, in the CAT_DYN detection table the F describes a transition on the related pin from 1 to 0 and the R describe a transition on the related pin from 0 to 1.
In the inputs and outputs columns, the 0s and 1s indicate the state of the pins in the related patterns. In the defects columns, the 0s indicate that the related patterns are not able to test the CAT faults, and the 1s indicate that the related patterns are able to test the CAT faults.

3.3.4 CAT faults and SAFs detection comparison

Before proceeding with the second main step of the proposed approach it is necessary to spend few words about the difference between the SAFs and CAT faults. The needed analysis will be made on a simple full adder cell named FA with three inputs (A, B, CI) and two outputs (CO, S), shown in figure 3.12, where the SAFs are also reported.

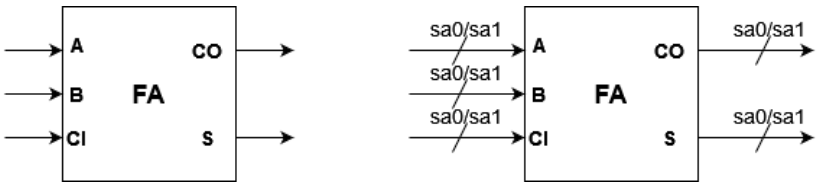


Figure 3.12: On the left the full adder cell, on the right the full adder cell with SAFs

These faults can be detected on only one port at a time and the effect is always the same. This means that when a sa0 is present, the port can only assume the zero value, instead, if a sa1 is present the port can only assume the one value. Moreover, the SAFs act only on one port at a time. This concept is not valid for CAT faults. For example, the cell has a CAT_STAT detection table like 3.2.

A	B	CI	CO	S	D1	D2	D3	D4	D5
0	0	0	0	0	1	2	2	2	2
0	0	1	0	1	0	2	1	2	2
0	1	0	0	1	0	0	0	1	1
0	1	1	1	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0

Table 3.2: FA CAT_STAT detection table

In the table the 1s indicate that the defects are detectable on the first output port CO, instead, the 2s indicate that the defects are detectable on the output port S.
The CAT_STAT fault D1 has the same effect as a sa1 on the port CO, but only when the input pattern A=0, B=0 and CI=0 is present. When this fault is excited, the output port CO instead of producing a zero will produce a 1.
The CAT_STAT fault D2 has both the effect of sa1 or sa0 on the output port S. In fact, it is detectable when S assumes both the value 0 and 1. During a test, if the pattern A=0, B=0 and CI=0 is present at the inputs, the CAT_STAT fault is exited and the output port S produce a 1 instead of a 0. If the pattern A=0, B=0 and CI=1 is present at the inputs, the CAT_STAT fault is exited and the output port S produce a 0 instead of a 1. This means that D2 have the same effect of a sa1 and sa0, depending from the input pattern.
The CAT_STAT fault D3 has the same effect as a sa1, but it is detectable on two different output ports, CO and S. So it has the behaviour of two different sa1 and it is excited by two patterns.
The CAT_STAT fault D4 has both the effect of sa1 and sa0 on the output port S and has the same effect of a sa1 on the port CO. So it has the behaviour of three different SAFs.
The CAT_STAT fault D5 has both the effect of sa1 and sa0 on the output port S and has both the same effect of a sa1 and sa0 on the port CO. So it has the behaviour of four different SAFs.
It is possible to conclude that the CAT faults can be classified in comparison to the SAFs based on where they can be detected (different output ports) and the effect on the outputs that they create when they are present (if in the specific input pattern they behave like sa0 or sa1). In chapter 4 the CAT faults are categorized according to these properties.

3.4 CAT fault list generator

The second main step in the CAT flow is the CAT fault list generation. Starting from the gate-level netlist of a UUT and using the CAT faults models of the cells present inside, a specific tool it is able to create a complete CAT fault list for the CAT fault simulator. The netlist of an IC consists of interconnections of cells present in the synthesis library.

The usual HDL used to describe the netlist is Verilog. The behaviour of the fault list generator depends on the structure and the standard of the HDL used to describe the netlist. Also, the output list produced depends on the tools used to fault simulate the CAT faults. This means that this step strictly depends on the specific implementation and the instruments used. In order to explain how a possible fault list generator could work, the proposed approach presents only the needed elements from the Verilog gate-level netlist; also the resulting output fault list will be formed by the CAT faults described in a general form that does not refer to a precise format needed by a specific tool.
The Verilog files resulting from the logic synthesis can be constituted by one single module that contains all the cells instantiated in the IC in a "flat" arrangement, or can be composed with a "top module" that contains all the submodules with a hierarchical structure, where the cells instances are the leaves. The figure 3.13 shown this concept.

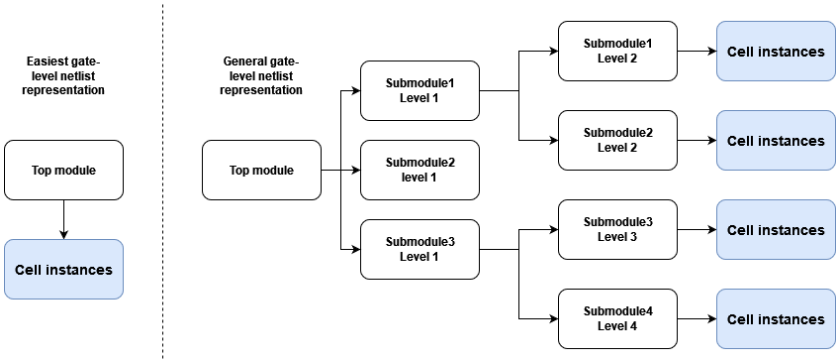


Figure 3.13: General structure of a gate-level netlist of an UUT

In the output fault list regarding the proposed approach, the main elements that need to be present to make a general CAT fault simulator work are:

- The unique ID to identify the CAT faults in the cell;
- The path to the cell instance in the gate-level netlist;
- The type of CAT faults (CAT_STAT or CAT_DYN).

To create these elements, the module structure needs to be read from the Verilog netlist. Referring to the example reported in the figure 3.14

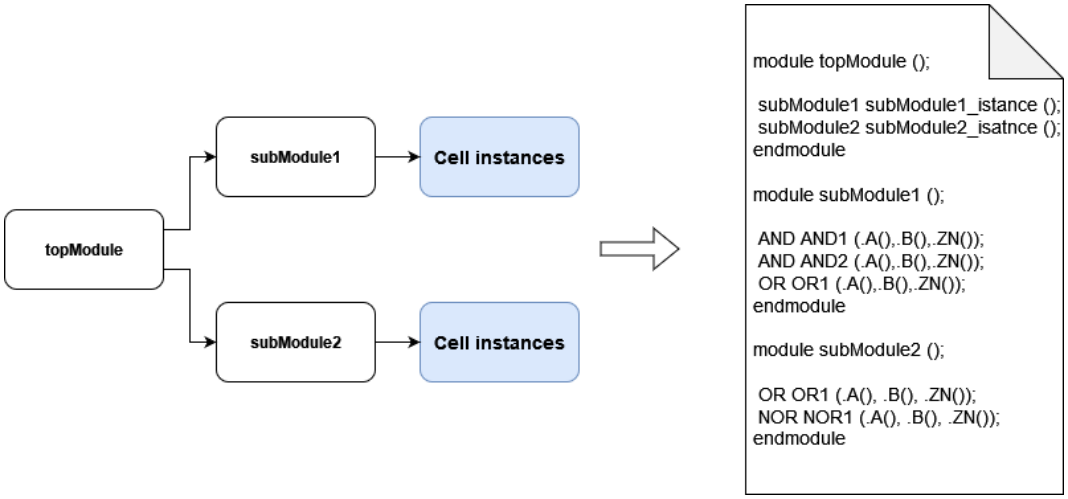


Figure 3.14: Module structure from the Verilog gate-level netlist

it is possible to see how in Verilog the cells inside a module are represented with 3 main fields. The first is the name of the cell to which it refers, the second is the instance name, which is unique only inside the module in which is present, and the third is made by the cell interface connection ports connected to the wires of the netlists. This information is used to create a unique identifier to describe a cell inside the gate-level netlist. Starting from the top module, the path through the sub-modules to the desired cell is built and it is made with the name of the instances. The ID to identify the CAT fault is taken from the CAT fault model like for the type of CAT fault. The unique ID to identify a CAT fault in the fault list is made by the combination of the ID and the path to the cell. A possible example regarding the netlist in the figure 3.14 can be the following one. The CAT fault model of the AND cell library presented in the logic library reports three CAT_STAT faults (D1, D2, D3) and two CAT_DYN faults (D4 and D5). The correct representation of the CAT defects in the fault list is:

- CAT_STAT subModule1_istance/AND1/ZN D1;
- CAT_STAT subModule1_istance/AND1/ZN D2;
- CAT_STAT subModule1_istance/AND1/ZN D3;
- CAT_DYN subModule1_istance/AND1/ZN D4;
- CAT_DYN subModule1_istance/AND1/ZN D5.

As already mentioned, this format of the CAT fault list is a general one that does not refer to specific tools that perform the CAT fault simulation. Despite that, this example mimics well the fundamental concepts used in the chapter 4 for the real implementation of a fault list generator.
The algorithm that creates the CAT faults list has the following main steps:

- For every cell in the gate-level netlist, create a unique path to it;
- For every fault in the CAT fault model, create an element in the final fault list with the path to the cell created previously, the ID of the CAT faults and the type (CAT_STAT or CAT_DYN);
- When the cell of the netlist has been all analyzed, create a final CAT fault list with all the faults.

The figure 3.15 reports the main steps of the algorithm.

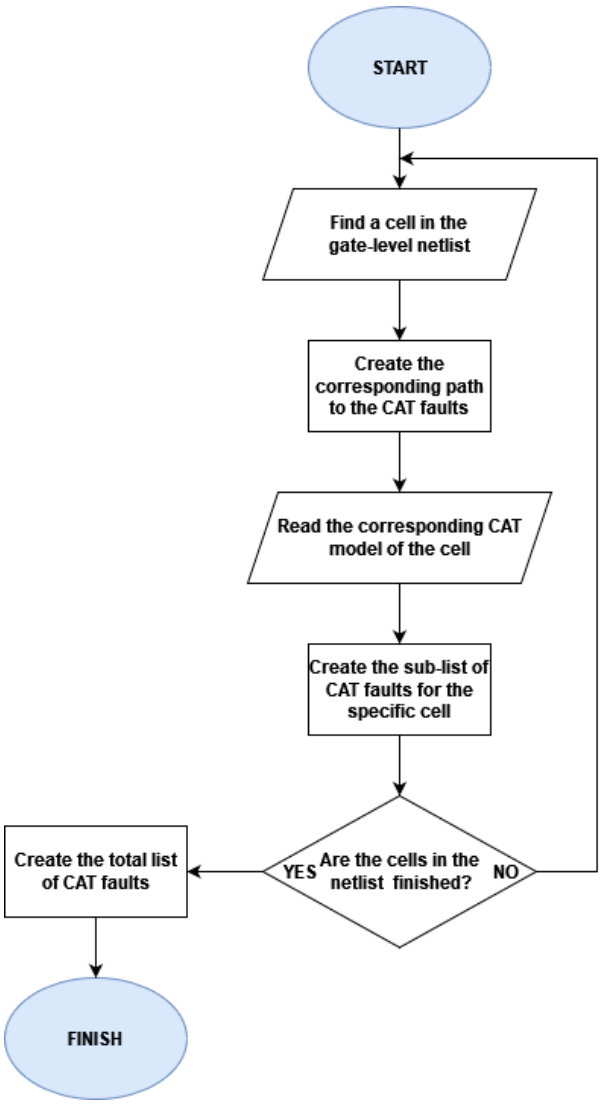


Figure 3.15: Algorithm for the CAT fault list generator

3.5 Test stimuli format

The last element of the proposed CAT flow, the CAT fault simulator, needs to be able to read and simulate the test stimuli of the UUT. The test patterns can be represented with different standards. In this thesis the 1450-1999 - IEEE Standard Test Interface Language (STIL) have been used, an ASCII-based format for dump files generated by EDA logic simulation tools named Value Change Dump (VCD) and Extended VCD (EVCD), very similar to the first version but with additional information like the direction of the signals.
According to the standard [12], the STIL files structure is divided into the following blocks:

- Signals block which contains the information about the pins of the UUT, in particular the pins names and the direction and type of the pins. For example, the pin direction could be In, Out, InOut, Supply etc...
- SignalGroups block which is used to create a simple group reference name to one or more pins of the UUT;
- Timing Block which defines the timing modalities for applying the test vectors to the signals. Inside a timing block that could be instantiated one or more WaveformTable blocks (Wfts), in which are defined a period and a set of events each with its own timing;
- DCLevels block which defines the voltages level to apply to the signals in each signals groups for every test vector;
- Pattern block which defines the data for the test pattern to simulate;
- PatternBurst block which specifies for the tests performed in the patterns sequence;
- PatternExec block in which are specified all the passage to perform the tests of the stimuli, for example inside it is specified the timing block to use.

An example of STIL file is reported in the figure 3.16.

```
Signals {
         input0 In; input1 In; input2 In; input3 In;
         output0 Out; output1 Out; output2 Out; output3 Out;
}
SignalGroups {
              inputSignals='input0 + input1 + input2 I+ input3';
              outputSignals='output0 + output1 + output2 + output3';
              ALL ='inputSignals + outputSignals';
}
Timing "basic_timing" {
       WaveformTable "one" {
         Period '500ns';
         Waveforms {
                    inputSignals { 01 { '10ns' D/U; }}
                    outputSignals { HLZ { '0ns' Z; '0ns' X; '260ns' H/L/T; '280ns' X;}}
                    }
        } // end WaveformTable one
} // end Timing "basic_timing"

PatternBurst "pat1_burst" {
    PatList { "pattern_1";
} // end PatternBurst "pat1_burst"

PatternExec {
    Timing "basic_timing";
    PatternBurst "pat1_burst";
} //end PatternExec

Pattern " pattern_1" {
      W "one";
      V { ALL=0000LLLL; }
      V { ALL=0010HLLL; }
      V { ALL=0001LHLL; }
      V { ALL=0100LLHL; }
      V { ALL=1000LLLH; }
      V { ALL=1100LLLL; }
      V { ALL=1110LLLL; }
      V { ALL=0011LLLL; }
      V { ALL=0101LLLL; }
Stop;
} // end Pattern " pattern_1"
```

Figure 3.16: STIL file example

The VCD format is constituted by 4 sections:

- Header section, specifying the name of the tool that creates the VCD and the timescale;
- Variable definition section, specifying the scope information expressed in a Verilog format with modules, tasks and functions. There are also specified the signals instances for every scope;
- $dumpvars section, specifying all the initials values of all variables dumped;
- Value change section, specifying all the changes for the signals in the VCD file. The changes are ordered according to the time in which they are collocated.

Figure 3.17 reports an example of VCD file. The eVCD format is similar to it.

```
$version
        Z01X (TM) Version: R-2020.12-SP2 (64-bit, May 19 2021) for Linux 64-bit
        +vcd+dumpvars
        +vcd+dumpfile+../VCDS/b02ScanZOIXVCD.dump.vcd
$end
$date
        2022-05-18 11:22:25 CET = 2022-05-18 09:22:25 UTC
$end
$comment
        Design: sim
        User:   i.guglielminetti
        Host:   buono Linux 4.4.165-81-default
$end
$timescale
 1 ps
$end
$scope module b02_Noscan_tb $end
$var wire 1 ! chain1 $end
$var wire 1 " test_si $end
$var wire 1 # out_u $end
$var reg 1 $ clk $end
$var reg 1 % reset $end
$var reg 15 & SEED [14:0] $end
$var reg 15 ' SEED2 [14:0] $end
$var reg 1 ( test_si_retarded $end
$var reg 1 ) test_se $end
```

Figure 3.17: VCD example

3.6 CAT fault simulator

The last part of the proposed CAT flow is the CAT fault simulator. The inputs needed to perform the fault simulation are the same that a normal fault simulator needs to work, as explained in the section 2.3. In particular, for the CAT fault simulator the input needed are:

- The gate-level netlist of the UUT;
- The CAT fault list produced by the CAT fault list generator with all the faults that need to be simulated;
- The test stimuli to simulate on the UUT. In particular, the CAT fault simulator has to work perfectly with the functional test patterns because they are the main focus of this thesis.

The most important steps of the CAT fault simulator are summarized in the figure 3.18.

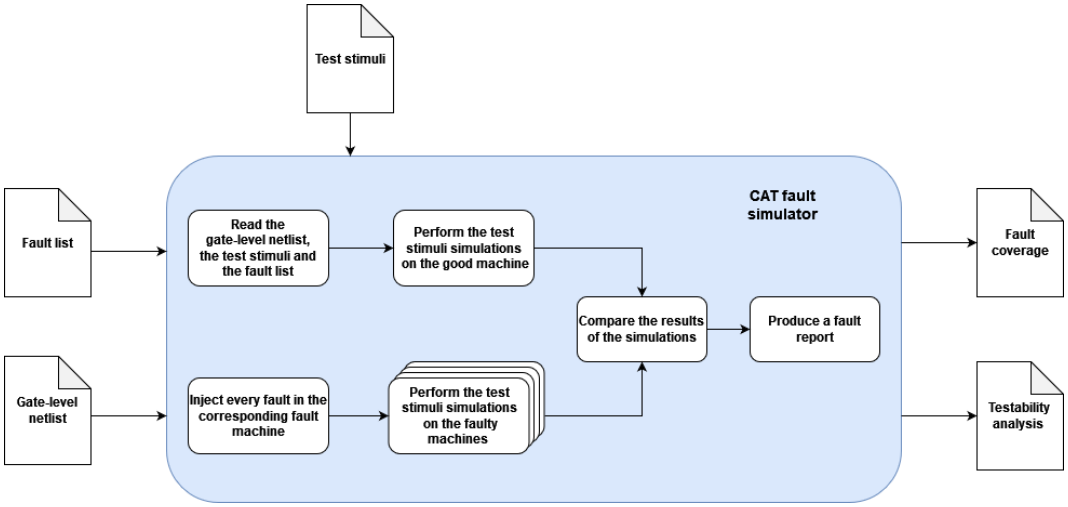


Figure 3.18: CAT fault simulator main steps

First, the gate-level netlist of the UUT and the test stimuli are read. Then the test patterns are applied to the netlist and the outputs results are checked with the expected values contained in the test stimuli. This is a good simulation of the UUT. Then the CAT fault list is read. The fault simulator takes one CAT defect at a time and injects it into the UUT, creating a faulty machine. The test patterns are simulated on the faulty machine and if at least a mismatch is observable on the outputs the CAT fault is detected. If no mismatches are observable, the fault is not detected. A faulty machine is created for every CAT fault in the input list, if they are simulated one by one or at the same time, using one process or in parallel depend on the tool used. The two types of CAT faults need to be fault simulated with different methods:

- The CAT_STAT faults do not need to be fault simulated at speed. This means that the clock speed of the UUT, while the test stimuli are applied on the faulty machine, does not need to be the same as the normal working frequency; the test modalities are similar to the ones applied for SAFs;
- The CAT_DYN faults need to be fault simulated at speed. The clock speed of the faulty machine needs to be equal to the normal working frequency of the UUT. In particular, for every test applied a couple of inputs stimuli create transitions on the outputs. The test modalities are similar to the ones applied for TDFs. Note that TDF fault simulation could be also not made at speed.

3.6.1 CAT results

The test coverage is the sum of the detected faults and the potential detected faults over the testable faults. The fault coverage is a sum of the detected faults and the potential detected faults over the total number of faults present in the UUT. The precise formulas

to calculate the test coverage and fault coverage depends by the tool used to perform the CAT fault simulation.

4 Implementation

This chapter presents the specific implementation of the testing flow able to work with CAT fault model and test any possible ICs and SOCs described at the gate-level netlist. First the tool used is reported to create the CAT fault model for every cell in the logic synthesis library. Then three specific examples of cells characterization are presented. After that, the CAT flow needed by the fault simulator Z01X to work correctly is explained. In particular, due to the rules of representation of the CAT faults in Z01X, the fault simulations for developing and validating the selected format for the CAT fault list are presented. Then, the CAT fault list generator developed for Z01X using python and bash scripts is explained. Finally, how to correctly read the fault simulation results from Z01X is presented.

4.1 Generation of CAT fault model for a synthesis library using CMGen

In section 3.3 the CAT fault model creator was introduced. This step of the CAT developed flow is implemented with the tool CMGen, developed by Synopsys. This software was developed with the specific purpose of creating the CAT fault model used by Synopsys ATPG and fault simulators like TMAX and Z01X. The version of CMGen used for this thesis is S-2021.06-SP4. The input files needed by this tool to create a CAT fault model for a cell of a library are reported in figure 4.1

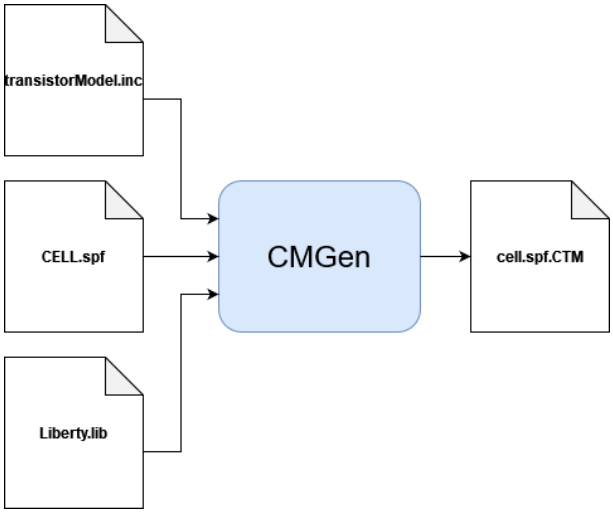


Figure 4.1: CMGen flow for create a CAT fault model for a library cell

The first fundamental input file is the "cell.spf", representing the spice module of the cell. The spice module offers a description at transistor-level of the cell. The module is defined inside the ".SUBCKT" section. Here the input ports and output ports of the cell that allow connections and communications with the outside of the cell are defined. Inside the module, electronic elements are defined and connected together using nodes, whose names are unique inside a cell but they can be reused outside it. Figure 4.2 reports an example of spf file from the nangate library. This example refers to the AND2_X1 cell.

```
.SUBCKT AND2_X1 VDD VSS A1 A2 ZN
*.PININFO VDD:P VSS:G A1:I A2:I ZN:O
*.EQN ZN=(A1 * A2)
M_M3 N_3_M0_d N_A1_M0_g N_VDD_M0_s VDD PMOS_VTL W=0.315000U L=0.050000U
M_M4 N_VDD_M1_d N_A2_M1_g N_3_M0_d VDD PMOS_VTL W=0.315000U L=0.050000U
M_M5 N_ZN_M2_d N_3_M2_g N_VDD_M1_d VDD PMOS_VTL W=0.630000U L=0.050000U
M_M0 net_0 N_A1_M3_g N_3_M3_s VSS NMOS_VTL W=0.210000U L=0.050000U
M_M1 N_VSS_M4_d N_A2_M4_g net_0 VSS NMOS_VTL W=0.210000U L=0.050000U
M_M2 N_ZN_M5_d N_3_M5_g N_VSS_M4_d VSS NMOS_VTL W=0.415000U L=0.050000U
C_x_PM_AND2_X1%VDD_c0 x_PM_AND2_X1%VDD_39 VSS 4.06218e-17
C_x_PM_AND2_X1%VDD_c1 x_PM_AND2_X1%VDD_36 VSS 2.74884e-18
C_x_PM_AND2_X1%VDD_c2 x_PM_AND2_X1%VDD_26 VSS 2.61603e-16
C_x_PM_AND2_X1%VDD_c3 N_VDD_M1_d VSS 7.05698e-17
C_x_PM_AND2_X1%VDD_c4 x_PM_AND2_X1%VDD_19 VSS 1.89932e-17
C_x_PM_AND2_X1%VDD_c5 x_PM_AND2_X1%VDD_18 VSS 3.50788e-17
C_x_PM_AND2_X1%VDD_c6 N_VDD_M0_s VSS 3.64083e-17
C_x_PM_AND2_X1%VDD_c7 x_PM_AND2_X1%VDD_13 VSS 1.92462e-17
C_x_PM_AND2_X1%VDD_c8 x_PM_AND2_X1%VDD_12 VSS 2.334e-16
C_x_PM_AND2_X1%VDD_c9 x_PM_AND2_X1%VDD_9 VSS 5.77955e-16
R_x_PM_AND2_X1%VDD_r10 VDD x_PM_AND2_X1%VDD_39 0.13879
R_x_PM_AND2_X1%VDD_r11 VDD x_PM_AND2_X1%VDD_35 0.392137
R_x_PM_AND2_X1%VDD_r12 VDD x_PM_AND2_X1%VDD_26 0.13879
R_x_PM_AND2_X1%VDD_r13 x_PM_AND2_X1%VDD_26 VDD 3.84471
R_x_PM_AND2_X1%VDD_r14 x_PM_AND2_X1%VDD_25 x_PM_AND2_X1%VDD_36 0.0731438
```

Figure 4.2: SPF file example

In the cell subcircuit, there are 3 fundamental elements: the resistors, the capacitors and the MOSFETs. The resistors are identified with an R followed by a string that creates a unique name inside a cell. This element has two terminals that can be connected to two nodes specifying their names. The last element needed by the resistor is the value of the resistance in Ohm.
The capacitors are identified with a C followed by strings that creates unique names inside a cell. Each of such elements has two terminals that can be connected to two nodes specifying their names. The last elements needed by the capacitors are the values or the capacities in Farad. The MOSFETs are identified with an M followed by a string that creates a unique name inside a cell. Then there is specified the name of the node connected to the drain, the name of the node connected to the gate, the name of the node connected to the source and the name of the node connected to the bulk. The MOSFETs parameters are usually expressed in .MODEL statement inside other files. This allows the separation of the internal structure of the cell to the description of the transistors that are usually made with many electronic parameters. In the proposed example there are specified 2 models: PMOS_VTL and NMOS_VTL. The last 2 elements W and L are the channel width (in meters) and the Channel length (in meters). The 2 models are defined in the second input file, the "transistorModel.inc". In figure 4.3 it is possible to see the definition of the different parameters used for modeling the transistors.

```
.model  NMOS_VTL  nmos ( level = 54

* parameters related to the technology node
+tnom = 27         epsrox = 3.9
+eta0 = 0.006     nfactor = 2.1    wint = 5e-09
+cgso = 1.1e-10    cgdo = 1.1e-10    xl = -2e-08

* parameters customized by the user
+toxe = 1.14e-09    toxp = 1e-09    toxm = 1.14e-09    toxref = 1.14e-09
+dtox = 0.14e-09    lint = 3.75e-09
+vth0 = 0.322       k1 = 0.4   u0 = 0.045   vsat = 148000
+rdsw = 155        ndep = 3.4e+18    xj = 1.98e-08



+version = 4.0            binunit = 1             paramchk= 1             mobmod  = 0
+capmod  = 2              igcmod  = 1             igbmod  = 1             geomod  = 1
+diomod  = 1              rdsmod  = 0             rbodymod= 1             rgatemod= 1
+permod  = 1              acnqsmod= 0             trnqsmod= 0

+ll      = 0              wl      = 0             lln     = 1             wln     = 1
+lw      = 0              ww      = 0             lwn     = 1             wwn     = 1
+lwl     = 0              wwl     = 0             xpart   = 0

+k2      = 0              k3      = 0
+k3b     = 0              w0      = 2.5e-006      dvt0    = 1             dvt1    = 2
+dvt2    = 0              dvt0w   = 0             dvt1w   = 0             dvt2w   = 0
+dsub    = 0.1            minv    = 0.05          voffl   = 0             dvtp0   = 1e-010
+dvtp1   = 0.1            lpe0    = 0             lpeb    = 0
+ngate   = 3e+20          nsd     = 2e+020        phin    = 0
+cdsc    = 0              cdscb   = 0             cdscd   = 0             cit     = 0
```

Figure 4.3: Parameters related to the MOSFETs models

The last file needed by CMGen to create the CAT model is the liberty file of the library. Here the timing and the power of every cell are specified. An example of a section of this file is reported in figure 4.4.

```
/*****************************************************************************************
   Module               : AND2_X1
   Cell Description     : Combinational cell (AND2_X1) with drive strength X1
  *****************************************************************************************/

  cell (AND2_X1) {

        drive_strength          : 1;

        area                    : 1.064000;
        pg_pin(VDD) {
                voltage_name : VDD;
                pg_type      : primary_power;
        }
        pg_pin(VSS) {
                voltage_name : VSS;
                pg_type      : primary_ground;
        }


        cell_leakage_power      : 25.066064;

        leakage_power () {
                when            : "!A1 & !A2";
                value           : 20.324370;
        }
        leakage_power () {
                when            : "!A1 & A2";
                value           : 30.850688;
        }
```

Figure 4.4: Section of the liberty file

The output of CMGen is the CAT fault model of the cells. The file has a specific format that is presented in the section 4.1.2.
Section 4.1.1 summarizes the linux environment parameters needed for running the CMGen tool.

4.1.1 Environment configuration for CMGen

CMGen to correctly work need that the user sets three environment variables:

- PYTHONPATH=:/path/to/python/modified/lib64/python2.7:/usr/lib64/python2.7/lib-dynload
- SYNOPSYS=/path/to/hspice/HSPICE_2014.09-SP1-1/hspice
- SYNOPSYS_TMAX=/path/to/TMAX/TMAX_2021.06-SP4/

PYTHONPATH is used to specify the desired version of python to use.
The variable SYNOPSYS is used to specify the desired version of HSPICE, which is an analogue circuit simulator that need to be present in the environment.
SYNOPSYS_TMAX is used to specify the desired version of TMAX.

4.1.2 CTM file format

The CTM file produced by CMGen contains the list of CAT defects of each cell, which could be undetected, detected statically or detected by a specific combination of input values. All the details about the CTM file format were taken from the CMGen user guide [6]. Figure 4.5 reports an example of a defect in the CTM file of the cell AND2_X1 of the nangate library.

```
- Id: D4
  Type:  short
  Description:  C_x_PM_AND2_X1%VDD_c3 N_VDD_M1_d VSS 1  status=static_det
  Instance: C_x_PM_AND2_X1%VDD_c3
  Attributes:
  - Class: DT
  - Behavior: stuck
```

Figure 4.5: Example of defect of AND2_X1

The mandatory section of the defect representation are:

- Id, in this case, D4, each defect has a unique one;
- Type, that represents the type of CAT faults, that can be short, open, transistor-short, transistor-open etc...;
- Description, regarding the defect location, size (resistor values), and detection status;
- Instance, the name of the defect shows which SPICE element is associated with the defect. In particular, in this case, D4 is associated with a capacitor that could be also seen in figure 4.2 ;
- Attributes, that refers to a defect, contain the equivalent defect ID, the defect behaviour (stuck, delay or small-delay), and the detection class i.e., if the defect is detected (DT), undetected (UD) and potentially detected (PD).

Regarding the attributes section, a defect will have the -DetEquivalent field when it can be detected in the same way as the prime defect, i.e, the defect in the static and dynamic detection tables to which it refers. The concept of the static and dynamic detection tables will be explained below. For better explaining the -DetEquivalent concept, in figure 4.6 the defect D30 can be detected in the same way as the defect D4, so they are equivalent.

```
- Id: D30
  Type:  short
  Description:  C_x_PM_AND2_X1%A1_c0 x_PM_AND2_X1%A1_14 VSS 1  status=static_det
  Instance: C_x_PM_AND2_X1%A1_c0
  Attributes:
  - Class: DT
  - DetEquivalent: D4
  - Behavior: stuck
```

Figure 4.6: Example of equivalent defect of AND2_X1

Regarding the field Class in attributes, it can assume 3 values, detected (DT), undetected (UD) and potentially detected (PD).
Regarding the description section, as we can see in figure 4.5, the CAT defects that can be detected statically belong to only just one type. The “status” section in the description could assume four values:

- static_det, the first CAT type, explained below, with associated class DT;
- dynamic_det, the second CAT type, explained below, with associated class DT;
- undet, when is impossible to detect the defect on the outputs, with associated class UD;
- det, when the simulation detected the fault, but the fault simulation fails at detecting it, so it can be see as an undet, in fact, the associated class is UD.

As already explained in section 3.3.2, CAT faults can be classified into two different typologies, CAT_STAT and CAT_DYN. According to the proposed implementation and the CMGen user guide [6], a CAT defect is assumed to be CAT_STAT if it causes a 10% variation in the analogue voltage value between the good machine and the faulty one. A CAT defect is assumed to be a CAT_DYN when in the faulty machine the output voltage can reach the expected value but with a delay with respect to the good one.
After the list of CAT defects, the CTM file has 2 detection tables, one for the CAT_STAT and one for the CAT_DYN. Here, the input patterns are reported with the expected value of the output and a column for every defect. The table values depend on the fact that a cell has multiple outputs or not. Let us start with the most simple example possible, the AND2_X1 cell. This is a combinational cell, so it does not have a clock input. It has 2 inputs, A1 and A2, and 1 output ZN. Figure 4.7 reports the detection table generated by CMGen for the CAT_STAT and CAT_DYN. The format of these two tables is the same presented in section 3.3.3. The inputs and the output can assume 2 values, 0 and 1. The defects columns are constituted by 0s and 1s. The 0 indicates that the pattern in that particular row is not able to detect the defect. The 1 indicates that it is able to detect the fault.

```
- [Table, Static]
- [A1,A2, ZN, D4,D7,D20]
- [0,  0,  0,  0, 0,  1]
- [0,  1,  0,  0, 1,  1]
- [1,  0,  0,  0, 0,  1]
- [1,  1,  1,  1, 0,  0]
- [Table, Dynamic]
- [A1,A2,ZN,D1,D2,D5,D41,D43,D46]
- [0,  R, 0, 0, 0, 0,  0,  1,  0]
- [R,  0, 0, 0, 0, 0,  0,  1,  0]
- [0,  F, 0, 0, 0, 0,  0,  1,  0]
- [R,  1, R, 0, 1, 0,  0,  0,  1]
- [F,  0, 0, 0, 0, 0,  0,  1,  0]
- [1,  R, R, 0, 1, 0,  0,  0,  1]
- [F,  1, F, 0, 1, 1,  1,  1,  0]
- [1,  F, F, 1, 1, 1,  0,  1,  0]
```

Figure 4.7: Detection tables of AND2_X1

In the AND2_X1 detection table for CAT_STAT, defect D4 is detectable by the last pattern and when ZN is equal to one. So when D4 is present it causes the output to be 0.
In the AND2_X1 detection table for CAT_DYN, the defect D1 is detectable when A1 is stable to 1, a fall transition is applied to A2 and on ZN a falling transition is observable. D1 causes a slow-to-fall on the output ZN.
CMGen can create the CTM file of multi-outputs cells. To understand how it does that, the static detection table of the cell FA_X1, a combinational cell with 2 outputs in the nangate library, is reported in figure 4.8. The cell is a full adder, it has 3 inputs, A, B, and CI and 2 outputs CO and S.

```
- [Table, Static]
- [A, B,CI,CO, S,D2,D3,D4,D5,D6,D7,D13,D14,D16,D18,D42,D54,D55,D57,D58,D59,D63,D66,D70,D75,D76,D77,D82,D84,D89,D92,D96,D98,D102,D103,D104,D108]
- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  1]
- [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,  0,  0,  0,  1,  3,  0,  1,  0,  1,  0,  1,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
- [0, 1, 0, 0, 1, 1, 0, 1, 3, 0, 0,  0,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0]
- [0, 1, 1, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  0,  0,  3,  3,  0,  0,  3,  0,  1,  1]
- [1, 0, 0, 0, 1, 1, 0, 0, 3, 0, 0,  1,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  0,  3,  0,  0,  1,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0]
- [1, 0, 1, 1, 0, 0, 0, 1, 0, 2, 0,  0,  1,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  0,  0,  0,  1,  1]
- [1, 1, 0, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  3,  3,  0,  3,  0,  0,  1,  1]
- [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2,  2,  2,  2,  2,  0,  1,  1,  0,  1,  1,  0,  0,  0,  1,  1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0]
```

Figure 4.8: Static detection table of FA_X1

As shown, the defects columns can assume the values 0,1,2 and 3. According to CMGen user guide:

- 1, the defect is detectable on the first output port, in this case, CO;
- 2, the defect is detectable on the second output port, in this case, S;

- 3, the defect is detectable on both output ports.

Finally, the detection table of a flip-flop is reported. The cell taken into account is named DFFR_X1 and is part of the nangate library. In this flip-flop, there are 3 inputs, CK (clock), RN (reset) and D and 2 outputs Q and QN, where QN is the opposite of Q. The static detection table is reported in figure 4.9.

```
- [Table, Static]
- [CK,RN, D, Q,QN,Q-,QN-, D1,D2,D4,D5,D6,D14,D22,D52,D131,D173,D174]
- [P,  1, 0, 0, 1, 0,  1,  1, 1, 1, 1, 0,  3,  3,  0,   0,   1,   0]
- [P,  1, 1, 1, 0, 0,  1,  2, 3, 3, 3, 3,  3,  0,  3,   0,   0,   2]
- [P,  1, 1, 1, 0, 1,  0,  2, 2, 2, 3, 2,  2,  0,  3,   3,   0,   2]
- [P,  1, 0, 0, 1, 1,  0,  1, 3, 2, 0, 3,  3,  3,  0,   0,   1,   0]
```

Figure 4.9: Static detection table of DFFR_X1

According to CMGen user guide, 'P' represents the active state of the clock CK and '0' is representative of the clock off state. Also, the 2 outputs are duplicated, in fact, in the static table there are Q- and QN-, which represent the previous state value of the outputs pins. These additional pins appear only in register cells.
The main purpose of this thesis is to develop and evaluate a flow to test ICs by means of functional test programs and the CAT fault model. To achieve that, Z01X was used. The Z01X fault simulator to work correctly with CAT faults needs in inputs a CAT faults list with the structure reported in the section 4.3. To develop it tests were made on the nangate cells XNOR2_X1, FA_X1 and DFFR_X1. The first represents a generic combinational cell with 1 output and the CAT model created by CMGen is reported in the section 4.1.3. The second represents a generic combinational cell with 2 outputs and the CAT model created by CMGen is reported in the section 4.1.4. The third represents a generic sequential cell with 2 outputs and the CAT model created by CMGen is reported in the section 4.1.5.

4.1.3 XNOR2_X1 CAT model

This is the first CAT model created by CMGen presented with a closer and complete look. This cell is a combinational one, it has 2 inputs, A and B, and one output ZN. The total number of CAT defects present in the CTM file is 146. The CAT_STATs are 41, the CAT_DYNs are 67, so there are 38 undetected defects. The detection tables for the CAT_STAT and CAT_DYN are reported in figure 4.10.

```
- [Table, Static]
- [A, B, ZN, D1,D6,D21,D32,D34,D35,D42]
- [0, 0,  1,  1, 1,  0,  0,  0,  0,  0]
- [0, 1,  0,  0, 0,  1,  0,  0,  0,  1]
- [1, 0,  0,  0, 0,  1,  1,  0,  1,  0]
- [1, 1,  1,  0, 1,  0,  0,  1,  1,  0]
- [Table, Dynamic]
- [A, B, ZN, D3,D57,D59,D60,D65,D67,D74,D75,D76,D111,D129,D143]
- [0, R,  F,  0,  1,  0,  1,  0,  0,  1,  1,  0,   0,   0,   0]
- [R, 0,  F,  0,  1,  0,  0,  1,  1,  0,  1,  0,   1,   0,   0]
- [0, F,  R,  1,  0,  0,  0,  1,  0,  0,  0,  1,   1,   0,   1]
- [R, 1,  R,  0,  0,  1,  0,  0,  0,  0,  1,  1,   1,   0,   0]
- [F, 0,  R,  1,  0,  0,  0,  0,  0,  1,  0,  1,   0,   0,   1]
- [1, R,  R,  0,  0,  1,  0,  0,  0,  0,  1,  1,   1,   0,   0]
- [F, 1,  F,  1,  1,  0,  1,  0,  0,  1,  0,  0,   0,   1,   0]
- [1, F,  F,  1,  1,  0,  0,  1,  1,  0,  0,  0,   1,   0,   0]
```

Figure 4.10: XNOR2_X1 static and dinamic tables

These are only the prime CAT faults, the others equivalent static defects are:

- D2, D5, D8 equivalent to D1;
- D52, D53, D54, D55, D56 equivalent to D6;
- D22, D23, D24, D25, D26, D27, D28, D29, D30, D31 equivalent to D21;
- D33, D36, D37, D38, D39 equivalent to D32;
- D40, D41, D43, D49, D50, D51 equivalent to D34;
- D44, D45, D46, D47, D48 equivalent to D42.

The other equivalent dynamic defects are:

- D7 equivalent to D3;
- D58, D61, D63, D64, D73, D81, D85, D86, D102, D128, D137 equivalent to D57;
- D62, D69, D71, D72, D77, D78, D98, D99, D100, D101, D109, D110, D115, D121, D122, D123, D124, D126, D127, D133, D138 equivalent to D59;
- D80, D96 equivalent to D60;
- D66, D68, D107, D108, D112, D113, D114 equivalent to D65;
- D70, D105, D106 equivalent to D67;
- D79, D92, D93, D94, D95, D97 equivalent to D74;
- D116, D117, D118, D119 equivalent to D75.

4.1.4 FA_X1 CAT model

This is the second CAT model created by CMGen presented with a closer and complete look. This cell is a combinational one, it has 3 inputs, A, B and CI and 2 outputs CO and S. The total number of CAT defects present in the CTM file is 309. The CAT_STATs are 88, and the CAT_DYNs are 151, so there are 70 undetected defects. The detection table for the CAT_STAT is reported in the figure 4.11.

```
- [Table, Static]
- [A, B,CI,CO, S,D2,D3,D4,D5,D6,D7,D13,D14,D16,D18,D42,D54,D55,D57,D58,D59,D63,D66,D70,D75,D76,D77,D82,D84,D89,D92,D96,D98,D102,D103,D104,D108]
- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  1]
- [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,  0,  0,  0,  1,  3,  0,  1,  0,  1,  0,  1,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
- [0, 1, 0, 0, 1, 1, 0, 1, 3, 0, 0,  0,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0]
- [0, 1, 1, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  0,  0,  3,  3,  0,  0,  3,  0,  1,  1]
- [1, 0, 0, 0, 1, 1, 0, 0, 3, 0, 0,  1,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  0,  3,  0,  0,  1,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0]
- [1, 0, 1, 1, 0, 0, 0, 1, 0, 2, 0,  0,  1,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  0,  0,  0,  1,  1]
- [1, 1, 0, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  3,  3,  0,  3,  0,  0,  1,  1]
- [1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,  2,  2,  2,  2,  2,  0,  1,  1,  0,  1,  1,  0,  0,  0,  1,  1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0]
```

Figure 4.11: FA_X1 static table

The CAT_DYN table is too large to be reported here. The table columns are only the prime CAT faults, the other equivalent static defects are:

- D10, D11, D116 equivalent to D2;
- D12 equivalent to D3;
- D15 equivalent to D4;
- D17, equivalent to D5;
- D19, D41, equivalent to D6;
- D43, D44, D45, D46, D47, D48, D49, D50, D51, D52, D53 equivalent to D42;
- D61, D62, D74, D80, D81, D88, D94, D95 equivalent to D54;
- D56 equivalent to D55;
- D64, D65 equivalent to D57;
- D67, D68, D69 equivalent to D66;
- D71, D72, D73 equivalent to D70;
- D78, D83, D85, D86 equivalent to D77;
- D90, D91, D97 equivalent to D89;
- D99, D100, D101 equivalent to D92;
- D105, D106, D107 equivalent to D104;
- D109, D110, D111, D112, D113, D114, D115 equivalent to D108.

The other equivalent dynamic defects are:

- D202, D205, D206, D232, D237, D239, D240, D241, D257, D300, D301 equivalent to D117;
- D121, D122, D127, D128, D291, D296, D297, D298 equivalent to D118;
- D123, D166, D167 equivalent to D119;
- D231, D233, D234, D235, D236 equivalent to D120;
- D165 equivalent to D124;
- D126, D171, D172 equivalent to D125;
- D135, D136, D137, D138 equivalent to D134;
- D190, D191, D194 equivalent to D139;
- D195 equivalent to D140;
- D196 equivalent to D141;
- D143 equivalent to D142;
- D151 equivalent to D144;
- D159 equivalent to D156;
- D271 equivalent to D160;
- D163, D178, D210, D222, D270 equivalent to D161;
- D279 equivalent to D162;
- D197 equivalent to D164;
- D170 equivalent to D169;
- D174, D175 equivalent to D173;
- D177 equivalent to D176;
- D180, D181, D182, D183 equivalent to D179;
- D186, D275, D278 equivalent to D185;
- D302 equivalent to D198;
- D201 equivalent to D200;
- D263 equivalent to D203;

- D209 equivalent to D204;
- D221 equivalent to D207;
- D225 equivalent to D208;
- D224 equivalent to D223;
- D262 equivalent to D227;
- D261 equivalent to D229;
- D267 equivalent to D230;
- D243, D244 equivalent to D242;
- D246 equivalent to D245;
- D248 equivalent to D247;
- D250, D251, D293, D295 equivalent to D249;
- D253, D254, D255, D256 equivalent to D252;
- D265, D266, D268, D269 equivalent to D260;
- D273, D274, D277 equivalent to D272;
- D286 equivalent to D281;
- D289 equivalent to D288.

4.1.5 DFFR_X1 CAT model

This is the third CAT model created by CMGen presented with a closer and complete look. This cell is a flip-flop, it has 3 inputs, CK, RN and D and 2 outputs Q and QN. The total number of CAT defects present in the CTM file is 456. The CAT_STATs are 101, and the CAT_DYNs are 97, so there are 258 undetected defects. The detection tables for the CAT_STAT and CAT_DYN are reported in the figure 4.12.

```
- [Table, Static]
- [CK,RN, D, Q,QN, Q-,QN-, D1,D2,D4,D5,D6,D14,D22,D52,D131,D173,D174]
- [P,  1, 0, 0, 1, 0,  1,   1, 1, 1, 1, 0,  3,  3,  0,   0,   1,   0]
- [P,  1, 1, 1, 0, 0,  1,   2, 3, 3, 3, 3,  3,  0,  3,   0,   0,   2]
- [P,  1, 1, 1, 0, 1,  0,   2, 2, 2, 3, 2,  2,  0,  3,   3,   0,   2]
- [P,  1, 0, 0, 1, 1,  0,   1, 3, 2, 0, 3,  3,  3,  0,   0,   1,   0]
- [Table, Dynamic]
- [CK,RN, D, Q,QN, Q-,QN-, D7,D199,D273]
- [ P, 1, F, F, R, 1,   0,  3,   2,   1]
```

Figure 4.12: DFFR_X1 static and dynamic table

The table columns are only the prime CAT faults, the others equivalent defects are:

- D12, equivalent to D1;
- D3, D13, D15, D20 equivalent to D2;
- D17, equivalent to D4;
- D16, D18, D19, D56 equivalent to D5;
- D21, equivalent to D6;
- D102, D103, D104, D105, D106, D197, D108, D109, D110, D111, D112, D113 equivalent to D22;
- D114, D115, D116, D117, D118, D119, D159, D160, D161, D162, D163, D164 equivalent to D22;
- D165, D166, D167, D168, D169, D170, D171, D172 equivalent to D22;
- D56, D59, D61, D72, D73, D74, D98, D99, D100, D101, D120, D121, D122 equivalent to D52;
- D123, D124, D125, D126, D127, D128, D129, D130, D132, D133, D134 equivalent to D52;
- D135, D136, D137, D138, D139, D140, D141, D142, D143, D144, D145 equivalent to D52;
- D146, D147, D148, D149, D150, D151, D152, D153, D154, D155, D156, D157, D158 equivalent to D52.

The other equivalent dynamic defects are:

- D8, D23, D24, D25, D57, D62, D63, D64, D66, D68 equivalent to D7;
- D69, D71, D75, D76, D77, D78, D79, D80, D81, D82 equivalent to D7;

- D83, D84, D85, D86, D87, D88, D89, D90, D91, D92 equivalent to D7;
- D93, D94, D95, D96, D97, D194, D195, D217, D235, D246 equivalent to D7;
- D249, D255, D256, D257, D258, D259, D260, D261, D262, D264 equivalent to D7;
- D265, D283, D291, D303, D304, D305, D306, D323, D324, D325 equivalent to D7;
- D326, D328, D329, D335, D339, D350, D375, D376, D382 equivalent to D7;
- D385, D398, D408, D409, D411, D412, D413, D428, D431, D437 equivalent to D7;
- D441, D446, D447 equivalent to D7;
- D200, D201, D248, D250, D251, D252, D318, D348, D404, D434 equivalent to D199;
- D274, D452 equivalent to D273.

4.2 Z01X fault simulator

After using the CMGen tool to create the CAT fault model of every cell in the logic synthesis library, as explained in the chapter 3, the remaining two passages are the creation of a CAT fault list generator and the use of a CAT fault simulator. The first one depends on the chosen CAT fault simulator. For the implementation, the tool selected as CAT fault simulator is Z01X, which according to the user guide [5] "is a compiled-code logic simulator that handles gate-level, switch-level, RTL, and behavioral designs. It also has built-in toggle simulation capability. Z01X is the only compiled-code Verilog fault simulator, providing traditional stuck-at fault, transition, bridge and IDDQ fault models required for the detection of deep sub-micron manufacturing defects. It is deterministic, providing actual results based on rigorous examination of the entire design".
The Z01X internal flow is divided into 3 main steps. The first one is the compilation of the design, where an executable is created with the default zoix.sim and it will be use in the next steps. The second passage is the logic simulation. It is a recommended prerequisite to fault simulation and it is used to simulate the test stimuli on the UUT and check that the simulation results correspond to the expected ones. The third phase is the fault simulation.
Given a list of faults of a certain fault model and a set of input patterns, a simulation is performed to understand what faults can be detected and which not. Z01X uses the concepts of good machine and faulty machine to perform the fault simulation. As already explained, the good machine represents the UUT response to the test stimuli without any internal defects. For every fault, in the fault list, a faulty machine (FM) is created. In particular, a FM is the union of the good machine and the injection of a fault inside it. When the fault simulation is performed, the FMs are simulated with the test stimuli. The general Z01X flow is reported in figure 4.13.

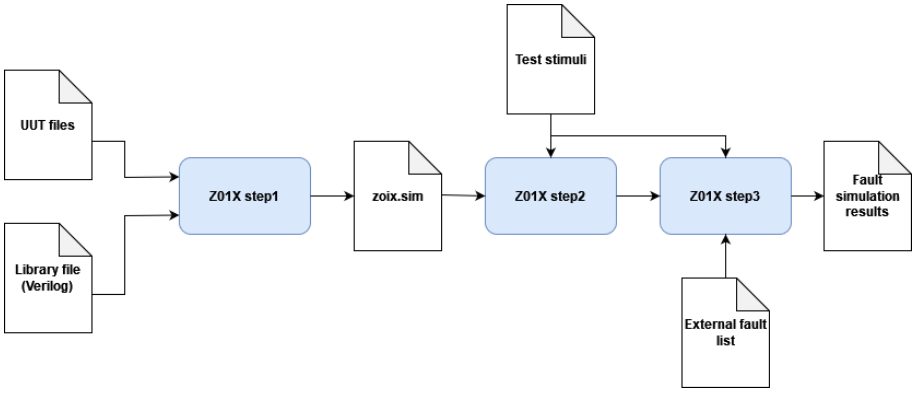


Figure 4.13: Z01X general flow

In the chapter 5 for every result is presented the Z01X flow used to create them.

4.2.1 Z01X and CAT flow

Z01X is a fault simulator that can handle different input patterns formats and different fault models. According to the user guide, [5], to simulate the CATs faults, the user has to write a list. For every defect on a cell, we need to specify:

- if it is a sa0 or a sa1 i.e., if the defect is detected when the output of the cell is 0, it is represented with sa1, otherwise with sa0;
- the type of fault before the simulation, (Not Attempted);
- the path to the defect, for example, if a the cell is instantiated with the name "reg1", and is inside a module instantiated with the name "registers", that is inside the module topModule and the name of the output port is Q, the path will be: "topModule/registers/reg1/Q";
- the name of the CAT fault in the CTM file, for example, D1, D34 ecc...;
- Cell Test Model Generator File (the path to the CTM files of the cells in the design).

According to Z01X user guide, the list of CAT faults for the UUT can be taken from TMAX, but considering the fault simulations performed in the section 4.3 and the analysis in the section 4.2.2 the TMAX CAT fault list can not be used with Z01X because it is inaccurate.
It is also worthwhile explaining two modification to made on the CTM files. As already shown in the figure 4.5, in the description of CAT fault, there is an attribute named "Instance" with the name of the instance of the defect in the SPF file of the cell. If Z01X tries to read the CTM file without any change, like the one reported in the figure 4.5, Z01X will throw an error. The error is reported in the figure 4.14.

```
Z01X (TM) Version: S-2021.09-SP1-1 (64-bit, Jan 11 2022) for Linux 64-bit

Info:    Using standard file format
Command: fr2fdef -fr __globfiles__/ScriptCtmStep2.sff +format+standard +verbose -l fr2fdef.log

Info:    Loading design
Info:    Reading fault source file __globfiles__/ScriptCtmStep2.sff
Info:    Using Standard Fault Format Parser
Info:    Language = Verilog
Info:    Delimiter = .
Info:    Escape = \
Info:    Loading CTM File:
/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myFA_X1.spf.CTM
Error!     Line 42: Unknown keyword: Instance:
Info:    Error parsing ctm file:
Info:    Error parsing standard file
Info:    Writing Merge Report
```

Figure 4.14: Z01X Istance message

So it is needed to comment the "Instance" attribute in the CTM files like show in the figure 4.15.

```
- Cell: FA_X1
  InSignals: [A, B, CI]
  OutSignals: [CO, S]
  Defects:
    - Id: D2
      Type:  short
      Description:  C_x_PM_FA_X1%VDD_c1 x_PM_FA_X1%VDD_64 VSS 1  status=static_det
#      Instance: C_x_PM_FA_X1%VDD_c1
      Attributes:
      - Class: DT
      - Behavior: stuck
```

Figure 4.15: Z01X Istance commented

Z01X recognizes only the open and short defects in the CTM files. In fact, if the tool tries to read a transistor defect in the CTM file, it will throw an error as the one shown in the figure 4.16.

```
Z01X (TM) Version: S-2021.09-SP1-1 (64-bit, Jan 11 2022) for Linux 64-bit

Info:    Using standard file format
Command: fr2fdef -fr __globfiles__/ScriptCtmStep2.sff +format+standard +verbose -l fr2fdef.log

Info:    Loading design
Info:    Reading fault source file __globfiles__/ScriptCtmStep2.sff
Info:    Using Standard Fault Format Parser
Info:    Language = Verilog
Info:    Delimiter = .
Info:    Escape = \
Info:    Loading CTM File:
/home/i.guglielminetti/prove_zoix/iacopoLibrary/comparison_TMAX_ZOIX/myCTM/myFA_X1.spf.CTM
Error!     Line 1134: Unknown defect type: transistor_short_gate_drain
Info:    Error parsing ctm file:
Info:    Error parsing standard file
Info:    Writing Merge Report
```

Figure 4.16: Z01X transistor defect message

So in all the analyses with Z01X, only defects with shorts and opens will be used.

4.2.2 Z01X and TMAX faults list

According to the Z01X user guide [5], it is possible to take the CAT faults list from TMAX and reuse it in Z01X but the following tests show that this option will produce wrong results.
The first test was performed on the cell FA_X1 whose model is reported in the section 4.1.4. This cell has 2 outputs, but for this experiment a Verilog circuit was created in which only the output CO is instantiated, as in figure 4.17.

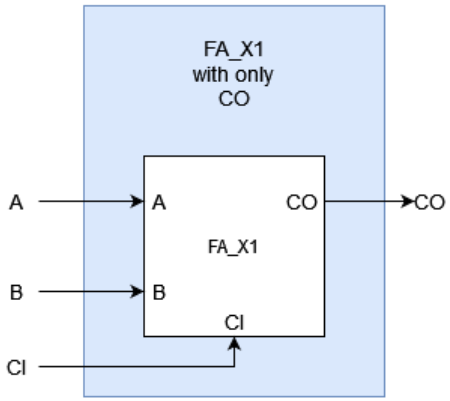


Figure 4.17: FA_X1 cell with only CO output

To generate the TMAX list of CAT_STAT faults and the test patterns able to cover them, it was created a TMAX script that uses TMAX CAT ATPG. The data flow is summarized in the figure 4.18.

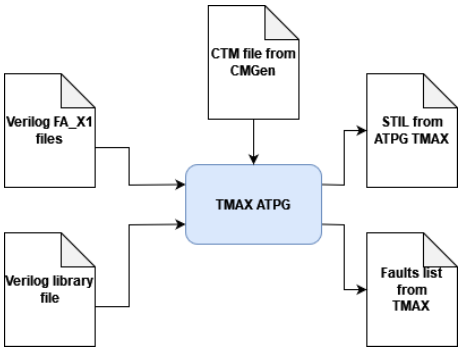


Figure 4.18: TMAX ATPG flow

The list of faults created by TMAX is in figure 4.19.

```
sa0   NC   FAX1_i/CO D3
sa0   NC   FAX1_i/CO D102
sa0   NC   FAX1_i/CO D98
sa0   NC   FAX1_i/CO D92
sa0   NC   FAX1_i/CO D89
sa0   NC   FAX1_i/CO D84
sa0   NC   FAX1_i/CO D77
sa0   NC   FAX1_i/CO D75
sa1   NC   FAX1_i/CO D70
sa1   NC   FAX1_i/CO D66
sa0   NC   FAX1_i/CO D59
sa0   NC   FAX1_i/CO D58
sa0   NC   FAX1_i/CO D57
sa1   NC   FAX1_i/CO D42
sa0   NC   FAX1_i/CO D18
sa1   NC   FAX1_i/CO D16
sa0   NC   FAX1_i/CO D14
sa0   NC   FAX1_i/CO D13
sa0   NC   FAX1_i/CO D7
sa0   NC   FAX1_i/CO D6
sa1   NC   FAX1_i/CO D5
sa0   NC   FAX1_i/CO D4
```

Figure 4.19: TMAX fault list

The number of patterns in the STIL file created by TMAX is 6 and are reported in the table 4.1.

A	B	CI	CO
1	1	0	1
0	0	1	0
0	1	1	1
0	1	0	0
1	0	1	1
1	1	1	1

Table 4.1: STIL patterns created by TMAX

According to TMAX, these patterns covers all the faults. The following step was to simulate the pattern created by TMAX using Z01X. The flow for the fault simulator is reported in figure 4.20.

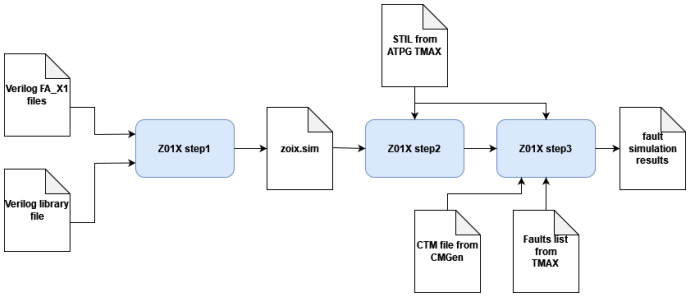


Figure 4.20: ZOIX STIL flow

After the fault simulation, the obtained results are the ones shown in figure 4.21 .

```
sa0 DD myFA_X1/FAX1_i/CO D3
sa0 DD myFA_X1/FAX1_i/CO D102
sa0 DD myFA_X1/FAX1_i/CO D98
sa0 DD myFA_X1/FAX1_i/CO D92
sa0 DD myFA_X1/FAX1_i/CO D89
sa0 DD myFA_X1/FAX1_i/CO D84
sa0 DD myFA_X1/FAX1_i/CO D77
sa0 DD myFA_X1/FAX1_i/CO D75
sa1 DD myFA_X1/FAX1_i/CO D70
sa1 DD myFA_X1/FAX1_i/CO D66
sa0 DD myFA_X1/FAX1_i/CO D59
sa0 DD myFA_X1/FAX1_i/CO D58
sa0 DD myFA_X1/FAX1_i/CO D57
sa1 DD myFA_X1/FAX1_i/CO D42
sa0 ND myFA_X1/FAX1_i/CO D18
sa1 NI myFA_X1/FAX1_i/CO D16
sa0 DD myFA_X1/FAX1_i/CO D14
sa0 DD myFA_X1/FAX1_i/CO D13
sa0 NI myFA_X1/FAX1_i/CO D7
sa0 NI myFA_X1/FAX1_i/CO D6
sa1 DD myFA_X1/FAX1_i/CO D5
sa0 DD myFA_X1/FAX1_i/CO D4
```

Figure 4.21: ZOIX results

As it is possible to observe, not all the faults are covered. So which report is the right one?
Looking at the data, it is possible to deduce that the 2 tools interpret the detection table of multi-port CAT in an opposite way. When on the detection table of a file CTM TMAX reads a 2, it creates a faults on the first port CO. Z01X instead, when reads 2 on the detection table, it understands that the defect is on the second port, S. For example, the defect D6 in the report was created on the port CO. In the static detection table in figure 4.11, it is represented with only "2" values. Z01X was not able to detect this CAT fault. Also, it does not detect the other CAT faults represented with only 2 values, D7 and D16. Finally, in the Z01X results, the defect D18 is not detected. Looking at the static detection table in figure 4.11, D18 is detectable on CO with the pattern A=0, B=0 CI=1 and CO is expected to be 0. This pattern is present in the STIL file, so why does it not detect D18? In the fault list, this CAT faults is represented as sa0. According to Z01X user guide, when a CAT fault is detected by a pattern, if the output port is 0, it

needs to be label in the fault list as sa1. If the output port is 1, it needs to be labelled in the fault list as sa0. As already said, TMAX and Z01X interpret the 1 and the 2 in the detection table of the CTM file in the opposite way. So for TMAX D18 is detectable on S by the pattern A=0, B=0 CI=1 and S=0, so it represent the CAT faults using sa1. According to Z01X, D18 is detectable on CO output port, in the pattern A=0, B=0 CI=1 CO is equal to 0, so it need to be represented with sa1.
Now that we are aware that TMAX and Z01X read the CTM files in an opposite ways regarding the CAT faults detectable on two output ports, the second test consists in inverting the values in the detection table of the multi-output cells, to see if TMAX results can be compared with Z01X. Another version of FA_X1 static detection table was created, reported in the figure 4.22

```
- [Table, Static]
- [A,B, CI,CO, S,D2,D3,D4,D5,D6,D7,D13,D14,D16,D18,D42,D54,D55,D57,D58,D59,D63,D66,D70,D75,D76,D77,D82,D84,D89,D92,D96,D98,D102,D103,D104,D108]
- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  0,  2]
- [0, 0, 1, 0, 1, 2, 0, 0, 0, 0, 0,  0,  0,  0,  2,  3,  0,  2,  0,  2,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
- [0, 1, 0, 0, 1, 2, 0, 2, 3, 0, 0,  0,  0,  1,  0,  3,  0,  0,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  2,  0,  0,  0,  0]
- [0, 1, 1, 1, 0, 0, 2, 0, 0, 1, 0,  2,  0,  0,  1,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  0,  0,  3,  3,  0,  0,  3,  0,  2,  2]
- [1, 0, 0, 0, 1, 2, 0, 0, 3, 0, 0,  2,  0,  1,  0,  3,  0,  0,  0,  0,  0,  0,  3,  0,  0,  2,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0]
- [1, 0, 1, 1, 0, 0, 0, 2, 0, 1, 0,  0,  2,  0,  1,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  0,  0,  0,  0,  2,  2]
- [1, 1, 0, 1, 0, 0, 2, 0, 0, 1, 0,  2,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  3,  3,  0,  3,  0,  0,  2,  2]
- [1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,  1,  1,  1,  1,  0,  2,  2,  0,  2,  2,  0,  0,  0,  2,  2,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0]
```

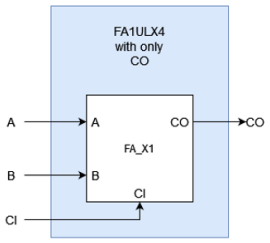
Figure 4.22: FA_X1 static table reversed

The 2 values in the table become 1, the 1 becomes 2 and the 3 stay the same. The TMAX flow in the figure 4.18 was used, with this new CTM file, and the obtained fault list in showed the figure 4.26

```
sa1   NC   FAX1_i/CO D2
sa1   NC   FAX1_i/CO D108
sa0   NC   FAX1_i/CO D104
sa1   NC   FAX1_i/CO D103
sa0   NC   FAX1_i/CO D102
sa0   NC   FAX1_i/CO D98
sa1   NC   FAX1_i/CO D96
sa0   NC   FAX1_i/CO D92
sa0   NC   FAX1_i/CO D89
sa0   NC   FAX1_i/CO D84
sa1   NC   FAX1_i/CO D82
sa0   NC   FAX1_i/CO D77
sa1   NC   FAX1_i/CO D76
sa0   NC   FAX1_i/CO D75
sa1   NC   FAX1_i/CO D70
sa1   NC   FAX1_i/CO D66
sa1   NC   FAX1_i/CO D63
sa0   NC   FAX1_i/CO D59
sa1   NC   FAX1_i/CO D58
sa0   NC   FAX1_i/CO D57
sa1   NC   FAX1_i/CO D55
sa0   NC   FAX1_i/CO D54
sa1   NC   FAX1_i/CO D42
sa1   NC   FAX1_i/CO D18
sa0   NC   FAX1_i/CO D14
sa0   NC   FAX1_i/CO D13
sa1   NC   FAX1_i/CO D5
sa1   NC   FAX1_i/CO D4
sa0   NC   FAX1_i/CO D3
```

Figure 4.23: FA_X1 fault list from reverse CTM file

This time the faults inserted on CO are the same needed by Z01X. This means that by creating the reversed file of the normal CMGen file of a cell with 2 outputs and reading it with TMAX, TMAX will select the same Z01X faults.
Could this problem of the "inversion" of faults of multi-output cells, depend on the library used? To add one more example, it was taken into account the FA1ULX4 cell from the 130nm HCMOS9A standard cell library by STMicroelectronics. This is a full-adder like the precedent cell FA_X1. It was develop a Verilog circuit where the cell have only one output instantiated, like in figure 4.24. Next to it is also shown a piece of the static detection table of the cell.




```
- [Table,Static]
- [A,B,CI,CO,Z,D1,D3,D5,D8,D11,D12,D14,D22,D23,D28,D45,D66,D67,D82,D86,D90,D91,D94,D106,D108]
- [0,0, 0, 0,0, 0, 2, 0, 0,  0,  0,  1,  0,  2,  1,  0,  0,  0,  0,  1,  0,  0,  1,   0,   1]
- [0,0, 1, 0,1, 3, 3, 3, 0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  1,   0,   0]
- [0,1, 0, 0,1, 3, 3, 0, 0,  0,  0,  0,  0,  2,  0,  0,  0,  3,  0,  0,  3,  0,  1,   0,   0]
- [0,1, 1, 1,0, 0, 0, 0, 0,  1,  3,  1,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  1,   3,   1]
- [1,0, 0, 0,1, 3, 0, 0, 0,  0,  0,  1,  0,  2,  0,  0,  0,  3,  0,  1,  0,  3,  1,   0,   0]
- [1,0, 1, 1,0, 0, 3, 0, 0,  0,  3,  0,  0,  2,  0,  3,  0,  0,  3,  0,  0,  0,  1,   3,   1]
- [1,1, 0, 1,0, 0, 3, 0, 3,  0,  3,  0,  0,  2,  0,  0,  1,  0,  0,  0,  0,  0,  1,   3,   1]
- [1,1, 1, 1,1, 0, 2, 0, 0,  1,  0,  1,  1,  2,  0,  0,  0,  0,  0,  0,  0,  0,  1,   2,   0]
```

Figure 4.24: FA1ULX4 scheme and static detection table

The flow showed in figure 4.18 was used again to make TMAX create the CAT_STAT fault list, that is reported in the figure 4.25.

```
sa1 NC FA1ULX4_i/CO D1
sa0 NC FA1ULX4_i/CO D910
sa0 NC FA1ULX4_i/CO D368
sa1 NC FA1ULX4_i/CO D343
sa1 NC FA1ULX4_i/CO D306
sa1 NC FA1ULX4_i/CO D186
sa1 NC FA1ULX4_i/CO D132
sa0 NC FA1ULX4_i/CO D106
sa1 NC FA1ULX4_i/CO D91
sa1 NC FA1ULX4_i/CO D90
sa0 NC FA1ULX4_i/CO D82
sa1 NC FA1ULX4_i/CO D67
sa0 NC FA1ULX4_i/CO D45
sa1 NC FA1ULX4_i/CO D23
sa0 NC FA1ULX4_i/CO D12
sa0 NC FA1ULX4_i/CO D8
sa1 NC FA1ULX4_i/CO D5
sa1 NC FA1ULX4_i/CO D3
```

Figure 4.25: FA1ULX4 scheme and static detection table

Symilary as for the nangate cell FA_X1, the CAT_STAT defects are switched in comparison to Z01X. In fact, D23 is only detectable on Z but is present in the list. D11, D14, D22, D28, D66, D86, D94, D108 are detectable only on CO and they are not present in the fault list of TMAX. So the inversion of the output ports in comparison to Z01X does not depend on the library used, but it is an implementation decision for TMAX. Now, after using the CTM files reversed in TMAX to create the correct list of faults, it is not possible to copy the faults list of TMAX and simulate it with Z01X because as already mentioned the CAT faults are not equivalent to a sa1 or sa0, they could be both of them and detectable on more output port.
The last test demonstrated that. A STIL file with 7 patterns was created, reported in the table 4.2.

Pattern number	A	B	CI	CO
1	0	1	1	1
2	0	0	0	0
3	0	0	1	0
4	0	1	0	0
5	1	1	1	1
6	1	1	0	1
7	1	0	1	1

Table 4.2: STIL patterns

The following step was to fault simulate the faults of the TMAX list with Z01X using this pattern, according to the flow 4.20. The results are reported in the figure 4.11

```
sa0 DD myFA_X1/FAX1_i/CO D3
sa0 DD myFA_X1/FAX1_i/CO D102
sa0 DD myFA_X1/FAX1_i/CO D98
sa0 DD myFA_X1/FAX1_i/CO D92
sa0 DD myFA_X1/FAX1_i/CO D89
sa0 DD myFA_X1/FAX1_i/CO D84
sa0 DD myFA_X1/FAX1_i/CO D77
sa0 DD myFA_X1/FAX1_i/CO D75
sa1 DD myFA_X1/FAX1_i/CO D70
sa1 DD myFA_X1/FAX1_i/CO D66
sa0 DD myFA_X1/FAX1_i/CO D59
sa0 DD myFA_X1/FAX1_i/CO D58
sa0 DD myFA_X1/FAX1_i/CO D57
sa1 DD myFA_X1/FAX1_i/CO D42
sa0 ND myFA_X1/FAX1_i/CO D18
sa1 NI myFA_X1/FAX1_i/CO D16
sa0 DD myFA_X1/FAX1_i/CO D14
sa0 DD myFA_X1/FAX1_i/CO D13
sa0 NI myFA_X1/FAX1_i/CO D7
sa0 NI myFA_X1/FAX1_i/CO D6
sa1 DD myFA_X1/FAX1_i/CO D5
sa0 DD myFA_X1/FAX1_i/CO D4
```

Figure 4.26: Z01X results for the faults from the REVERSE CTM file

D76 could be detected from pattern 5, but it does not happen because the TMAX faults list is incomplete for Z01X.

4.3 Correct representation of Z01X fault list

Before proceeding with the presentation of the CAT fault list generator developed for Z01X, it is important to present the correct CAT fault list representation created by me specifically for the Z01X tool as described in 3.3.4. There were developed 5 models that describe the CAT faults. They are explained in the following sections.

4.3.1 CAT normal stuck

According to the TMAX list format and Z01X user guide, [5], every cell-aware defect can be represented in the fault list with sa0 or a sa1. This kind of CAT faults will be called normal-stuck (NS). An example is the defect D174 in the cell DFFR_X1. According to table 4.12, this defect could be represented in this way: "sa1 NA DFFR_X1/QN D174".

4.3.2 CAT multi-port

There are CAT faults that, in cells with 2 output ports, could be detected on both of them. This kind of CAT faults will be called multi-port (MP). For example on the cell DFFR_X1, in the static detection table in the figure 4.12, the defect D1 can be detected on both the outputs. To correctly simulate these faults it is necessary to add in the fault list this defect on both the ports and then create a process of post-simulation able to collapse the result of these 2 faults in the original fault.
To present the correct Z01X flow, a specific example has been developed. The simple UUT is reported in the figure 4.27.

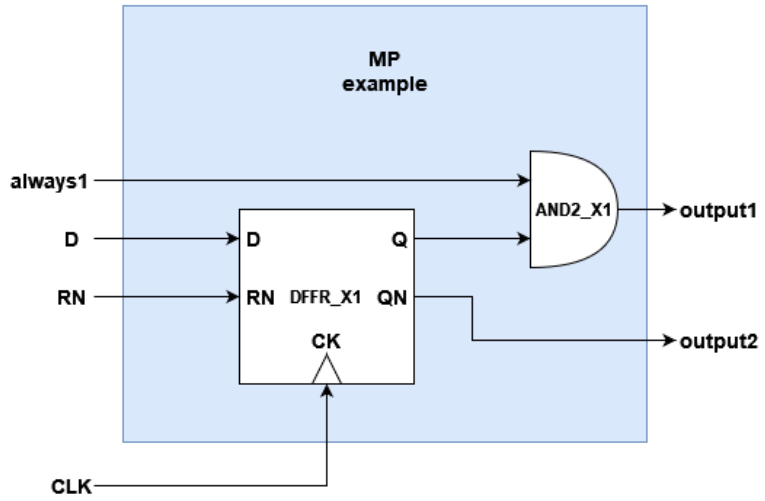


Figure 4.27: Simple example circuit for MP

The input fault list given to Z01X includes the following data:

- sa1 NA myDFFR_X1_i/DFFRX1_i/Q D1;

- sa1 NA myDFFR_X1_i/DFFRX1_i/QN D1.

This means that in the fault list the fault D1 has been placed on both the output ports. The test stimuli are contained in a testbench that has been simulated with Z01X. The results of the simulation has been recorded in a VCD file. At last, this VCD has been fault simulated with Z01X. The complete flow is reported in the figure 4.28.

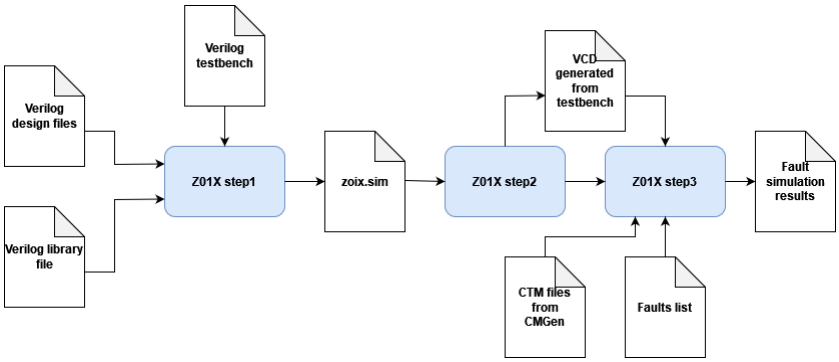


Figure 4.28: Z01X flow with testbench patterns

The results are reported in figure 4.29.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/zoix/myCTM/myDFFR_X1.spf.CTM")

    sa1 DD b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/Q D1
    sa1 NI b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/QN D1
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                        Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                             2                 2
#
# Detected                          DG      1  50.00%         1  50.00%
#   Dropped Detected                  DD      1  50.00%         1  50.00%
# Unselected                        NG      1  50.00%         1  50.00%
#   Not Injected                      NI      1  50.00%         1  50.00%
# Potential                         PG      0   0.00%         0   0.00%
#   Dropped Potential                 PD      0   0.00%         0   0.00%
# Not Detected                      ND      0   0.00%         0   0.00%
#
# Test Coverage                             50.00%            50.00%
# Fault Coverage                            50.00%            50.00%
#-------------------------------------------------------------------------
```

Figure 4.29: fault simulation result for MP

The test stimuli only detect D1 on the Q output port. These data are correct, they show that is important to set the MP faults on both the output ports. After collapsing the faults D1, we can delete in the final coverage report 1 NI and take only the DD. If we had set D1 only on QN, the faults in the final report would be set like NI, decreasing the total fault coverage. It is important to underline that, for detecting the CAT fault D1, if it has been correctly set on both the output ports, it is enough to detect the fault on one port.

4.3.3 CAT multi-stuck

There are faults that could be detected only on one output port of the cell, for example, D35 regarding XNOR2_X1. But this defect is special, in fact, it could be detected when the output port ZN is both 0 or 1, as we can see in the figure 4.30.

```
- [Table, Static]
- [A, B, ZN, D1,D6,D21,D32,D34,D35,D42]
- [0, 0,  1,  1, 1,  0,  0,  0,  0,  0]
- [0, 1,  0,  0, 0,  1,  0,  0,  0,  1]
- [1, 0,  0,  0, 0,  1,  1,  0,  1,  0]
- [1, 1,  1,  0, 1,  0,  0,  1,  1,  0]
- [Table, Dynamic]
- [A, B, ZN, D3,D57,D59,D60,D65,D67,D74,D75,D76,D111,D129,D143]
- [0, R,  F,  0,  1,  0,  1,  0,  0,  1,  1,  0,   0,   0,   0]
- [R, 0,  F,  0,  1,  0,  0,  1,  1,  0,  1,  0,   1,   0,   0]
- [0, F,  R,  1,  0,  0,  0,  1,  0,  0,  0,  1,   1,   0,   1]
- [R, 1,  R,  0,  0,  1,  0,  0,  0,  0,  1,  1,   1,   0,   0]
- [F, 0,  R,  1,  0,  0,  0,  0,  0,  1,  0,  1,   0,   0,   1]
- [1, R,  R,  0,  0,  1,  0,  0,  0,  0,  1,  1,   1,   0,   0]
- [F, 1,  F,  1,  1,  0,  1,  0,  0,  1,  0,  0,   0,   1,   0]
- [1, F,  F,  1,  1,  0,  0,  1,  1,  0,  0,  0,   1,   0,   0]
```

Figure 4.30: XNOR2_X1 static and dinamic tables

So this is not enough to represent this CAT with sa0 or sa1, is both of them, this kind of CAT faults will be called multi-stuck (MS). In order to simulate this particular defect, I used the simple circuit reported in the figure 4.31.

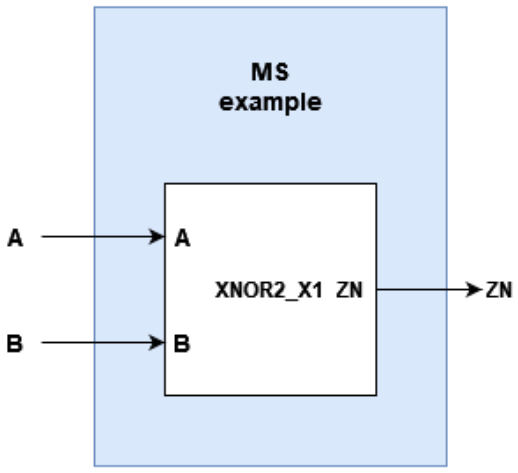


Figure 4.31: Simple example circuit for MS

The input fault list given to Z01X has the following data:

- sa0 NA myXNOR2_X1_i/XNOR2X1_i/ZN D35;
- sa1 NA myXNOR2_X1_i/XNOR2X1_i/ZN D35.

Then test stimuli are contained in a testbench that has been simulated with Z01X. The results of the simulation have been recorded in a VCD file. At last, this VCD has been fault simulated with Z01X. The complete flow is reported in the figure 4.28 .
The results are reported in the figure 4.32.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/zoix/myCTM/myXNOR2_X1.spf.CTM")

    sa0 DD b02_Noscan_tb/myXNOR2_X1_i/XNOR2X1_i/ZN D35
    sa1 NC b02_Noscan_tb/myXNOR2_X1_i/XNOR2X1_i/ZN D35
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                    Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                         2                 2
#
# Detected                        DG        1  50.00%         1  50.00%
#   Dropped Detected                DD        1  50.00%         1  50.00%
# Unselected                      NG        1  50.00%         1  50.00%
#   Not Controlled                  NC        1  50.00%         1  50.00%
# Potential                       PG        0   0.00%         0   0.00%
#   Dropped Potential               PD        0   0.00%         0   0.00%
# Not Detected                    ND        0   0.00%         0   0.00%
#
# Test Coverage                             50.00%            50.00%
# Fault Coverage                            50.00%            50.00%
#-------------------------------------------------------------------------
```

Figure 4.32: Fault simulation result for MS

It is important to correctly set the MS defects on both **sa0** and **sa1** values, otherwise, we will possibly lose some test coverage. To avoid depending on only VCD test stimuli, the MS CAT fault has been tested with a different flow, more precisely the TMAX ATPG has been used to create a STIL file with 3 patterns:

- A=0, B=1 and ZN=0;
- A=0, B=0 and ZN=1;
- A=1, B=1 and ZN=1.

These patterns, according to the static table in the figure 4.30, are able to detect D35 when it is represented with sa0. Then Z01X flow for STIL stimuli is reported in the figure 4.20. For the first simulation, it was used the fault list from TMAX and the results are reported in the figure 4.33.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myXNOR2_X1.spf.CTM")

    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D1
    sa1 DD myXNOR2_X1/XNOR2X1_i/ZN D42
    sa1 ND myXNOR2_X1/XNOR2X1_i/ZN D35
    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D34
    sa1 NI myXNOR2_X1/XNOR2X1_i/ZN D32
    sa1 DD myXNOR2_X1/XNOR2X1_i/ZN D21
    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D6
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                        Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                             7                 7
#
# Detected                        DG        5  71.43%         5  71.43%
#   Dropped Detected                DD        5  71.43%         5  71.43%
# Unselected                      NG        1  14.29%         1  14.29%
#   Not Injected                    NI        1  14.29%         1  14.29%
# Potential                       PG        0   0.00%         0   0.00%
#   Dropped Potential               PD        0   0.00%         0   0.00%
# Not Detected                    ND        1  14.29%         1  14.29%
#
# Test Coverage                             71.43%            71.43%
# Fault Coverage                            71.43%            71.43%
#-------------------------------------------------------------------------
```

Figure 4.33: Fault simulation result for TMAX faults list

D35 results as ND (not detected). If D35 was correctly represented, like a MS faults, D35 will be detected at least one time. In fact, the third pattern could detect D35 only when it is present in the list like a sa0. This is another proof that the TMAX faults lists are incomplete for Z01X. The results of the second simulation are reported in the figure 4.34.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myXNOR2_X1.spf.CTM")

    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D1
    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D6
    sa1 DD myXNOR2_X1/XNOR2X1_i/ZN D21
    sa1 NI myXNOR2_X1/XNOR2X1_i/ZN D32
    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D34
    sa0 DD myXNOR2_X1/XNOR2X1_i/ZN D35
    sa1 ND myXNOR2_X1/XNOR2X1_i/ZN D35
    sa1 DD myXNOR2_X1/XNOR2X1_i/ZN D42
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                        Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                             8                 8
#
# Detected                        DG        6  75.00%         6  75.00%
#   Dropped Detected                DD        6  75.00%         6  75.00%
# Unselected                      NG        1  12.50%         1  12.50%
#   Not Injected                    NI        1  12.50%         1  12.50%
# Potential                       PG        0   0.00%         0   0.00%
#   Dropped Potential               PD        0   0.00%         0   0.00%
# Not Detected                    ND        1  12.50%         1  12.50%
#
# Test Coverage                             75.00%            75.00%
# Fault Coverage                            75.00%            75.00%
#-------------------------------------------------------------------------
```

Figure 4.34: Fault simulation result for faults list with MS

4.3.4 CAT half-multi-stuck

As already presented, there are defects that can be detected on both the output ports of a cell, the MP. Also, we have seen MS, which need to be represented with both sa0 and sa1. These 2 kinds of CAT fault representations can be fused together, in 2 different ways. The first category is half-multi-stuck (HMS). For example, D4 regarding the cell FA_X1 is an HMS. In fact, referring to the table in figure 4.35,

```
- [Table, Static]
- [A, B,CI,CO, S,D2,D3,D4,D5,D6,D7,D13,D14,D16,D18,D42,D54,D55,D57,D58,D59,D63,D66,D70,D75,D76,D77,D82,D84,D89,D92,D96,D98,D102,D103,D104,D108]
- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,   0,   1,   0,   1]
- [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,  0,  0,  0,  1,  3,  0,  1,  0,  1,  0,  1,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,   0,   0,   0,   0]
- [0, 1, 0, 0, 1, 1, 0, 1, 3, 0, 0,  0,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,   0,   0,   0,   0]
- [0, 1, 1, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  0,  0,  3,  3,  0,  0,   3,   0,   1,   1]
- [1, 0, 0, 0, 1, 1, 0, 0, 3, 0, 0,  1,  0,  2,  0,  3,  0,  0,  0,  0,  0,  0,  0,  3,  0,  0,  1,  0,  1,  0,  0,  0,  0,   0,   0,   0,   0]
- [1, 0, 1, 1, 0, 0, 0, 1, 0, 2, 0,  0,  1,  0,  2,  0,  0,  0,  3,  3,  3,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  0,   0,   0,   1,   1]
- [1, 1, 0, 1, 0, 0, 1, 0, 0, 2, 0,  1,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,  0,  3,  0,  0,  3,  3,  0,   3,   0,   1,   1]
- [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2,  2,  2,  2,  2,  0,  1,  1,  0,  1,  1,  0,  0,  0,  1,  1,  0,  0,  0,  1,  0,  0,  0,   0,   0,   0,   0]
```

Figure 4.35: FA_X1 static table

it is possible to see that D4 is an MP, but also on the output port CO, D4 is an MS. So this single defect to be correctly simulated need to be added 3 times. In order to simulate this particular defect the simple circuit reported in the figure 4.36 has been used.

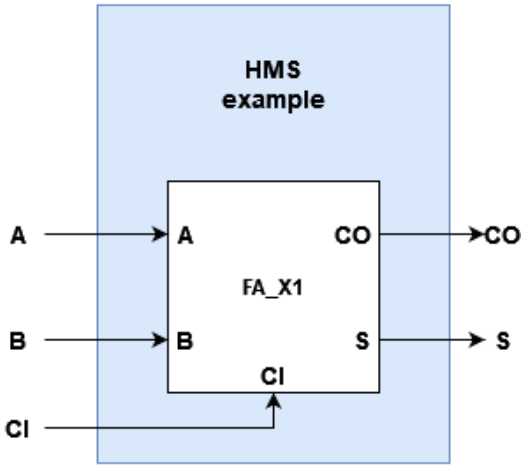


Figure 4.36: Simple example circuit for HMS

The input fault list given to Z01X have the following data:

- sa0 NA myFA_X1_i/FAX1_i/CO D4;
- sa1 NA myFA_X1_i/FAX1_i/CO D4;
- sa0 NA myFA_X1_i/FAX1_i/S D4.

The test stimuli are contained in a testbench that has been simulated with Z01X. The results of the simulation have been recorded in a VCD file. At last, this VCD has been fault simulated with Z01X. The complete flow is reported in the figure 4.28 .
The results are reported in the figure 4.37.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/zoix/myCTM/myFA_X1.spf.CTM")

    sa0 DD b02_Noscan_tb/myFA_X1_i/FAX1_i/S D4
    sa1 NC b02_Noscan_tb/myFA_X1_i/FAX1_i/CO D4
    sa0 DD b02_Noscan_tb/myFA_X1_i/FAX1_i/CO D4
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                      Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                           3                 3
#
# Detected                         DG     2  66.67%         2  66.67%
#   Dropped Detected                 DD     2  66.67%         2  66.67%
# Unselected                       NG     1  33.33%         1  33.33%
#   Not Controlled                   NC     1  33.33%         1  33.33%
# Potential                        PG     0   0.00%         0   0.00%
#   Dropped Potential                PD     0   0.00%         0   0.00%
# Not Detected                     ND     0   0.00%         0   0.00%
#
# Test Coverage                           66.67%            66.67%
# Fault Coverage                          66.67%            66.67%
#-------------------------------------------------------------------------
```

Figure 4.37: Fault simulation result for HMS

As for the MP and MS faults, it is important to correctly set the HMS defects on both the values sa0 and sa1 and on both the ports, otherwise, they could not be detected. In this particular example, it is possible to observe how multiplying a single defect can temporarily increase the test coverage. More precisely, the CAT defect D4 is represented three times but is actually one fault. After that the fault simulation produces the results, it is important to compress them in one single classification that will be taken into account for the test coverage. That is why when for a single defect there are actually 2 DD, it is important to consider only one for the coverage.

4.3.5 CAT full-multi-stuck

The second way in which MS and MP defects could mix is the category full-multi-stuck (FMS). For example, D2 regarding the cell DFFR_X1 is an FMS. In fact, looking at the table in the figure 4.12, D2 is an MP, but also on the output ports Q and QN, D2 is an MS. So this single defect to be correctly simulated needs to be added 4 times. The circuit in figure 4.27 has been reused to simulate this particular defect.
Then I wrote the list of faults, with this data:

- sa0 NA myDFFR_X1_i/DFFRX1_i/Q D2;
- sa0 NA myDFFR_X1_i/DFFRX1_i/QN D2;
- sa1 NA myDFFR_X1_i/DFFRX1_i/Q D2;
- sa0 NA myDFFR_X1_i/DFFRX1_i/QN D2.

The test stimuli are contained in a testbench that has been simulated with Z01X. The results of the simulation have been recorded in a VCD file. At last, this VCD has been

fault simulated with Z01X. The complete flow is reported in the figure 4.28 .
The results are reported in figure 4.38.

```
FaultList{
    LoadCTM("/home/i.guglielminetti/prove_zoix/iacopoLibrary/zoix/myCTM/myDFFR_X1.spf.CTM")

    sa0 ND b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/Q D2
    sa0 DD b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/QN D2
    sa1 DD b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/Q D2
    sa1 ND b02_Noscan_tb/myDFFR_X1_i/DFFRX1_i/QN D2
}
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                      Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                           4                 4
#
# Detected                         DG     2  50.00%         2  50.00%
#   Dropped Detected                 DD     2  50.00%         2  50.00%
# Potential                        PG     0   0.00%         0   0.00%
#   Dropped Potential                PD     0   0.00%         0   0.00%
# Not Detected                     ND     2  50.00%         2  50.00%
#
# Test Coverage                           50.00%            50.00%
# Fault Coverage                          50.00%            50.00%
#-------------------------------------------------------------------------
```

Figure 4.38: Fault simulation result for FMS

As for the MP and MS faults, it is important to correctly set the FMS defects the sa0 and sa1 values and on both the ports, otherwise, it is possible to lose some test coverage. More precisely, the CAT defect D2 is represented four times but is actually one fault. After that the fault simulation produces the results, it is important to compress them in one single classification that will be taken into account for the test coverage.
In this particular example, it is possible to observe how multiplying a single defect can temporarily increase the test coverage. That is why when for a single defect there are actually 2 DD, it is important to consider only one for the coverage.

4.4 CAT fault generator for Z01X

Now that the correct and full representation of CAT faults on ZOIX has been explained, the next step is to develop an automatic CAT faults generator that creates the CAT fault list from the gate-level netlist of the UUT and the CTM files cell produced by CMGen.
As already reported in the figure 3.13, the logic synthesis of an IC or SoC could produce a Verilog file with one or different modules inside it. In order to create a list of CAT faults compatible with the one reported in 4.2.1, the CAT fault generator needs to be able to create the correct path for every fault in the cells of the netlist. If the UUT is made of only one module, this is quite simple. For example for a cell instantiated as "stato0reg000" and of type DFFR_X1, the result of the fault creation is:

- sa1 NA stato0reg000/Q D1 #MP
- sa1 NA stato0reg000/QN D1 #MP

- sa0 NA stato0reg000/Q D2 #FMS
- sa1 NA stato0reg000/Q D2 #FMS
- sa1 NA stato0reg000/QN D2 #FMS
- sa0 NA stato0reg000/QN D2 #FMS
- ecc...

For a UUT with many modules, an example of the CAT faults representation of a cell with instance name "data_sync_reg_0_" and type DFFR_X1 is the following:

- sa1 NA clock_module_0/sync_reset_por/data_sync_reg_0_/Q D1 # MP
- sa1 NA clock_module_0/sync_reset_por/data_sync_reg_0_/QN D1 # MP
- sa0 NA clock_module_0/sync_reset_por/data_sync_reg_0_/Q D2 # FMS
- sa1 NA clock_module_0/sync_reset_por/data_sync_reg_0_/Q D2 # FMS
- sa1 NA clock_module_0/sync_reset_por/data_sync_reg_0_/QN D2 # FMS
- sa0 NA clock_module_0/sync_reset_por/data_sync_reg_0_/QN D2 # FMS
- ecc...

The developed CAT faults generator is a program able of generating the correct and complete list of CAT_STAT faults named "CellAwareFaultGenerator.sh". It is divided into several scripts and produces different output files with different data. The complete data flow is reported in figure 4.39.

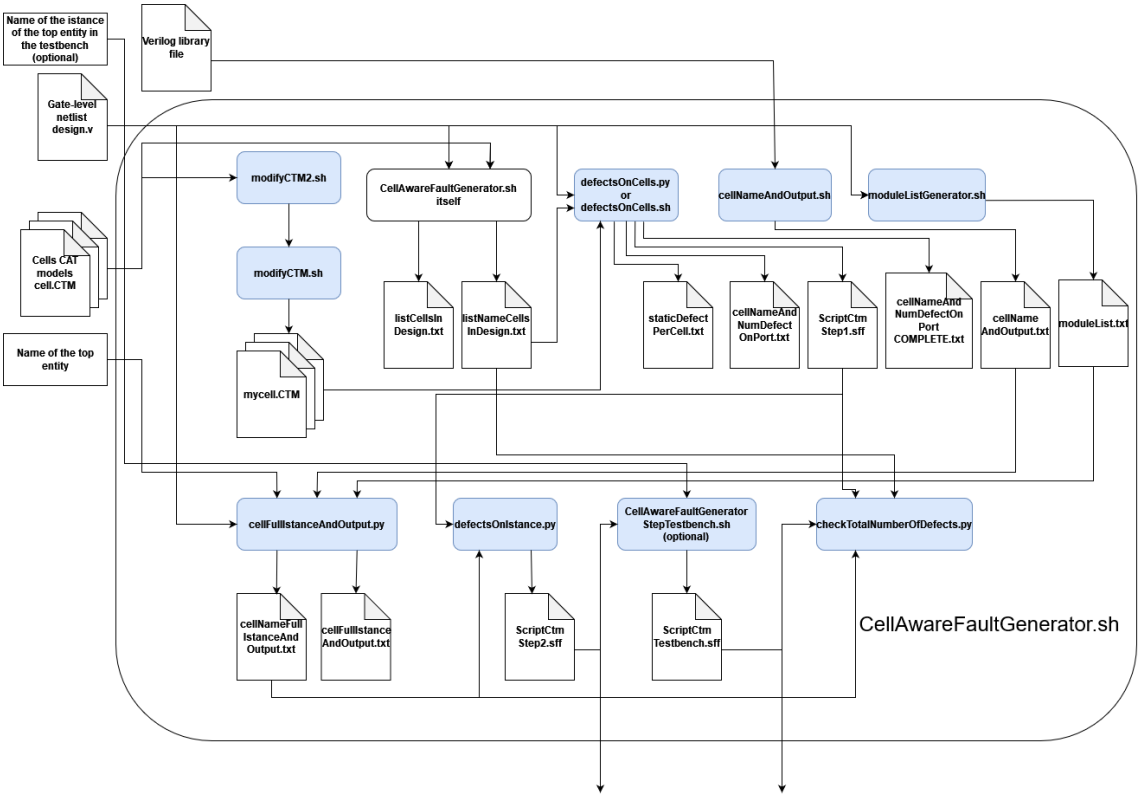


Figure 4.39: cellAwareFaultGeneratorFlow.sh internal structure

The blue blocks are the scripts used. In the next subsections the scripts that form the CAT fault generator are explained.

4.4.1 CellAwareFaultGenerator.sh

This is the main script. With the correct input files, at the end of the procedure, the correct CAT fault list for Z01X is created. The input files needed by the script to work correctly are:

- Verilog (.v) file containing the descriptions of all library cells;
- Verilog (.v) file from the logic synthesis of the design;
- the folder with the CTM files created by CMGen.

The CellAwareFaultGenerator can work in four different modalities that produce 4 different types of fault lists. According to that, there are 4 types of input arguments that the script needs to work correctly. The user has to know the name of the top entity in the design, the path to the folder with the CTM files created by CMGen and the path of the Verilog file from the logic synthesis. For example, let us imagine that the

CTM files created for the nangate cell library are stored in the folder with the path "my/Path/To/CTM", and the Verilog synthesis file is stored in a folder with the path "my/Path/To/VerilogDesign" and the name of the top entity in the design is "b02", the four different commands are:

- ./CellAwareFaultGenerator.sh my/Path/To/CTM my/Path/To/VerilogDesign/b02.v 1 dut/ b02
- ./CellAwareFaultGenerator.sh my/Path/To/CTM my/Path/To/VerilogDesign/b02.v 3 dut/ b02
- ./CellAwareFaultGenerator.sh my/Path/To/CTM my/Path/To/VerilogDesign/b02.v 0 b02
- ./CellAwareFaultGenerator.sh my/Path/To/CTM my/Path/To/VerilogDesign/b02.v 2 b02

The effects of the commands are the following:

- when the user types 1 as third argument, in the faults paths, at the beginning, the string typed as fourth argument will be appended ("dut/" in the example). This is useful when in the Z01X flow for the CAT faults there is a testbench with the design under test instantiated with a certain name. Also when it is specified 1, the program creates a fault list with prime and equivalent CAT. The prime CAT are the ones that appear in the detection tables of the CTM files, the equivalent are all the others, this concept is more clear looking at the examples of the CAT cells models in the sections 4.1.3, 4.1.4 and 4.1.5;
- when the user types 3 as third argument, only the prime CAT faults are listed in the output files, also the string inserted by the command line as fourth argument ("dut/" in the example) will be appended in the faults paths.
- when the user types 0 as third argument, in the output CAT fault list will be inserted prime and equivalent faults, without any string inserted at the beginning of the faults paths;
- when the user types 2 as third argument, in the output CAT fault list will be inserted only the prime faults, without any string inserted at the beginning of the faults paths.

With the current version, the program is able to create only Z01X faults list with CAT_STAT. The output fault list produced by the script could have a different name, depending on the modality selected by the command line. If as third argument the user inserted 1 or 3, the Z01X CAT faults list is in the file ScriptCtmTestbench.sff. In the picture 4.40 there is an example of ScriptCtmTestbench.sff.

```
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOR2_X1.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOR3_X1.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOR3_X2.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOR4_X1.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myXNOR2_X1.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myXNOR2_X2.spf.CTM")
LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myXOR2_X1.spf.CTM")
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D1 #MP
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D1 #MP
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D3 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D3 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D3 #MP full-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D3 #MP full-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D4 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D4 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D4 #MP full-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D4 #MP full-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D5 #MP half-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D5 #MP half-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D5 #MP half-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D7 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D7 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D7 #MP full-MS
sa0 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/QN D7 #MP full-MS
sa1 NA openMSP430/clock_module_0/sync_reset_por/data_sync_reg_0_/Q D14 #MP half-MS
```

Figure 4.40: ScriptCtmTestbench.sff example

In this particular example, the string inserted by the command line was “openMSP430/”. If as third argument the user inserted 0 or 2 the Z01X CAT faults list is in the file ScriptCtmStep2.sff. Figure 4.41 reports an example of this output file created for the circuit b02.

```
StatusDefinitions
{
 Redefine NA NA "Not attemped";
}
FaultList
{
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myAND2_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myAOI211_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myDFFR_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myINV_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myNOR2_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOAI21_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOAI221_X1.spf.CTM")
  LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myOR2_X1.spf.CTM")
 sa1 NA stato0reg000/Q D1 #MP
 sa1 NA stato0reg000/QN D1 #MP
 sa0 NA stato0reg000/Q D2 #MP full-MS
 sa1 NA stato0reg000/Q D2 #MP full-MS
 sa1 NA stato0reg000/QN D2 #MP full-MS
 sa0 NA stato0reg000/QN D2 #MP full-MS
 sa0 NA stato0reg000/Q D4 #MP full-MS
 sa1 NA stato0reg000/Q D4 #MP full-MS
 sa1 NA stato0reg000/QN D4 #MP full-MS
 sa0 NA stato0reg000/QN D4 #MP full-MS
 sa0 NA stato0reg000/Q D5 #MP half-MS
 sa1 NA stato0reg000/Q D5 #MP half-MS
```

Figure 4.41: ScriptCtmStep2.sff example

The scripts inside CellAwareFaultGenerator.sh and their behavior are presented in the following subsections.

4.4.2 modifyCTM.sh and modifyCTM2.sh

The bash scripts modifyCTM and modifyCTM2 together create a folder to store the CTM files correctly modified to be read by Z01X. In the files, the lines with the "Instance" attributes are commented like already shown in the figure 4.22, and the other parts of the CTM files are not modified. The complexity of these two scripts is O(n).

4.4.3 defectsOnCells.sh and defectsOnCells.py

These 2 scripts create the CAT fault model for the cells used in the design taken into account. This means that after their job, a file named ScriptCtmStep1.sff is created, where for every cell in the Verilog synthesis files there is a list of complete and correct CAT_STAT faults. By reading the detection table of every cell, the CAT faults are reported like CAT normal stuck (NS), CAT multi-port (MP), CAT multi-stuck (MS), CAT half-multi-stuck (HMS) and CAT full-multi-stuck (FMS). Figure 4.42 show the model for the cell DFFR_X1.
The script also introduces comments in the file (introduced by the character "#") in order to understand the type of CAT faults. The faults in the image are prime CAT faults.

```
 LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myDFFR_X1.spf.CTM")
sa1 NA DFFR_X1/Q D1 #MP
sa1 NA DFFR_X1/QN D1 #MP
sa0 NA DFFR_X1/Q D2 #MP full-MS
sa1 NA DFFR_X1/Q D2 #MP full-MS
sa1 NA DFFR_X1/QN D2 #MP full-MS
sa0 NA DFFR_X1/QN D2 #MP full-MS
sa0 NA DFFR_X1/Q D4 #MP full-MS
sa1 NA DFFR_X1/Q D4 #MP full-MS
sa1 NA DFFR_X1/QN D4 #MP full-MS
sa0 NA DFFR_X1/QN D4 #MP full-MS
sa0 NA DFFR_X1/Q D5 #MP half-MS
sa1 NA DFFR_X1/Q D5 #MP half-MS
sa1 NA DFFR_X1/QN D5 #MP half-MS
sa0 NA DFFR_X1/Q D6 #MP full-MS
sa1 NA DFFR_X1/Q D6 #MP full-MS
sa1 NA DFFR_X1/QN D6 #MP full-MS
sa0 NA DFFR_X1/QN D6 #MP full-MS
sa0 NA DFFR_X1/Q D14 #MP full-MS
sa1 NA DFFR_X1/Q D14 #MP full-MS
sa1 NA DFFR_X1/QN D14 #MP full-MS
sa0 NA DFFR_X1/QN D14 #MP full-MS
sa1 NA DFFR_X1/Q D22 #MP
sa0 NA DFFR_X1/QN D22 #MP
sa0 NA DFFR_X1/Q D52 #MP
sa1 NA DFFR_X1/QN D52 #MP
sa0 NA DFFR_X1/Q D131 #MP
sa1 NA DFFR_X1/QN D131 #MP
sa1 NA DFFR_X1/Q D173 #NS
sa1 NA DFFR_X1/QN D174 #NS
 LoadCTM ("/home/i.guglielminetti/prove_zoix/iacopoLibrary/myCTM/myDFFS_X1.spf.CTM")
sa1 NA DFFS_X1/Q D1 #MP
```

Figure 4.42: ScriptCtmStep1.sff example

defectsOnCells.py creates a model of the cell where only the prime defects are included,

instead defectsOnCells.sh creates a model of the cell where the prime and equivalent defects are included. defectsOnCells.py is also able to create the model for prime and equivalent CAT. ScriptCtmStep1.sff is not the only output of this script. The other files produced are staticDefectPerCell.txt, cellNameAndNumDefectOnPort.txt and cellNameAndNumDefectOnPortCOMPLETE.txt. staticDefectPerCell.txt has a debug and analysis purpose. Here there is presented a list of the cell model used in the design with the number of CAT defects after the CAT-NS, CAT-MP, CAT-MS, CAT-HMS and CAT-FMS has been created. For example, a possible list could be:

- 7 MUX2_X1
- 4 NAND2_X1
- 2 NAND2_X2
- ecc...

cellNameAndNumDefectOnPort.txt has a debug and analysis purpose. It contains a list with the name of the cells present in the design and the number of CAT defects on the outputs port. For the cells with one output only, the second parameter is always zero. This file is useful for understanding how in the cells with multiple outputs the CAT defects are distributed. An example of this list could be:

- CLKBUF_X2 2 0
- DFFR_X1 15 14
- DFFS_X1 15 15
- FA_X1 36 25
- INV_X16 2 0
- ecc...

cellNameAndNumDefectOnPortCOMPLETE.txt has a debug and analysis purpose. The list present in the file is divided in columns, the first is the name of the cell model, the second is the number of times that the cell is instantiated, and the third is the number of CAT_STAT present in the file before they are correctly represented like NS, MP, MS, HMS, FMS, and the last ones are the numbers of CAT_STAT defects present on the output ports, with the same rules used in the cellNameAndNumDefectOnPort.txt. An example of this file is reported in the figure 4.43.

```
AND2_X1 130 staticDefects: 3 ExpansedDefectsOnExitPort: 3 0
AND2_X2 5 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AND3_X1 14 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
AND4_X1 12 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AOI211_X1 8 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AOI21_X1 46 staticDefects: 3 ExpansedDefectsOnExitPort: 3 0
AOI21_X2 1 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AOI221_X1 133 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AOI221_X2 2 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
AOI222_X1 57 staticDefects: 5 ExpansedDefectsOnExitPort: 5 0
AOI22_X1 141 staticDefects: 3 ExpansedDefectsOnExitPort: 3 0
AOI22_X2 1 staticDefects: 4 ExpansedDefectsOnExitPort: 4 0
BUF_X1 2 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
BUF_X2 1 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
BUF_X4 4 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
CLKBUF_X1 100 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
CLKBUF_X2 1 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
DFFR_X1 483 staticDefects: 11 ExpansedDefectsOnExitPort: 15 14
DFFS_X1 8 staticDefects: 11 ExpansedDefectsOnExitPort: 15 15
FA_X1 96 staticDefects: 32 ExpansedDefectsOnExitPort: 36 25
INV_X16 2 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
INV_X1 845 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
INV_X2 3 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
INV_X4 2 staticDefects: 2 ExpansedDefectsOnExitPort: 2 0
```

Figure 4.43: cellNameAndNumDefectOnPort.txt example

To understand how these two scripts work the overall behaviour of the python version that works with prime faults is explained; the bash version that generates a list of prime and equivalent CAT faults have approximately the same behaviour. The first step consists in reading the list of the names of the cells in the design. For every cell creates a list of prime defects of all the 2 types CAT_STAT and CAT_DYN. Then the static detection table of the CTM file creates an internal model, then the count of the input and the output of the cell is performed in order to distinguish them from the defects. Then only the static defect are selected from the list of prime defects. For every CAT_STAT prime present in the file, the internal model of the detection table is read. When a defect could be found on the output/outputs, a set of flags are set. According to the flags sets, the CAT_STAT will be CAT-NS, CAT-MP, CAT-MS, CAT-HMS and CAT-FMS. Let m be the dimension of the static detection table and n the number of prime defects in the CTM file, the overall complexity will be O(m*n).

4.4.4 cellNameAndOutput.sh

This script reads the Verilog file containing descriptions of all library cells and creates a list that is saved in the file cellNameAndOutput.txt. This list consists in the cells name surrounded by “k” and the names of the output or outputs, an example could be:

- kDFFS_X1k .Q(.QN(
- kDFFS_X2k .Q(.QN(
- kDFF_X1k .Q(.QN(
- kDFF_X2k .Q(.QN(
- kDLH_X1k .Q(
- kDLH_X2k .Q(
- ecc...

This output file will be used by the following one in the flow to create the final version of the Z01X CAT faults list. The complexity of this algorithm is O(n).

4.4.5 moduleListGenerator.sh

This script reads the Verilog gate-level netlist and lists the name of the modules encountered in an output file named moduleList.txt. An example of this format could be:

- submodule1
- submodule2
- submodule3
- topmodule

This file will be used by cellFullIstanceAndOutput.py, the complexity of this algorithm is O(n).

4.4.6 cellFullIstanceAndOutput.py

This script has the duty to reconstruct the path of every cell in the gate-level netlist, this is important when the design has multiple modules inside. The output data is saved in two files, cellNameFullIstanceAndOutput.txt and cellFullIstanceAndOutput.txt, where the first will be used by the following algorithms in the flow. I report a picture of this file in the figure 4.44. As we can see, for every cell in the netlist there is the name of the cell of the library, the path created by the modules instances in which the cell is present, from the top module to the last one and the name/names of the output/outputs. This file will be used to inject the CAT faults on every cell.

```
DFFS_X1 clock_module_0/sync_reset_por/data_sync_reg_0_ .Q( .QN(
DFFS_X1 clock_module_0/sync_reset_por/data_sync_reg_1_ .Q( .QN(
INV_X1 clock_module_0/sync_reset_por/U3 .ZN(
INV_X1 clock_module_0/scan_mux_por/U1 .ZN(
AOI22_X1 clock_module_0/scan_mux_por/U2 .ZN(
INV_X1 clock_module_0/scan_mux_por/U3 .ZN(
INV_X1 clock_module_0/scan_mux_puc_rst_a/U1 .ZN(
AOI22_X1 clock_module_0/scan_mux_puc_rst_a/U2 .ZN(
INV_X1 clock_module_0/scan_mux_puc_rst_a/U3 .ZN(
DFFR_X1 clock_module_0/sync_cell_puc/data_sync_reg_0_ .Q( .QN(
DFFR_X1 clock_module_0/sync_cell_puc/data_sync_reg_1_ .Q( .QN(
INV_X1 clock_module_0/sync_cell_puc/U3 .ZN(
INV_X1 clock_module_0/scan_mux_puc_rst/U1 .ZN(
AOI22_X1 clock_module_0/scan_mux_puc_rst/U2 .ZN(
INV_X1 clock_module_0/scan_mux_puc_rst/U3 .ZN(
INV_X1 clock_module_0/U19 .ZN(
INV_X1 clock_module_0/U21 .ZN(
INV_X1 clock_module_0/U23 .ZN(
INV_X1 clock_module_0/U25 .ZN(
INV_X1 clock_module_0/U27 .ZN(
INV_X1 clock_module_0/U29 .ZN(
INV_X1 clock_module_0/U31 .ZN(
INV_X1 clock_module_0/U33 .ZN(
INV_X1 clock_module_0/U35 .ZN(
INV_X1 clock_module_0/U37 .ZN(
INV_X1 clock_module_0/U39 .ZN(
INV_X1 clock_module_0/U41 .ZN(
```

Figure 4.44: cellNameFullIstanceAndOutput.txt example

The first step of this script reads the file with the module names (moduleList.txt) and creating an internal list. Then the design file is read in order to create a list of the modules and how and where they are instantiated. An example of this list could be:

- module1
- module2
- module3
- module1 istance1
- module2 istance2
- topmodule
- module3 istance3

This means that module1 and module2 don't have submodules, module3 has module1 and module2 as submodules and topmodule has module3 as submodule. Then the program read cellNameAndOutput.txt to create a list of names of the cells in the library. The algorithm reads again the netlist file, if it confirms that it encounters a cell by

checking the name with the list of names from the library, it starts to reconstruct the path to it. If the cell is inside the top module, there is nothing to do. If not, the program searches in the list of modules and instances created before. For example, it needs to reconstruct the path of a cell inside the module1 named register1. Reading the list, the program noticed that module1 is instantiated inside module3. So the program writes istance1/register1 in the path string and starts again but this time searching for module3. Then it is found that module3 is inside the topmodule. My program obviously knows the name of the topmodule of the design (passed by the command line) so it creates the path istance3/istance1/register1 and then proceeds with another cell. What happens if there are modules instantiated more than 1 time? For example, the list of modules could be:

- module1
- module2
- module3
- module4
- module1 istance1
- module2 istance2
- module5
- module1 istance1
- module3 istance3
- topmodule
- module4 istance4
- module5 istance5

In this case, module1 is instantiated 2 times. My script could handle this situation and create the correct number of paths, but this feature is not yet tested. The list of modules and instances is read more times for every cell. Fortunately, it has very few data in it, so if the number of cells in the design is n, the total complexity of this script is approximately $O(n \log_n)$.

4.4.7 defectsOnIstance.py

This script creates the first CAT fault list for Z01X that is complete and could be read by the fault simulator. The output file of this script is ScriptCtmStep2.sff. An example of it is in the figure 4.41. The input files needed by it are cellNameFullIstanceAndOutput.txt and ScriptCtmStep1.sff. For every instance in the first file, it reads the model of the corresponding library cell with all the CAT_STAT faults for ZOIX in order to create the

complete list of CAT faults for every cell. The results do not take into account if the cell with multiple output ports has only 1 output instantiated into the design. For example, imagine that there is a cell in the netlist with this form:
DFFR_X1 mycell (.D(d), .CK(clk), .RN(rn), .QN(qn)); in the file ScriptCtmStep2.sff there also will be all the defects on the output port Q. This "error" will be corrected by others scripts, but they are not part of CellAwareFaultGenerator.sh, they need to be launched after it.

4.4.8 CellAwareFaultGeneratorStepTestbench.sh

This script is optional, it will be used depending on the input command that the user wrote to CellAwareFaultGenerator.sh. This algorithm appends the optional string written at the command line before the path of the CAT faults in the file ScriptCtmStep2.sff. The complete CAT faults list for Z01X will be written in the file ScriptCtmTestbench.sff.

4.4.9 checkTotalNumberOfDefects.py

This is an additional script placed at the end of the CAT fault generator in order to check if everything works and if the number of faults in the output fault list is correct. Depending on the parameters given to CellAwareFaultGenerator.sh, it checks ScriptCtmStep2.sff or ScriptCtmTestbench.sff. If an error occurs, the messages are display on the console. Aside from the fault list, the other input files needed by this script are cellNameFullIstanceAndOutput.txt, ScriptCtmStep1.sff and listNameCellsInDesign.txt. But how does this algorithm work? For every cell in cellNameFullIstanceAndOutput.txt, it reads the path and memorizes it, then using ScriptCtmStep1.sff the program calculates the correct number of faults that should be in the fault list for that cell. Using the path memorized, it checks if in the fault list there is the correct number of faults for that cell. If this is not true, an error appears on the console. After this script finishes, the CAT fault list for Z01X is not ready because when Z01X is launched with ScriptCtmStep2.sff or ScriptCtmTestbench.sff it will throw an error in the file fr2fdef.log. More precisely it presents a warning with a list of untestable faults that need to be removed, otherwise, Z01X will not start to simulate. In the picture 4.46 is showed an example of this error.

```
--------------------------------------------------------------------------------
NA 0 {PORT "openMSP430.mem_backbone_0.per_dout_val_reg_13_.Q"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25044)
--------------------------------------------------------------------------------
NA 0 {PORT "openMSP430.mem_backbone_0.per_dout_val_reg_13_.Q"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25046)
--------------------------------------------------------------------------------
NA 1 {PORT "openMSP430.mem_backbone_0.per_dout_val_reg_13_.Q"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25048)
--------------------------------------------------------------------------------
NA 0 {PORT "openMSP430.mem_backbone_0.U3.ZN"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25050)
--------------------------------------------------------------------------------
NA 1 {PORT "openMSP430.mem_backbone_0.U3.ZN"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25051)
--------------------------------------------------------------------------------
NA 0 {PORT "openMSP430.mem_backbone_0.U5.ZN"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25052)
--------------------------------------------------------------------------------
NA 1 {PORT "openMSP430.mem_backbone_0.U5.ZN"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25053)
--------------------------------------------------------------------------------
NA 0 {PORT "openMSP430.mem_backbone_0.U7.ZN"} (file: __globfiles__/ScriptCtmTestbench.sff, line: 25054)
```

Figure 4.45: fr2fdef.log example

CellAwareFaultGenerator.sh scripts create all the CAT faults possible in the netlist. As already explained, if we have a cell DFFR_X1 instantiated in the design with only 1

output port (Q for example), my fault generator will create all the faults with both the output ports (Q and QN). So Z01X recognize that the port QN does not exist and lists all the faults on that port as untestable, so they have to be removed. The second reason is that some of the CAT faults in a design could be untestable by nature, the output exists and the CAT faults are correct, but due to the structure of the circuit these faults are untestable and they need to be deleted from the fault list. To create a final and correct version of the CAT fault list for Z01X I made a script named deleteUnusefullPort.py.

4.4.10 deleteUnusefullPort.py

This script deletes the untestable faults in the CAT fault list for Z01X. The output is a file with a list of CAT faults perfectly readable by Z01X. In input, it requires the original file with the defect list and the file fr2fdef.log created by Z01X. This script reads the faults listed in the Z01X log file and deletes the row in which they are present in the original fault list, creating a new correct one. An example of command needed by this script to run is:

- python deleteUnusefullPort.py /path/ScriptCtmTestbench.sff /path/runZoix/fr2fdef.log

The output will be a file with the name of the original one plus the string PURIFIEDpy, for example after the command that I showed, the output will be in ScriptCtmTestbenchPURIFIEDpy.sff.

4.5 Z01X CAT results

Due to the particular fault list format required by Z01X, the fault simulation results produced by Z01X need some modifications and the fault coverage created by the tool needs to be recalculated. The scripts created to recalculate the fault coverage and to "compress" the fault list in a more ordered shape are presented in the following subsections, but first the formulas used are reported by Z01X and TMAX to calculate the fault coverage and the test coverage.

4.5.1 Z01X and TMAX formulas

In the implementation proposed for this thesis the test coverage and the fault coverage of the CAT faults always coincide. The tools used to perform the CAT fault simulation, Z01X, performs a testability analysis before proceeding with the simulations. However the number of testable faults and the total number of faults present in the UUT is the same inside the fault simulations. According to Z01X, the formulas to calculate the CAT test coverage and fault coverage are:

$$Testcoverage(\%) = \frac{CAT_{DD} + \frac{1}{2}CAT_{PD}}{CAT_{testableFaults}}100;$$

$$Faultcoverage(\%) = \frac{CAT_{DD} + \frac{1}{2}CAT_{PD}}{CAT_{totalFaults}}100;$$

In chapter 5, the CAT results are compared with SAF simulation results performed with Z01X and TMAX. In this case, the formula of the SAF test coverage used by the two fault simulators are:

$$TestcoverageZ01X(\%) = \frac{SAF_{DD} + \frac{1}{2}SAF_{PD}}{SAF_{totalFaults} - SAF_{UG}}100;$$

$$TestcoverageTMAX(\%) = \frac{SAF_{DT} + \frac{1}{2}SAF_{PT}}{SAF_{totalFaults} - SAF_{UD}}100;$$

4.5.2 ricalcolaCoverageV2.py

The faults in the CAT fault list can be repeated one, two, three or four times according to the rules of the correct representation of CAT faults for Z01X reported in the section 4.3. After the simulation, the results of these representations can be different and they need to be compacted into only one type. "ricalcolaCoverageV2.py" transform the results of the multi-representation in only one type for every fault in the fault simulation result. Z01X saves the simulation results in a file name sim.rpt. The command needed to launch the script is:

- python ricalcolaCoverageV2.py /path/to/sim.rpt newFileName /path/to/store/the/Result

The faults are collapsed with a certain hierarchy, that is the following:

- if there is at least one DD, the other types of faults are not considered in the fault coverage. If the DDs are equal to a number n, n-1 DD are removed from the final fault coverage;
- if there are 0 DD and at least one ND, the other types of faults are not considered in the fault coverage. If the ND are equal to a number n, n-1 ND are removed from the final fault coverage;
- if there are 0 DD, 0 ND and at least one NI, the other types of faults are not considered in the fault coverage. If the NI are equal to a number n, n-1 NI are removed from the final fault coverage;
- if there are 0 DD, 0 ND, 0 NI and at least one NC, the other types of faults are not considered in the fault coverage. If the NC are equal to a number n, n-1 NC are removed from the final fault coverage;
- if there are 0 DD, 0 ND, 0 NI, 0 NC and at least one NO, the other types of faults are not considered in the fault coverage. If the NO are equal to a number n, n-1 NO are removed from the final fault coverage;

- if there are 0 DD, 0 ND, 0 NI, 0 NC, 0 NO and at least one NS, the other types of faults are not considered in the fault coverage. If the NS are equal to a number n, n-1 NS are removed from the final fault coverage;
- if there are 0 DD, 0 ND, 0 NI, 0 NC, 0 NO, 0 NS and at least one PD, the other types of faults are not considered in the fault coverage. If the PD are equal to a number n, n-1 PD are removed from the final fault coverage;
- if there are 0 DD, 0 ND, 0 NI, 0 NC, 0 NO, 0 NS, 0 PD and at least one HA, the other types of faults are not considered in the fault coverage. If the HA are equal to a number n, n-1 NS are removed from the final fault coverage;
- if there are 0 DD, 0 ND, 0 NI, 0 NC, 0 NO, 0 NS, 0 PD, 0 HA and at least one IA, the other types of faults are not considered in the fault coverage. If the IA are equal to a number n, n-1 IA are removed from the final fault coverage.

In the file sim.rpt Z01X creates a summary like the one reported in the figure 4.46, the scripts add the number of faults that need to be deleted from the fault coverage.

```
#-------------------------------------------------------------------------
# Fault Coverage Summary
#
#                                       Prime             Total
#-------------------------------------------------------------------------
# Total Faults:                         36821             36821
#
# Detected                        DG     7584  20.60%      7584  20.60%
#   Dropped Detected                DD     7584  20.60%      7584  20.60%
# Illegal                         IG    26083  70.84%     26083  70.84%
#   Illegal Access                  IA    26083  70.84%     26083  70.84%
# Unselected                      NG     2772   7.53%      2772   7.53%
#   Not Controlled                  NC      302   0.82%       302   0.82%
#   Not Injected                    NI     2109   5.73%      2109   5.73%
#   Not Observed                    NO      181   0.49%       181   0.49%
#   Not Strobed                     NS      180   0.49%       180   0.49%
# Potential                       PG       19   0.05%        19   0.05%
#   Dropped Potential               PD       19   0.05%        19   0.05%
# Not Detected                    ND      363   0.99%       363   0.99%
#
# Test Coverage                                20.62%            20.62%
# Fault Coverage                               20.62%            20.62%
#-------------------------------------------------------------------------
The DD that need to be removed are: 2051
The ND that need to be removed are: 324
The NI that need to be removed are: 774
The NC that need to be removed are: 65
The NO that need to be removed are: 4
The NS that need to be removed are: 81
The PD that need to be removed are: 0
The HA that need to be removed are: 0
The IA that need to be removed are: 6929
If everithing work, you schould have in total: 26593
```

Figure 4.46: ricalcolaCoverageV2.py output example

An example of the script behaviour is shown with the following example. The fault simulation has produced the results:

- sa0 ND stato0reg000/Q D2

- sa1 DD stato0reg000/Q D2
- sa1 NI stato0reg000/QN D2
- sa0 NC stato0reg000/QN D2

D2 is only 1 CAT fault, so it is compressed. Taking into account the hierarchy, it is a DD and from the fault, coverage need to be removed 1 ND, 1 NC and 1 NI.

4.5.3 compressione_CAT_ZOIX.py

This script is useful for viewing the results of every CAT fault in the fault simulation. In the sim.rpt file produced by Z01X there is a list of simulated faults, like the ones listed below:

- sa0 ND r7_reg_2_/Q D1
- sa1 DD r7_reg_2_/Q D1
- sa1 NI r7_reg_2_/QN D1

compressione_CAT_ZOIX.py recognizes that are the same type of fault and create a single row with the format: sa0_sa1_sa1 ND_DD_NI r7_reg_2_/Q_Q_QN D1.

5 Results

This chapter presents the results of the developed CAT flow, which uses Z01X fault simulator, described in the implementation chapter. In the first tests, the faults simulations are performed on a small circuit to see if the Z01X results are correct. Then it is tested if Z01X works correctly with the test stimuli generated by TMAX CAT ATPG. This tool was used to create functional test patterns for different gate-level netlists of circuits with variable complexity and the fault simulations results were recorded. The netlists were also modified with DfT logic (scan chains), and the same testing flow was applied to them. On the same circuits were also applied functional and scan random patterns. The test stimuli were created using LFSRs. The faults simulations of these patterns were performed with the SAF model and the developed CAT flow.
At last, Z01X was used to simulate functional patterns on a CPU of a microcontroller named openMSP430. These patterns were given in input to Z01X with different standard formats. The nangate library was used to synthetize all the UUTs.

5.1 Z01X results validation

This thesis aims to create a testing flow able to work with the CAT fault model and test any possible ICs and SOCs described at the gate-level netlist. Reporting only the results obtained on real netlist such the ones in the section 5.6 is not enough. The first step consists in testing the CAT flow on a small circuit and seeing if given some test stimuli, the CAT faults covered by them are the expected ones.
A gate-level netlist named simpleCircuit.v has been developed by hand, whose structure is reported in figure 5.1.

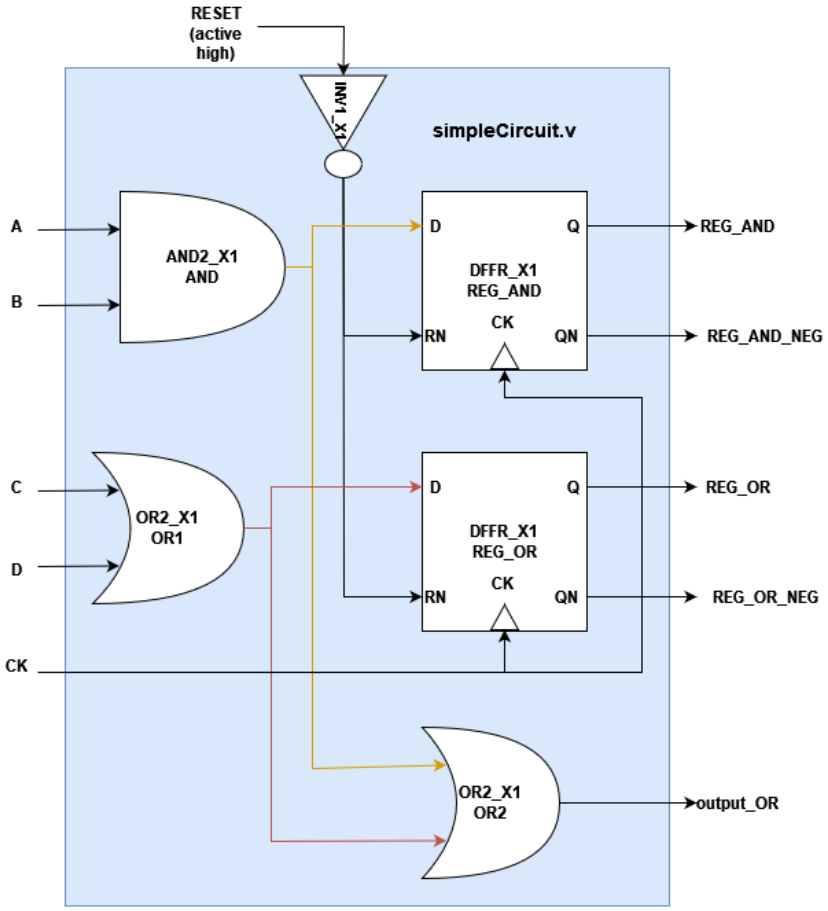


Figure 5.1: simpleCircuit gate-level netlist

The inputs of the circuit are A, B, C, D, CK (which is the clock) and RESET (which is the reset, active high). The outputs are REG_AND, REG_AND_NEG, REG_OR, REG_OR_NEG and output_OR. The test patterns applied on simpleCircuit are two and are reported in the table below.

A	B	C	D	RESET
1	0	0	0	0
1	1	0	1	0
REG_AND	REG_AND_NEG	REG_OR	REG_OR_NEG	output_OR
0	1	0	1	0
1	0	1	0	1

Table 5.1: Input configuration and expected outputs

The first pattern is repeated for three clock cycles, and the second pattern is repeated

for two clock cycles. The static detection tables present in the CTM files of the cells inside the netlist are reported in the tables below.

A1	A2	ZN	D4	D7	D20
0	0	0	0	0	1
0	1	0	0	1	1
1	0	0	0	0	1
1	1	1	1	0	0

A1	A2	ZN	D2	D6	D20	D30	D34
0	0	0	0	0	1	0	0
0	1	1	0	1	0	0	1
1	0	1	0	1	0	1	0
1	1	1	1	1	0	0	0

Table 5.2: CAT model of AND_X1 on the left. CAT model of OR_X1 on the right.

A	ZN	D3	D15
0	1	1	0
1	0	0	1

Table 5.3: CAT model of INV_X1

The static detection table of the last cell DFFR_X1 was already reported in the figure 4.12. The test patterns could be represented with different standards, so they have been used to create three test flows on Z01X:

- In the first Modelsim, a tool developed by Mentor Graphics [7], was used to simulate the test patterns and create an ecvd file. The test stimuli were written by hand inside a testbench, simpleCircuit_tb.v. Then the evcd was given in input to Z01X and the fault simulation results were collected;
- In the second, the ecvd created by Modelsim was given in input to TMAX which creates a STIL file with the test patterns. In particular, in the STIL file, there are five patterns, where three represent the first input configuration repeated three times and two represent the second one. The STIL was given in input to Z01X and the fault simulation results were collected;
- In the third, the testbench simpleCircuit_tb.v was given directly in input to Z01X and the fault simulation results were collected.

For example, I report the ecvd test stimuli read by Z01X in the figure 5.2.

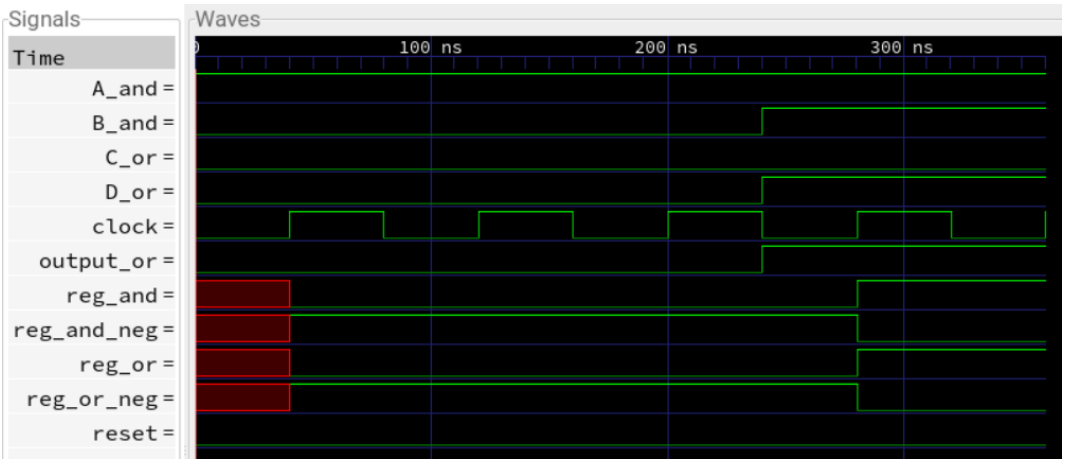


Figure 5.2: evcd patterns

According to the CTM files in the netlist 37 CAT_STAT prime faults are present. In particular, checking by hand the input values given to every cell, from the Z01X fault simulation is expected to detect the following CAT_STAT faults:

- For the cell named AND: D4 and D20;
- For the cell named OR1: D6, D20 and D34;
- For the cell named OR2: D2, D6 and D20;
- For the cell named INV: D3;
- For the cell named REG_AND: D1, D2, D4, D5, D6, D14, D22, D52, D173 and D174;
- For the cell named REG_OR: D1, D2, D4, D5, D6, D14, D22, D52, D173 and D174.

In total the expected detected faults are 29.
The results of the three tests are summarised in the table 5.4.

Test name	DD	NI	NC	IA
Test1 (evcd)	24	11	1	1
Test2 (STIL)	25	11	1	0
Test3 (TESTBENCH)	25	11	1	0

Table 5.4: Fault simulations results

In the Test1, fault D3 in the cell INV was not detected, in particular, it is marked in the output fault list as illegal access (IA). The other 4 faults that the fault simulation

was not able to cover were D22, D173 of the cell REG_AND and D22, D173 of the cell REG_OR.
The results of the tests Test2 and Test3 were the same. These times the only faults not covered by the fault simulations were 4, D22, D173 of the cell REG_AND and D22, D173 of the cell REG_OR. All three tests fail to cover the same four CAT_STAT faults. Looking at the static detection table of the cells of type DFFR_X1 in the figure 4.12 and to the test stimuli in the figure 5.2 the patterns simulated on the two cells of type DFFR_X1 named REG_OR and REG_AND are the first two of the detection table. The Z01X results show that according to the fault simulator only the second pattern is present in the fault simulation because it is not able to detect D173 and D22. To check if the first pattern is present during the simulation, a test logic was inserted in the netlist of simpleCircuit. In particular, this test logic has an output pin that turns to one when the conditions of the first test pattern in the static detection table in the figure 4.12 are satisfied. The modified netlist of simpleCircuit is reported in the figure 5.3.

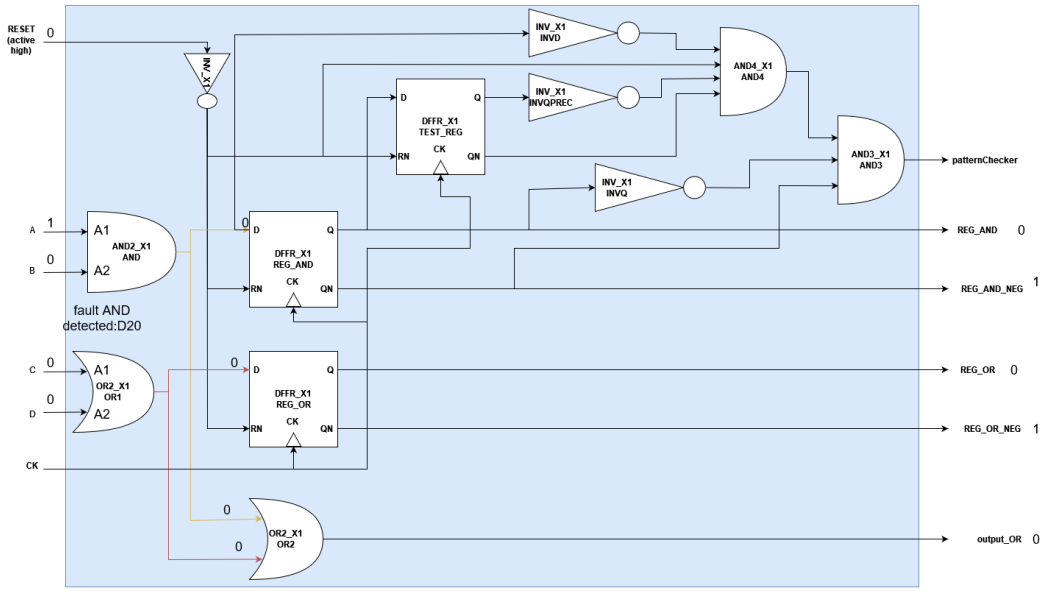


Figure 5.3: Modified simpleCircuit netlist

In particular, the condition of the first pattern in the static detection table of the cell REG_AND of type DFFR_X1 are:

- Clock port CK at the active state (P);
- reset port RN equal to 1;
- input port D equal to 0;
- output port Q equal to 0;
- output port QN equal to 1;

- precedent state on the output port Q equal to 0;
- precedent state on the output port QN equal to 1.

By simulating this version of simpleCircuit with the testlogic on Modelsim, the simulations results were the ones reports in the figure 5.4.

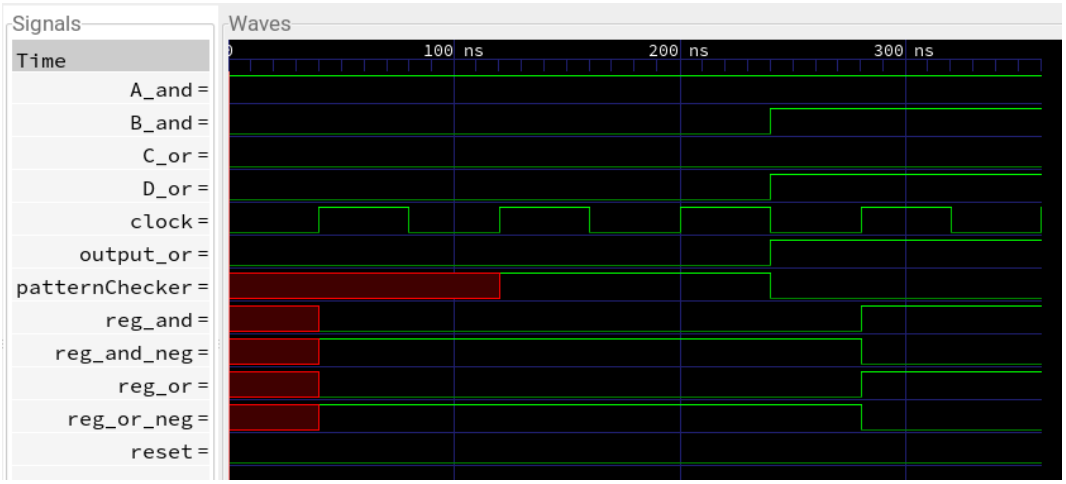


Figure 5.4: Modelsim simulation results

As it is possible to see, the patternChecker pin becomes one for at least one clock cycle during the simulation.

5.2 Z01X and TMAX ATPG patterns tests

The fault simulation performed in the section 5.3 uses for test stimuli the input patterns created by TMAX CAT ATPG. Before proceeding with the presentation of those results, it is important to understand if Z01X produces the same fault coverage results of TMAX CAT ATPG.
The comparison between TMAX and Z01X results is made on the simple nangate cell FA_X1 instantiated like for the precedent example in the figure 4.36. The test stimuli to apply on the UUT were made with the TMAX CAT ATPG flow already shown in the figure 4.18. Using the developed script CellAwareFaultGenerator.sh the list of CAT faults for Z01X was created and the flow used to fault simulate the TMAX pattern is reported in the figure 5.5.

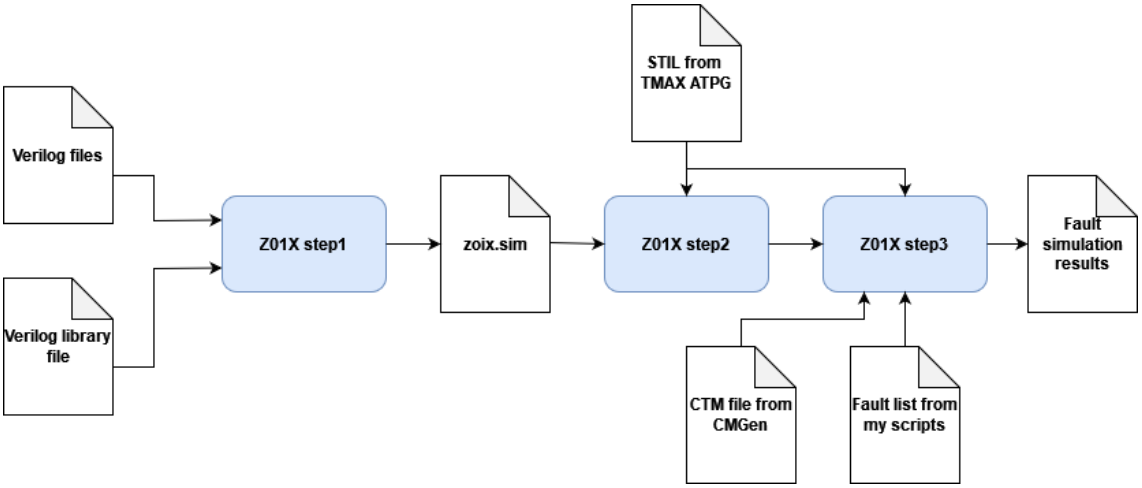


Figure 5.5: Z01X flow for TMAX CAT ATPG patterns

TMAX ATPG, in order to cover all the faults in the fault list, creates 8 patterns in a STIL file. The CTM file that was given in input to TMAX was reversed like in the one already showed in the figure 4.22, in order to make the TMAX and Z01X results compatible. The 8 patterns created by TMAX were manually split in 8 different STIL files. Then the files were fault simulated with Z01X to see if the correct CAT_STAT faults were detected by them. The patterns are summarised in the table 5.5.

Pattern number	A	B	CI	CO	S
1	0	0	0	0	0
2	0	0	1	0	1
3	0	1	0	0	1
4	0	1	1	1	0
5	1	0	0	0	1
6	1	0	1	1	0
7	1	1	0	1	0
8	1	1	1	1	1

Table 5.5: TMAX patterns

The faults detected by each pattern are reported in the list below. According to the CTM file of FA_X1, the results are correct.

- Pattern 1: D42, D103, D108;
- Pattern 2: D2, D18, D42, D55, D58, D63, D70;
- Pattern 3: D2, D4, D5, D16, D42, D66, D96;

- Pattern 4: D3, D6, D13, D18, D57, D58, D59, D89, D92, D102, D104, D108;
- Pattern 5: D2, D5, D13, D16, D42, D66, D76, D82;
- Pattern 6: D4, D6, D14, D18, D57, D58, D59, D77, D84, D104, D108;
- Pattern 7: D3, D6, D13, D18, D75, D77, D89, D92, D98, D104, D108;
- Pattern 8: D2, D3, D4, D5, D6, D7, D13, D14, D16, D18, D54, D55, D58, D59, D75, D76, D89.

5.3 Results on the ITC'99 benchmark circuits

The developed CAT fault simulation flow has been tested on a suite of RT-level benchmarks. The circuits specification were written in VHDL, mainly in behavioural code but also with structural parts. They were presented in the paper [2]. The RTL-code of 15 circuits inside the benchmarks were taken into account, from now they will be called "Bcircuits". They perform different functions and tasks, also the amount of area and library cells that are needed by them to be synthesized is different. The logic synthesis has been performed using the nangate library. The gate-level netlist of the Bcircuits has been used by the TMAX CAT ATPG to create the test stimuli that will be used by Z01X. The TMAX flow was the same presented in the figure 4.18. The comparison of Z01X and TMAX results has the purpose of checking the Z01X results with the ones produced by TMAX CAT ATPG. The testing flow used by Z01X to perform the faults simulation on the Bcircuits was the one shown in the picture 5.5. The test stimuli saved in a STIL file and created by TMAX are functional test patterns, in fact, the synthesized gate-level netlists of the Bcircuits have no DfT logic inside. The results of the TMAX CAT ATPG are shown in the table 5.6, the classification of faults assigned to the CAT defect by the ATPG is explained in the section 2.4. The test coverage shown in this table was calculated using the formula presented in the section 4.5.1.

Circuit name	CAT_STAT faults	DT	PT	UD	AU	ND	Test coverage
B02	93	92	1	0	0	0	99.46 %
B03	604	476	1	9	110	8	80.08 %
B04	1699	1610	2	0	71	16	94.82 %
B05	1709	161	2	46	1134	366	9.74 %
B06	211	208	1	0	1	1	98.82 %
B07	1428	113	2	2	646	665	7.99 %
B08	558	535	1	0	6	16	95.97 %
B09	612	583	1	0	3	25	95.34 %
B10	538	504	1	4	24	5	94.48 %
B11	1414	1248	1	0	53	112	88.30 %
B12	3764	775	5	45	282	2657	20.91 %
B13	1065	435	1	6	99	524	41.12 %
B14	12162	11572	1	0	221	368	95.15 %
B15	21669	2201	5	34	85	19344	10.18 %

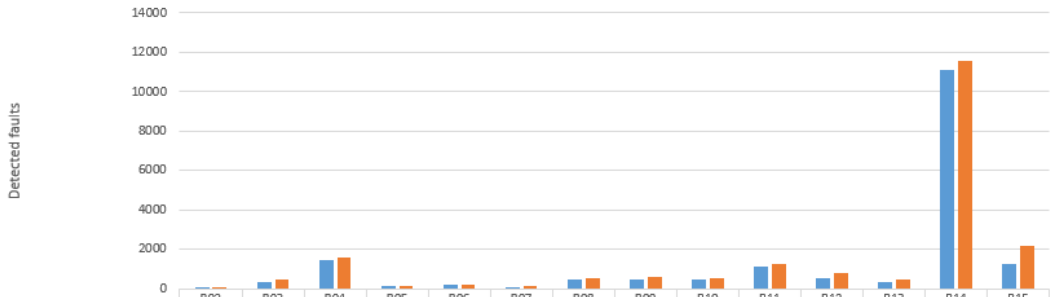
Table 5.6: TMAX CAT_STAT ATPG results

The results of the Z01X CAT fault simulation are shown in the table 5.7, the classification of faults assigned to the CAT defect by the fault simulator is explained in the section 2.4. The test coverage shown in this table was calculated using the formula presented in the section 4.5.1.

Circuit name	CAT_STAT faults	DD	ND	NI	NC	NO	Test coverage
B02	93	84	5	4	0	0	90.32 %
B03	604	358	68	117	61	0	59.27 %
B04	1699	1437	126	136	0	0	84.57 %
B05	1709	118	713	459	288	131	6.90 %
B06	211	194	6	11	0	0	91.94 %
B07	1428	37	4	84	77	1226	2.59 %
B08	558	480	44	34	0	0	86.02 %
B09	612	489	81	42	0	0	79.90 %
B10	538	470	46	22	0	0	87.36 %
B11	1414	1135	153	88	30	8	80.26 %
B12	3764	515	503	516	367	1863	13.68 %
B13	1065	331	78	99	37	520	31.07 %
B14	12162	11080	813	269	0	0	91.10 %
B15	21669	1221	1304	1772	1550	15822	5.63 %

Table 5.7: Z01X CAT_STAT fault simulations results

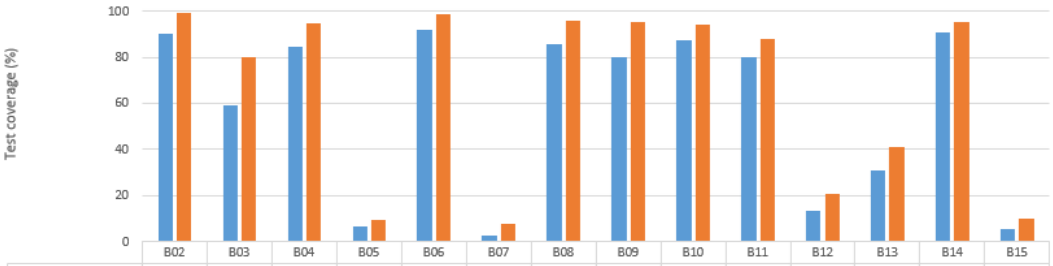
In the figure 5.6 are shown the Z01X DD faults in comparison with TMAX DT faults.



	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13	B14	B15
Z01X CAT_STAT DD faults	84	358	1437	118	194	37	480	489	470	1135	515	331	11080	1221
TMAX CAT_STAT DT faults	92	476	1610	161	208	113	535	583	504	1248	775	435	11572	2201

Figure 5.6: Z01X DD and TMAX DT comparisons

The Z01X test coverage in comparison with TMAX test coverage are shown in figure 5.7.



	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13	B14	B15
Z01X test coverage	90,32	59,27	84,57	6,9	91,94	2,59	86,02	79,9	87,36	80,26	13,68	31,07	91,1	5,63
TMAX test coverage	99,46	80,08	94,82	9,74	98,82	7,99	95,97	95,34	94,48	88,3	20,91	41,12	95,15	10,18

Figure 5.7: Z01X and TMAX test coverage comparisons

5.4 Results on the ITC'99 benchmark circuits with scan logic

The Bcircuits, after being tested with functional patterns, were modified by inserting some DfT logic i.e., scan chains. In the following tests, the netlists have additional pins for the scan chain inputs and scan chain outputs. The gate-level netlists with the scan chains of the Bcircuits have been used by the TMAX CAT ATPG to create the test stimuli that are then used by Z01X. The TMAX flow is presented in the figure 5.8.

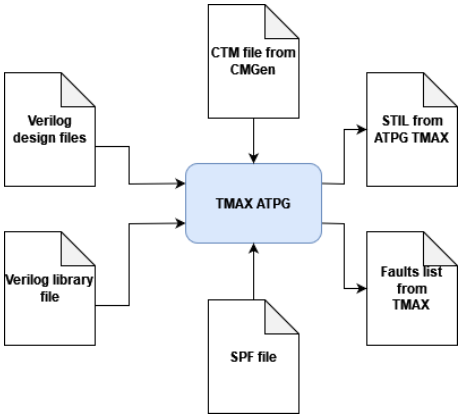


Figure 5.8: TMAX ATPG flow with scan chain circuits

The testing flow used by Z01X to perform the fault simulations on the Bcircuits was the one shown in the picture 5.5. The test stimuli saved in a STIL file and created by TMAX are can test patterns. The results of the TMAX CAT ATPG are shown in the table 5.8, the classification of faults assigned to the CAT defect by the ATPG is explained in the section 2.4. The test coverage shown in this table was calculated using the formula presented in the section 4.5.1.

Circuit name	CAT_STAT faults	DT	PT	UD	AU	ND	Test coverage
B02_scan	125	121	0	0	1	3	96.8 %
B03_scan	851	821	0	16	1	13	98.32 %
B04_scan	2234	2180	0	23	0	31	98.60 %
B05_scan	1983	1909	0	52	0	22	96.86 %
B06_scan	275	272	0	0	0	3	98.91 %
B07_scan	1785	1750	0	16	0	19	98.93 %
B08_scan	726	715	0	6	0	5	99.31 %
B09_scan	844	823	0	7	0	14	98.33 %
B10_scan	674	659	0	9	0	6	99.10 %
B11_scan	1654	1627	0	6	0	21	98.73 %
B12_scan	4720	4641	0	41	2	36	99.19 %
B13_scan	1426	1396	0	7	1	22	98.38 %
B14_scan	13889	13745	0	43	1	100	99.27 %
B15_scan	25001	24309	0	453	0	239	99.03 %

Table 5.8: TMAX CAT_STAT ATPG results

The results of the Z01X CAT fault simulations are shown in the table 5.9, the classification of faults assigned to the CAT defect by the fault simulator is explained in the section 2.4. The test coverage shown in this table was calculated using the formula presented

in the section 4.5.1.

Circuit name	CAT_STAT faults	DD	ND	NI	NC	NO	Test coverage
B02_scan	125	119	5	0	1	0	95.20 %
B03_scan	851	826	23	0	2	0	97.06 %
B04_scan	2234	2161	69	1	3	0	96.73 %
B05_scan	1983	1899	80	2	2	0	95.76 %
B06_scan	275	269	5	0	1	0	97.81 %
B07_scan	1785	1730	53	0	2	0	96.91 %
B08_scan	726	683	40	2	1	0	94.07 %
B09_scan	844	830	12	0	2	0	98.34 %
B10_scan	674	643	29	1	1	0	95.40 %
B11_scan	1654	1622	26	4	2	0	98.06 %
B12_scan	4720	4446	228	40	6	0	94.19 %
B13_scan	1426	1372	51	1	2	0	96.21 %
B14_scan	13889	13519	344	17	9	0	97.33 %
B15_scan	25001	23959	696	327	19	0	95.83 %

Table 5.9: Z01X CAT_STAT fault simulations results

In figure 5.9 the Z01X DD faults are shown in comparison with TMAX DT faults.

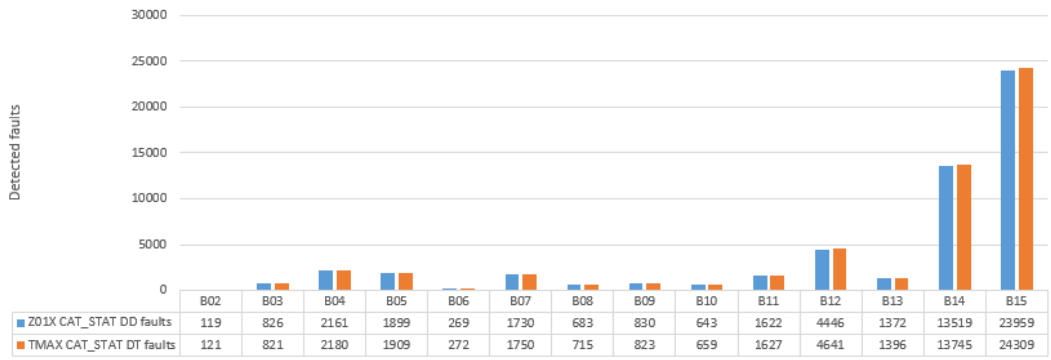
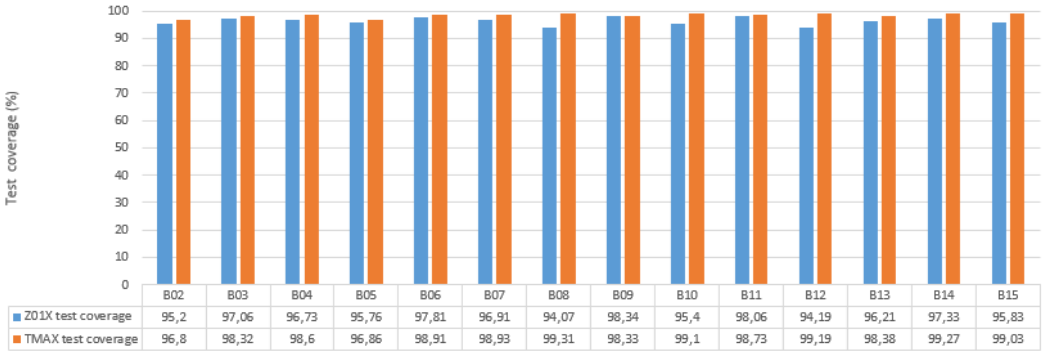


Figure 5.9: Z01X and TMAX fault coverage comparisons

In figure 5.10 the Z01X test coverage are shown in comparison with TMAX test coverage.



	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13	B14	B15
Z01X test coverage	95,2	97,06	96,73	95,76	97,81	96,91	94,07	98,34	95,4	98,06	94,19	96,21	97,33	95,83
TMAX test coverage	96,8	98,32	98,6	96,86	98,91	98,93	99,31	98,33	99,1	98,73	99,19	98,38	99,27	99,03

Figure 5.10: Z01X and TMAX test coverage comparisons

5.5 CAT and SAF comparison on ITC'99 benchmark circuits

The developed Z01X CAT testing flow is able to simulate the prime and equivalent CAT_STAT faults in the CTM files created by CMGen. TMAX does not consider equivalent faults in the coverage computation, so it is impossible to make a comparison between the two faults simulators. Taking into account that, it was decided to compare the test coverage of the CAT_STAT faults with the test coverage of the SAFs.
The gate-level netlists of the Bcircuits are the same used in the section 5.3 and 5.4. To create the test stimuli, at the inputs of the circuits, were used LFSRs. The resulting test patterns are random functional patterns for the Bcircuits without scan chains and random scan chain patterns for the Bcircuits with scan chains. The testing flow used to collect the results is reported in figure 5.11. In particular, the testbench of every circuit is directly simulated with the LFSRs inside Z01X and the resulting test stimuli are used for the fault simulations.

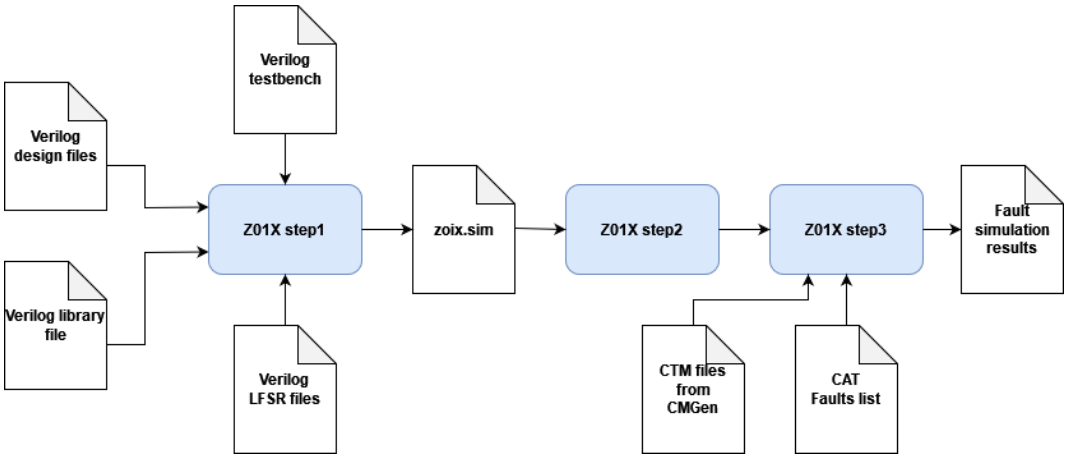


Figure 5.11: Z01X flow with testbench stimuli

The final Z01X results regarding the CAT_STAT prime and equivalent faults are reported

in the table 5.10. The Bcircuits do not have scan chains.

C. name	CAT_STAT	DD	ND	NI	NC	NO	NS	PD	HA
B02	644	580	56	4	0	0	0	4	0
B03	4550	2863	515	570	598	0	0	4	0
B04	11538	10136	668	673	57	0	0	4	0
B05	9911	5597	3555	570	175	0	0	14	0
B06	1356	1330	6	20	0	0	0	0	0
B07	8458	4151	2162	916	1064	161	0	4	0
B08	3826	3577	189	48	12	0	0	0	0
B09	4268	3472	657	139	0	0	0	0	0
B10	3561	3407	78	72	0	0	0	4	0
B11	7948	7652	201	91	0	0	0	4	0
B12	24444	4635	2966	2297	2475	12067	0	4	0
B13	7396	5598	823	639	332	0	0	4	0
B14	65305	60513	3599	792	244	0	134	8	15
B15	127171	17425	30897	11764	12622	54120	329	14	0

Table 5.10: Z01X CAT_STAT fault simulations results

The final Z01X results regarding the SAFs are reported in the table 5.11. The Bcircuits do not have scan chains.

C. name	SAF	DD	ND	NI	NC	NO	NS	PD	HA	UG
B02	128	122	2	0	0	0	0	4	0	0
B03	798	536	114	0	38	71	0	23	0	16
B04	2288	1832	387	0	3	3	0	17	0	46
B05	2614	1347	1178	0	50	17	0	8	0	14
B06	302	274	11	0	0	0	0	7	0	10
B07	1696	901	458	0	139	151	0	19	0	28
B08	770	671	59	0	5	0	0	11	0	24
B09	800	548	216	0	0	0	0	4	0	32
B10	886	819	50	0	0	0	0	11	0	6
B11	1778	1681	75	0	0	0	0	8	0	14
B12	5446	1098	852	0	539	2767	0	30	0	160
B13	1414	1177	137	0	25	3	0	33	0	39
B14	16240	14432	1436	0	171	39	0	150	6	260
B15	30024	4125	7983	0	2645	14705	0	101	0	465

Table 5.11: Z01X SAF simulations results

The test coverages of every simulation were calculated using the formula presented in

the section 4.5.1 and are presented in the table 5.12.

C. name	CAT_STAT Test coverage	SAF Test coverage
B02	90.37 %	96.87 %
B03	62,96 %	70,01 %
B04	87,86 %	82,09 %
B05	56,54 %	51,96 %
B06	98,08 %	95,03 %
B07	49,1 %	54,58 %
B08	93,49 %	90,68 %
B09	81,34 %	71,61 %
B10	95,73 %	93,69 %
B11	96,3 %	95,52 %
B12	18,96 %	21,05 %
B13	75,71 %	86,8 %
B14	92,66 %	89,36 %
B15	13,7 %	14,12 %

Table 5.12: Test coverage comparisons

The test coverage of the two different fault models is compared in the graph in the figure 5.12.

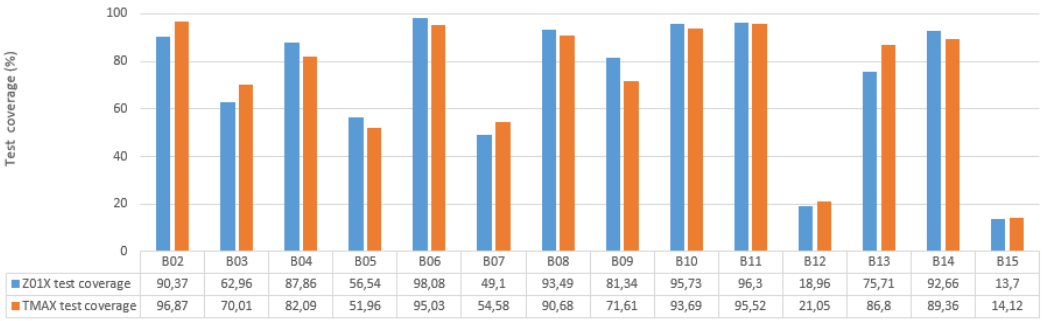


Figure 5.12: CAT_STAT and SAF test coverage comparisons

Regarding the Bcircuits version with scan chains, the final Z01X results with the CAT_STAT prime and equivalent faults are reported in the table 5.13.

C. name	CAT_STAT	DD	ND	NI	NC	NO	NS	PD	HA
B02_scan	1055	1049	2	0	0	0	0	4	0
B03_scan	7646	7235	407	0	0	0	0	4	0
B04_scan	18327	16858	1454	1	0	0	0	14	0
B05_scan	13410	11490	1863	41	12	0	0	4	0
B06_scan	2175	2141	30	0	0	0	0	4	0
B07_scan	12986	11825	818	202	137	0	0	4	0
B08_scan	5977	5527	243	203	0	0	0	4	0
B09_scan	7152	6787	361	0	0	0	0	4	0
B10_scan	5309	5274	10	5	16	0	0	4	0
B11_scan	11031	10770	255	2	0	0	0	4	0
B12_scan	36629	34077	2107	298	117	16	0	14	0
B13_scan	12014	8316	547	2224	206	286	0	435	0
B14_scan	13889	13519	343	17	9	0	0	0	1
B15_scan	169827	114837	26692	8220	3904	16110	0	64	0

Table 5.13: Z01X CAT_STAT fault simulations results

Regarding the Bcircuits version with scan chains, the final Z01X results with the SAFs are reported in the table 5.14.

C. name	SAF	DD	ND	NI	NC	NO	NS	PD	HA	UG
B02_scan	150	139	3	0	0	0	0	6	0	2
B03_scan	930	772	148	0	0	0	0	8	0	2
B04_scan	2576	2080	451	0	0	0	0	13	0	32
B05_scan	2768	2061	681	0	0	7	0	9	0	10
B06_scan	338	301	23	0	0	0	0	4	0	10
B07_scan	1888	1415	367	0	61	18	0	9	0	18
B08_scan	862	701	117	0	0	0	0	6	0	38
B09_scan	924	756	144	0	0	0	0	8	0	16
B10_scan	960	924	17	0	0	8	0	5	0	6
B11_scan	1910	1768	120	0	0	0	0	8	0	14
B12_scan	5944	4751	932	0	27	13	0	19	0	202
B13_scan	1610	947	245	0	97	179	0	100	5	37
B14_scan	17388	11849	4493	0	156	36	0	30	38	786
B15_scan	31758	13527	11013	0	1853	4913	0	57	0	395

Table 5.14: Z01X SAF simulations results

The test coverages of every simulation were calculated using the formula presented in the section 4.5.1 and are presented in the table 5.15.

C. name	CAT_STAT Test coverage	SAF Test coverage
B02_scan	99,62 %	95,94 %
B03_scan	94,65 %	83,62 %
B04_scan	92,02 %	82,01 %
B05_scan	85,69 %	74,89 %
B06_scan	98,52 %	92,37 %
B07_scan	91,07 %	75,9 %
B08_scan	92,5 %	85,43 %
B09_scan	94,92 %	83,7 %
B10_scan	99,37 %	97,11 %
B11_scan	97,65 %	93,45 %
B12_scan	93,05 %	82,9 %
B13_scan	71,02 %	63,38 %
B14_scan	97,33 %	71,46 %
B15_scan	67,63 %	43,22 %

Table 5.15: Test coverage comparisons

The test coverages of the two different fault models are compared in the graph in the figure 5.13.

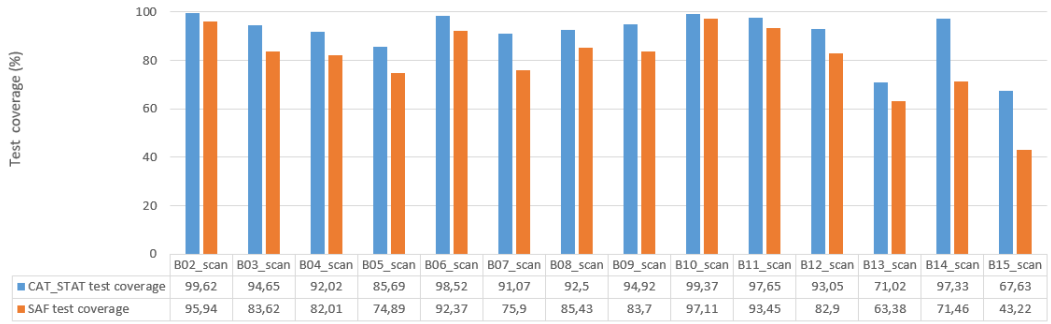


Figure 5.13: CAT_STAT and SAF test coverage comparisons with scan circuits

5.6 openMSP430 results

The developed CAT fault simulation flow has been tested, at last, on the openMSP430. It is a 16bit microcontroller core described at RT-level using Verilog [14]. The chip has Timer, GPIOs, DMA interface and other peripherals. It was synthesized with the nangate library. In particular, not all the microcontroller was synthesized, only the internal CPU.
The main purpose of this thesis was to create a CAT fault simulation flow that works

with functional test stimuli. The SBST technique is a type of functional test and it was adopted for the openMSP430 tests presented in this section. More precisely, they used Self-Test Libraries (STLs), test programs used to check if the UUT behaves correctly after the production phase. The STLs available for testing the openMSP430 were 39, but due to the high simulation type for the CAT faults, only 12 of them were used. The test patterns were described with VCD files and everyone has a unique ID. In this thesis the fault simulation results of every STL are presented with the original ID, to allow possible future works to understand what test stimuli have been used in this thesis. The openMSP430 was tested firstly with the SAF model. The fault simulations were made with TMAX and the original VCD of the STLs. The testing flow is reported in figure 5.14.

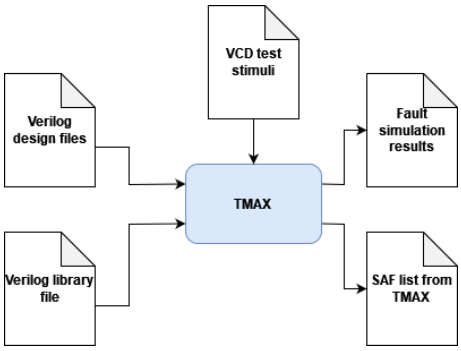


Figure 5.14: SAF testing flow with TMAX

The fault simulations results regarding the SAFs are reported in the table 5.16.

VCD ID	SAF	DT	PT	UD	AU	ND
47	33982	30301	326	804	0	2551
12	33982	30161	376	804	0	2641
13	33982	30173	331	804	0	2674
33	33982	30089	261	804	0	2828
7	33982	29978	312	804	0	2888
23	33982	29953	289	804	0	2936
20	33982	29072	289	804	0	3817
15	33982	28916	290	804	0	3972
25	33982	28766	272	804	0	4140
18	33982	27881	379	804	0	4918
35	33982	27892	259	804	0	5027
44	33982	27676	260	804	0	5242

Table 5.16: TMAX SAF simulations results

To test the CAT faults with Z10X, three different testing flows were used. In the first,

the original VCD test stimuli were read by TMAX to create the corresponding STIL test patterns. More precisely, the patterns in the STIL and VCD files of every STLs are the same, what changes is the format used to describe them. Then the STIL test patterns were read by Z01X to simulate the fault list. The complete flow is reported in figure 5.15.

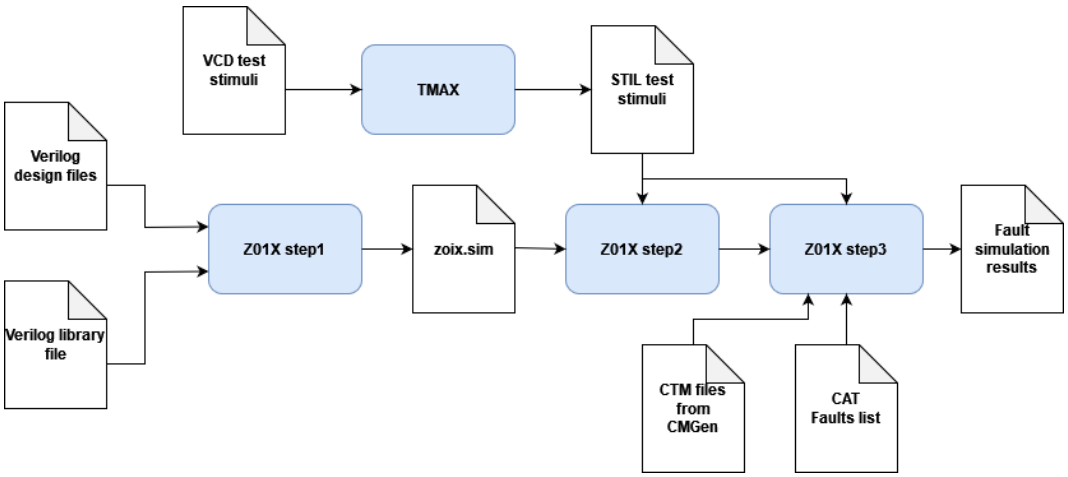


Figure 5.15: CAT_STAT testing flow with STIL test stimuli

The simulations results are reported in the table 5.17.

STIL ID	CAT_STAT	DD	ND	NI	NC	NO	NS	PD	HA	IA
47	26593	23094	1349	1341	224	340	99	145	1	0
12	26593	23225	1361	1347	243	170	99	148	0	0
13	26593	23111	883	1824	282	182	99	210	0	0
33	26593	23251	1133	1357	259	330	99	157	7	0
7	26593	23079	1243	1569	284	179	99	140	0	0
23	26593	22949	1251	1484	301	335	99	167	7	0
20	26593	22249	2127	1508	257	180	99	173	0	0
15	26593	22068	1724	2054	327	176	99	145	0	0
25	26593	22214	1467	2049	358	220	99	186	0	0
18	26593	21405	1843	2335	389	363	99	159	0	0
35	26593	21515	1850	2184	311	444	99	189	1	0
44	26593	21155	1998	2397	406	391	99	147	0	0

Table 5.17: Z01X CAT_STAT simulations results with STILs

In the second testing flow, the original VCD test stimuli were read by TMAX to create the corresponding testbench test patterns. More precisely, the patterns in the testbench and VCD files of every STLs are the same, what changes is format used to describe them. Then the Verilog file of the testbench was compiled with the UUT Verilog file

and the Z01X CAT fault simulation was performed. The complete flow is reported in the figure 5.16.

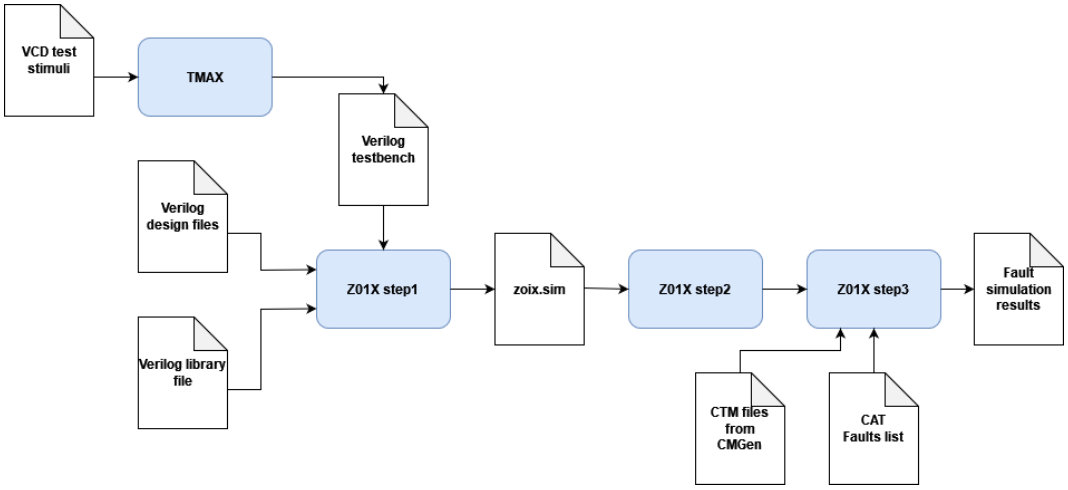


Figure 5.16: CAT_STAT testing flow with testbench test stimuli

The simulation results are reported in the table 5.18.

TB ID	CAT_STAT	DD	ND	NI	NC	NO	NS	PD	HA	IA
47	26593	23096	1348	1341	224	340	99	145	0	0
12	26593	23228	1358	1347	243	170	99	148	0	0
13	26593	23162	829	1824	282	182	99	215	0	0
33	26593	23282	1109	1357	259	330	99	157	0	0
7	26593	23153	1169	1569	284	179	99	140	0	0
23	26593	22954	1253	1484	301	335	99	167	0	0
20	26593	22254	2122	1508	257	180	99	173	0	0
15	26593	22076	1715	2054	327	176	99	146	0	0
25	26593	22230	1451	2049	358	220	99	186	0	0
18	26593	21405	1843	2335	389	363	99	159	0	0
35	26593	21525	1842	2184	311	444	99	188	0	0
44	26593	21155	1998	2397	406	391	99	147	0	0

Table 5.18: Z01X CAT_STAT simulations results with testbenchs

In the last testing flow, the original VCD test stimuli were directly read by Z01X and used to perform the fault simulation. The complete flow is reported in the figure 5.16.

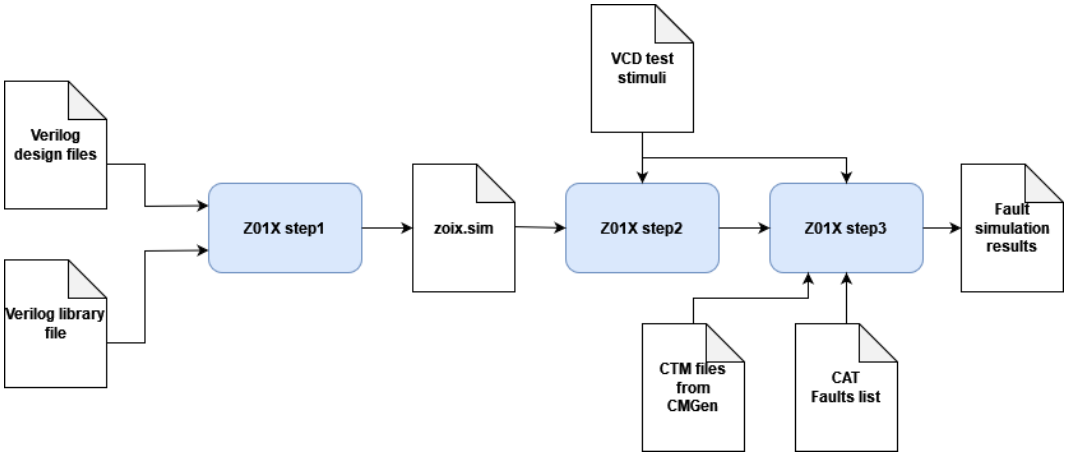


Figure 5.17: CAT_STAT testing flow with VCD test stimuli

The simulation results are reported in the table 5.19.

VCD ID	CAT_STAT	DD	ND	NI	NC	NO	NS	PD	HA	IA
47	26593	4874	527	1538	248	340	99	20	0	18947
12	26593	5092	241	1662	267	167	99	16	0	19049
13	26593	4983	155	2054	308	212	99	24	0	18758
33	26593	5040	312	1642	290	330	99	19	0	18861
7	26593	4988	301	1855	355	176	99	18	0	18801
23	26593	4857	395	1773	319	338	99	19	0	18793
20	26593	4471	777	1823	327	191	99	19	0	18886
15	26593	4361	377	2552	400	173	99	18	0	18613
25	26593	4788	305	2313	434	334	99	21	0	18299
18	26593	4494	368	2771	565	401	99	22	0	17873
35	26593	4659	327	2497	411	447	99	20	0	18133
44	26593	4238	304	3079	544	441	99	18	0	17870

Table 5.19: Z01X CAT_STAT simulations results with VCDs

From the fault simulations results, the corresponding test coverage were extracted. Regarding the TMAX SAF simulations, the test coverage was calculated with the formula presented in the section 4.5.1, like for the Z01X CAT_STAT faults. The results are reported in the table 5.20.

Test stimuli ID	SAF TC	STIL TC	TB TC	VCD TC
47	91,82 %	87,11 %	87,12 %	18,36 %
12	91,47 %	87,61 %	87,62 %	19,17 %
13	91,44 %	87,3 %	87,5 %	18,78 %
33	91,08 %	87,72 %	87,84 %	18,98 %
7	90,83 %	87,04 %	87,32 %	18,79 %
23	90,72 %	86,61 %	86,62 %	18,29 %
20	88,06 %	83,99 %	84 %	16,84 %
15	87,59 %	83,25 %	83,28 %	16,43 %
25	87,11 %	83,88 %	83,94 %	18,04 %
18	84,61 %	80,79 %	80,79 %	16,94 %
35	84,46 %	81,26 %	81,29 %	17,55 %
44	83,81 %	79,82 %	79,82 %	15,97 %

Table 5.20: Test coverage comparisons

The test coverages are also compared in the graph shown in 5.18.

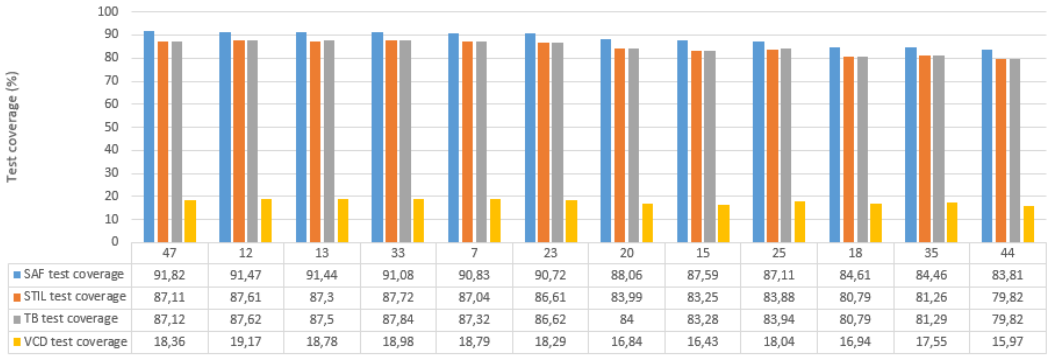


Figure 5.18: Test coverage comparisons

6 Discussion and Conclusions

The main purpose of this thesis was to create a CAT fault simulation flow able to work with functional test patterns. The first step was to understand the available works and tools for creating the testing flow. Then a general approach was developed and presented: in a few words starting from the synthesis library cells a CAT fault model is created for every cell. Then they are used to describe all the CAT faults presented in the UUT. At last, the created CAT fault list is simulated with the test stimuli in a fault simulator.
After the proposed approach, the specific implementation was reported. The software to create the CAT fault model of every cell, CMGen, was explained, followed by the scripts and programs that create the correct and complete CAT fault list for the selected fault simulator, Z01X.
At last, the results of the Z01X fault simulations are presented. The tests were made on different circuits and with different test stimuli. In particular, the results focus on the functional test patterns. This last chapter presents comments and conclusive remarks concerning for every result section.

6.1 Consideration about Z01X results validation

In the section 5.1 the results of the experiment conducted on a small gate-level netlist named simpleCircuit were shown. The open question is why Z01X does not detect the 4 missing faults D22, D173 of the cell REG_AND and D22 and D173 of the cell REG_OR. The pattern that detects the faults is particular because the precedent value of the output port in the precedent state (Q- in the static detection table reported in the figure 4.12) is equal to the actual value of the output port Q in the test pattern(Q). I made Z01X dump the faulty machine of the defect D22 to check the internal simulation values of the fault, i.e., I tell Z01X to create a file that can be read using Verdi [16], an advanced solution for simulation debugging. The good machine behavior of REG_AND in the simpleCircuit is reported in the figure 6.1.

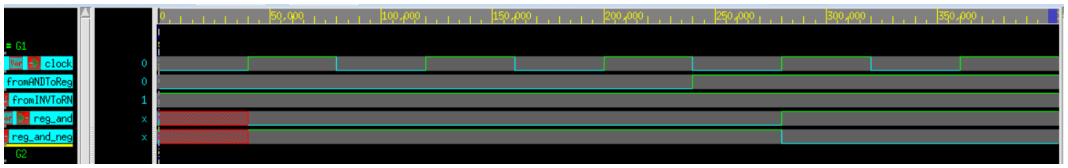


Figure 6.1: Good machine evcd waveform

The Z01X faulty machine created by the injection of the fault D22 is reported in the figure 6.2.

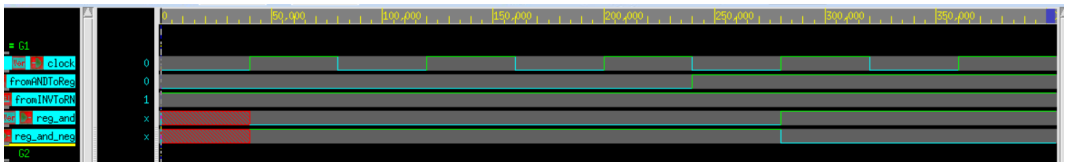


Figure 6.2: D22 faulty machine evcd waveform

The behaviours of the 2 machines are the same, this means that according to Z01X the pattern that is able to excite D22 was not present. Looking at the simulation, the value of the inputs and the precedent state of the flip-flop that is required to excite the fault are present.

6.2 Consideration about Z01X and TMAX ATPG patterns tests

In section 5.2 the expected results coincide with the values of the fault simulations.

6.3 Consideration about ITC'99 benchmark circuits results

TMAX CAT ATPG and Z01X CAT fault simulations results differ of some percentage points in the section 5.3. But which one is right? The performed analyses are inconclusive on this point, as there are not enough elements to declare which fault simulator is right. I explain how this problem can be resolved in the section 6.7.

6.4 Consideration about ITC'99 benchmark circuits with scan logic results

TMAX CAT ATPG was precisely developed to test circuits with DfT-like scan chains inside. In section 5.4 the test coverages of Z01X and TMAX differ by 1-2 percentage points. This is a positive result because if Z01X was able to produce the expected results regarding the scan-based netlist, probably it will work correctly for the functional test patterns.

6.5 Consideration about CAT and SAF comparison on ITC'99 benchmark circuits

In section 5.5 the test coverages of two different fault models were compared, the SAF model and CAT model. The test stimuli given in input for the fault simulations were the same for both fault models. The CAT faults are usually more numerous than the SAFs and normally the test coverage obtained from the SAF model is higher than the test coverage obtained from the CAT model. The results show that sometimes the CAT test

coverages of the normal versions of the Bcircuits are higher than the SAF test coverages. Does this imply that Z01X is committing errors in simulating the CAT faults? Not necessarily. It depends on the average probability of CAT faults in the CAT fault model to be detected in comparison to the probability of detecting SAFs. For example, I created two imaginary versions of the static detection table of an AND3 cell reported in 6.1.

A1	A2	A3	ZN	sa0	sa1	D1	D2	D3	D4	P. name
0	0	0	0	0	1	1	0	0	0	P1
0	0	1	0	0	1	0	0	0	0	P2
0	1	0	0	0	1	1	0	0	1	P3
0	1	1	0	0	1	0	0	0	1	P4
1	0	0	0	0	1	0	0	0	1	P5
1	0	1	0	0	1	0	0	0	0	P6
1	1	0	0	0	1	0	0	1	0	P7
1	1	1	1	1	0	0	1	1	0	P8
A1	A2	A3	ZN	sa0	sa1	D1	D2	D3	D4	P. name
0	0	0	0	0	1	1	0	0	1	P1
0	0	1	0	0	1	1	0	1	0	P2
0	1	0	0	0	1	1	0	1	0	P3
0	1	1	0	0	1	1	1	1	1	P4
1	0	0	0	0	1	1	0	1	1	P5
1	0	1	0	0	1	1	1	0	0	P6
1	1	0	0	0	1	0	1	1	0	P7
1	1	1	1	1	0	0	1	0	1	P8

Table 6.1: On the top, the first version of the static detection table, on the bottom the second version of the static detection table.

In the first, I calculate the probability of two patterns to detect the SAFs. For example, a couple of test patterns (P1,P8) are able to detect the two SAFs, but the couple (P1,P2) is not able to do that. Summing up, the probability of two input patterns to detect all the SAFs is equal to 21.87 %. The probability of detecting all the CAT faults with two test patterns is equal to 3.12 %. In the second version of the static detection table, the probability of detecting the SAFs with two patterns is still 21.87 %. Instead, the probability of detecting all the CAT faults with two test patterns is equal to 42.18 %, bigger than the SAF probability. So I think that it is not possible to affirm that Z01X made an error during the fault simulation of the CAT_STAT faults. An interesting study to perform on the logic synthesis library could be understood if effectively in the nangate library the CAT faults are slightly easy to detect than the SAFs. Checking by hand, I found two cells inside the nangate library that are quite different. The first is the FA_X1 cell already presented in the section 4.1.4, here the number of CAT faults is higher than the number of SAF. Also, the patterns necessary to detect all the CAT_STAT faults are

necessary 8, instead the input patterns necessary to detect the SAF on the input and output ports are only 5. The figure 6.3 reports the SAF of the cell.

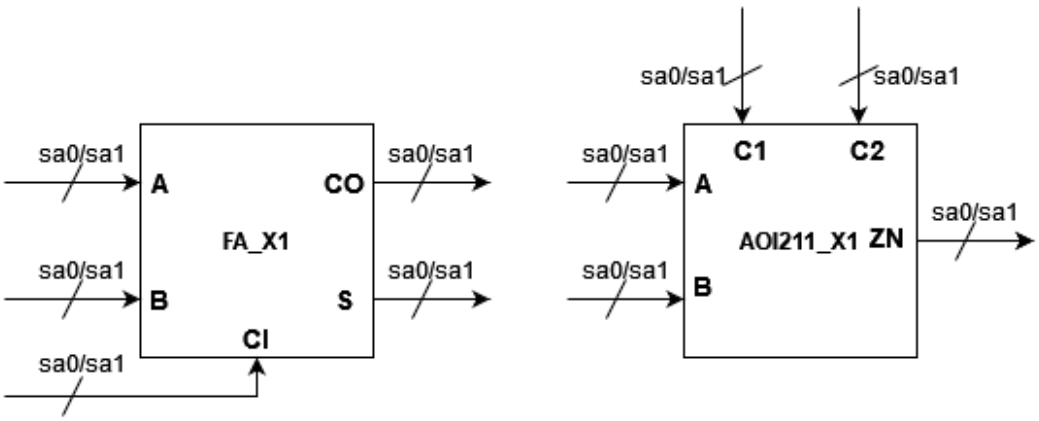


Figure 6.3: SAFs applied to the two cells

The 5 patterns that detect the SAFs are reported in the table 6.2.

A	B	CI
1	0	0
0	0	0
0	1	0
0	0	1
0	1	1

Table 6.2: Patterns that detect the FA_X1 SAFs

The second cell is labelled AOI211_X1, the static detection table is reported in the figure 6.4.

```
- [Table, Static]
- [A, B,C1,C2,ZN,D4,D18,D37,D41]
- [0, 0, 0, 0, 1, 1,  0,  0,  0]
- [0, 0, 0, 1, 1, 1,  0,  0,  0]
- [0, 0, 1, 0, 1, 1,  0,  0,  0]
- [0, 0, 1, 1, 0, 0,  1,  0,  0]
- [0, 1, 0, 0, 0, 0,  0,  1,  0]
- [0, 1, 0, 1, 0, 0,  0,  1,  0]
- [0, 1, 1, 0, 0, 0,  0,  1,  0]
- [0, 1, 1, 1, 0, 0,  0,  0,  0]
- [1, 0, 0, 0, 0, 0,  0,  0,  1]
- [1, 0, 0, 1, 0, 0,  0,  0,  1]
- [1, 0, 1, 0, 0, 0,  0,  0,  1]
- [1, 0, 1, 1, 0, 0,  0,  0,  0]
- [1, 1, 0, 0, 0, 0,  0,  0,  0]
- [1, 1, 0, 1, 0, 0,  0,  0,  0]
- [1, 1, 1, 0, 0, 0,  0,  0,  0]
- [1, 1, 1, 1, 0, 0,  0,  0,  0]
```

Figure 6.4: AOI211_X1 static detection table

The CAT_STAT faults can be detected with only 4 input patterns. In the table only the CAT_STAT prime faults are reported, but their number is lower than the number of SAFs on the cell ports shown in the figure 6.3. The input patterns needed to test the SAFs are 6, 2 more than the pattern needed to cover the CAT_STAT faults. They are reported in the table 6.3.

A	B	C1	C2
1	0	0	0
0	0	0	0
0	1	0	0
0	0	1	1
0	0	0	1
0	0	1	0

Table 6.3: Patterns that detect the AOI211_X1 SAFs

I can conclude that it is not necessarily true that CAT faults require more input patterns to be covered in comparison to SAFs. It depends on the characterization of the synthesis library.
In the results of the fault simulations of the Bcircuits with the scan chains (figure 5.13) the test coverage of the CAT_STAT faults sometimes is bigger than the SAT test coverage. This may be due to the fact that the scan flip-flops that replace the normal flip-flops contain easy-to-test CAT_STAT faults that increase the overall test coverage

of the Bcircuits.

6.6 Consideration about openMSP430 results

The openMSP430 results, presented in the section 5.6, show that the VCD format produce discrepant results in comparison to the ones created by the STIL and testbench test stimuli (figure 5.18). In the VCDs results, many faults are labelled as IA i.e., illegal access. According to Z01X this means that during the fault simulation:"the fault propagates to an unallocated array/class so it was dropped from the simulation". Due to the structure of the verilog file of the UUT, the presence of this kind of fault does not seem to be justified. In this case the flow or the tool may require further tweaking.
The IA faults created by the fault simulations on the openMSP430 confirm the suspect on the simulation performed with evcd on the simpleCircuit gate-level netlist. In fact, as reported in the section 5.1, the test with the test stimuli represented with ecvd mark an expected covered fault as IA. I suggest using STIL file or testbench when Z01X have to test CAT faults.

6.7 Conclusive remarks and future works

Despite some small discrepancies between the expected and the obtained results, the developed flow is able to produce sufficiently reliable and precise data to assess the effectiveness of a functional test set. Nonetheless, the experiments performed demonstrate that the CAT paradigm is quite new and the tools handling the related fault models may require some time to become fully mature. In fact, the experiment that I made on the simpleCircuit netlist in the section 5.1 confirm that Z01X work correctly with STIL and testbench stimuli for the CAT_STAT faults. I think that to be completely sure that the developed flow with Z01X fault simulations works, one of these two possible actions can be taken:

- Select another fault simulator able to work with CAT faults and compare the Z01X results with it;
- Develop an ad-hoc CAT fault simulator.

Developing an ad-hoc CAT fault simulator can be a valid future work. An example of implementation could be modifying the gate-level netlist of the UUT with a test logic able to understand if the CAT faults have been exited and propagate the effect in the netlist during the simulation of the test stimuli and check if some effects are observable on the outputs. I present a very simple example. Imagine that we want to test the CAT_STAT fault in the static detection table reported in the figure 6.5) with the gate-level netlist of the UUT.

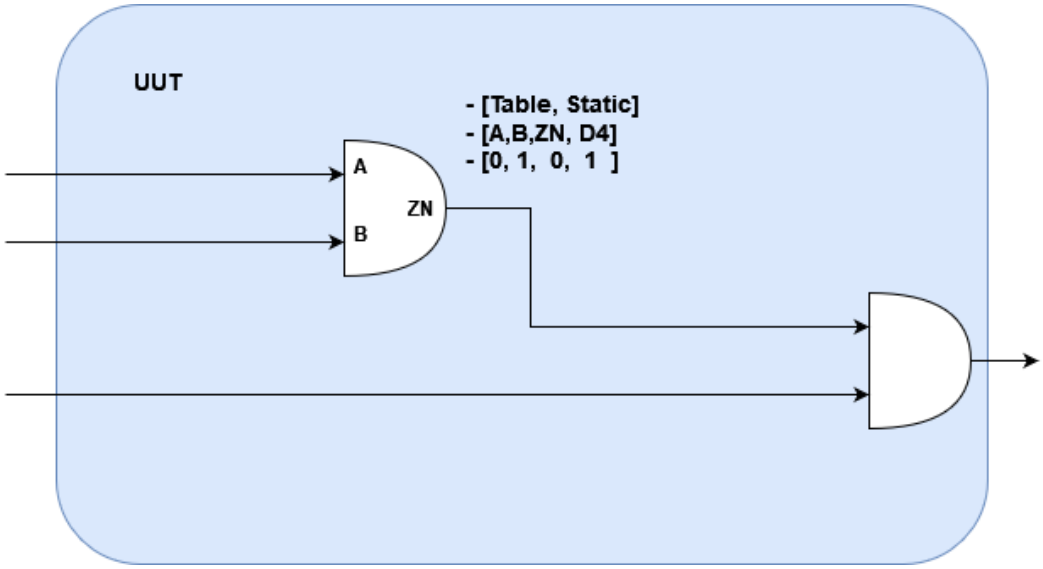


Figure 6.5: UUT netlist with static detection table example

We have to develop a program able to read the desired fault to check in the CTM files from CMGen. In particular, the program is able to understand the necessary inputs conditions to excite the CAT faults and the output effect to apply on the netlist. This conditions are translated into hardware and a faulty version of the UUT is created. For example the netlist of the faulty machine regarding the CAT faults D4 is reported in the figure 6.6).

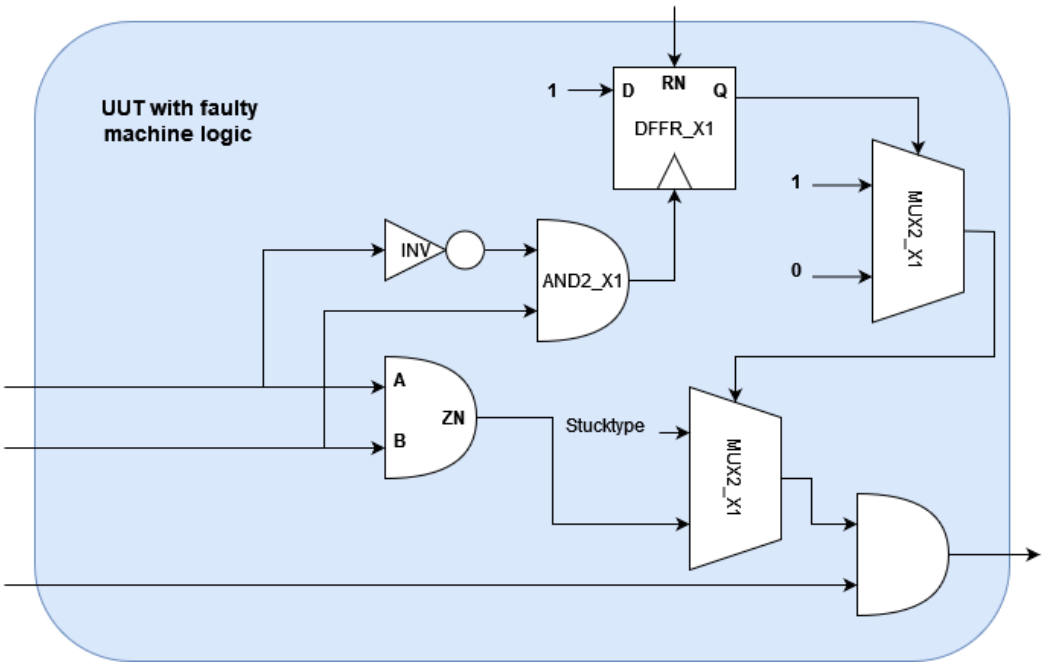


Figure 6.6: UUT with faulty machine logic

With this particular implementation, the test logic inserted is built using the cells present in the nangate library. Also a reset signal is needed that controls specifically the test logic. This is only a specific way to create a CAT fault simulator, many options are available.

7 List of acronyms

ATPG Automatic Test Pattern Generator
ATE Automatic Test Equipment
BIST Built-In Self-Test
CAT Cell-Aware Test
CAT_STAT Static CAT
CAT_DYN Dynamic CAT
DFT Design For Testability
DL Defect Level
DRC Design Rule Check
EVCD Extended Value Change Dump
FM Faulty Machine
FMS full-multi-stuck
FPGA Field Programmable Gate Arrays
FSM Finite-State Machine
HDL Hardware Description Language
HMS Half-Multi-Stuck
IC Integrated Circuit
LFSR Linear Feedback Shift Registers
nangate Silvaco Nangate 45nm
NS Normal-Stuck
MP Multi-Port
MS Multi-Stuck
ppm Part Per Million
RTL Register Transfer Level
SAF Stuck-At Fault
SBST Software-Based Self-Test
SoC System on Chip
STIL Standard Test Interface Language
STL Self-Test Library
TDF Transition Delay Fault
TMAX Tetramax
UUT Unit Under Test
VCD Value Change Dump
WfT WaveformTable

List of Figures

List of Tables

Bibliography

[1] S. Eichenberger, J. Geuzebroek, C. Hora, B. Kruseman and A. Majhi, "Towards a World Without Test Escapes: The Use of Volume Diagnosis to Improve Test Quality," 2008 IEEE International Test Conference, 2008, pp. 1-10, doi: 10.1109/TEST.2008.4700604.

[2] F. Corno, M. S. Reorda and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," in IEEE Design Test of Computers, vol. 17, no. 3, pp. 44-53, July-Sept. 2000, doi: 10.1109/54.867894.

[3] P. C. Maxwell, R. C. Aitken, V. Johansen and Inshen Chiang, "The effect of different test sets on quality level prediction: when is 80% better than 90%?," 1991, Proceedings. International Test Conference, 1991, pp. 358-, doi: 10.1109/TEST.1991.519695.

[4] TestMAX ATPG and TestMAX Diagnosis User Guide, Version S-2021.06-SP3, October 2021.

[5] Z01X Simulator Manufacturing Assurance User Guide, Version R-2020.12 December 2020.

[6] CMGen User Guide, Version T-2022.03, March 2022.

[7] Modelsim User's Manual, Software Version 10.1c, 2012.

[8] ASIC design flow theory available at: https://www.einfochips.com/blog/asic-design-flow-in-vlsi-engineering-services-a-quick-guide/.

[9] ASIC design flow theory available at: https://anysilicon.com/asic-design-flow-ultimate-guide/.

[10] Luciano Lavagno, Elettronica applicata, University Transparencies 2019/2020.

[11] Matteo Sonza Reorda, Testing and fault tolerance, University Transparencies, 2021/2022.

[12] STIL standard explanation: https://shop.micross.com/pdf/Micross_Technical_Paper-STIL_Language_Test_Vector_Format_Simplified.pdf.

[13] F. Hapke et al., "Cell-Aware Test", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 9, pp. 1396-1409, Sept. 2014, doi: 10.1109/TCAD.2014.2323216.

[14] O. Girard, openMSP430, 2009. [Online]. Available: https://opencores.org/projects/openmsp430.

[15] P. Bernardi et al., "Recent Trends and Perspectives on Defect-Oriented Testing," 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2022, pp. 1-10, doi: 10.1109/IOLTS56730.2022.9897647.

[16] Verdi User Guide, Version S-2021.09-SP2, March 2022.

[17] Open-source technology library Silvaco Nangate 45nm. Available: https://silvaco.com/news/silvaco-and-si2-release-unique-free-15nm-open-source-digital-cell-library/