

ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG

INSTITUT FÜR INFORMATIK

Lehrstuhl für Rechnerarchitektur

Prof. Dr. Bernd Becker



STUDIENARBEIT

Automatic Test Pattern Generation for
Power Droop Testing

Alejandro Czutro

(Student-ID 1133719)

Supervision: Dr. Ilia Polian

7th February 2006

CONTENTS

1	Introduction	9
2	Conventions	11
3	Power Droop Testing	15
3.1	On power droop	15
3.1.1	Low-frequency power droop	15
3.1.2	High-frequency power droop	16
3.2	A test method for power droop	18
4	ATPG Problem Formulation	21
5	Automatic Test Pattern Generation	25
5.1	The conventional D-algorithm	25
5.2	Dynamically constrained D-algorithm	27
5.3	Handling large M and N values	31
5.3.1	Step 1	31
5.3.2	Step 2	32
5.3.3	Step 3	33
5.3.4	Step 4	34
6	Implementation of the ATPG Procedure	37
6.1	Unfolded circuits	37
6.2	Implementation of the D-algorithm	41
6.2.1	Basic implementation	41
6.2.2	Applying Rules 1, 2 and 3	46
7	Experimental Results	53
7.1	The benchmarks	53
7.2	Settings	55
7.2.1	Choosing the aggressors	55
7.3	Results	57
7.4	Larger sequential circuits	68

4

CONTENTS

8 Conclusions 69

A Contents of the Attached CD-ROM 71

LIST OF FIGURES

2.1	Example circuit s00027	12
3.1	CUT connected to VRM, and parasitic inductance	15
3.2	CUT, VRM and capacitor that diminishes effects of $\frac{dI}{dt}$ event	16
3.3	Development of V_{DD} after a $\frac{dI}{dt}$ event	16
3.4	4-layer power grid	17
3.5	Highest switching activity cannot be 100%	18
4.1	Problem formulation: given circuit	21
4.2	Problem formulation: circuit under application of the generated test sequence, example for $T = 0 \rightarrow 1$	22
5.1	ATPG, an example	30
5.2	ATPG, an example	33
6.1	Circuit representation with <code>test_circ</code> and <code>unfcirc</code>	38
6.2	Recursive D-algorithm	42
6.3	D-algorithm, sub-procedure <code>imply_and_check()</code>	43
6.4	D-algorithm, sub-procedure <code>propagation()</code>	44
6.5	D-algorithm, sub-procedure <code>justification()</code>	44
6.6	Application of Rule 2	47
6.7	The cell-selecting procedure and the application of Rule 1	50
7.1	The victim's suitable neighbours	56
7.2	Experimental results: normalised average results	63

LIST OF TABLES

2.1	Roth's Algebra	13
7.1	Combinational ISCAS 85 circuits	53
7.2	Sequential ISCAS 89 circuits	54
7.3	Experimental results: $M = N = 10$, average figures	58
7.4	Experimental results: $M = N = 10$, rising transition	59
7.5	Experimental results: $M = N = 10$, falling transition	60
7.6	Average switching activity, 5000 cycles, random input patterns	62
7.7	Experimental results: average figures	62
7.8	Experimental results: $M = N = 30$, rising transition	64
7.9	Experimental results: $M = N = 50$, rising transition	65
7.10	Experimental results: $M = N = 100$, rising transition	66
7.11	Experimental results: $M = N = 150$, falling transition	67
7.12	Larger sequential ISCAS 89 circuits	68
7.13	Experimental results: larger sequential circuits, $M = N = 10$, rising transition	68

1

INTRODUCTION

WHEN A CIRCUIT'S SWITCHING ACTIVITY, WHICH IS A FUNCTION OF THE INPUT PATTERNS, CHANGES ABRUPTLY, SUDDEN DROP OR RISE IN POWER SUPPLY VOLTAGE MAY BE CAUSED. THIS CHANGE IS KNOWN AS POWER SUPPLY NOISE AND MAY BE THE ORIGIN OF TOO LARGE SWITCHING DELAYS. THE SITUATION CAN BE AGGRAVATED BY CAPACITIVE CROSS-TALK IN SIGNAL LINES. THE COMBINED EFFECT MAY CAUSE AN IC TO FAIL. NOWADAYS, SUCH FAILURES CANNOT BE SCREENED DURING TESTING AS CONVENTIONAL FAULT MODELS DO NOT ACCOUNT FOR COMBINATION OF SIGNAL INTEGRITY PROBLEMS. IN THIS WORK A TECHNIQUE FOR SCREENING SUCH FAILURES IS PRESENTED. THE TECHNIQUE IS BASED ON THE APPLICATION OF LONG SEQUENCES OF INPUT PATTERNS. THE WORK FOCUSES ON THE TEST PATTERN GENERATION ALGORITHM WHICH IS BASED ON THE CLASSICAL D-ALGORITHM.

State-of-the-art high-performance digital ICs manufactured in deep-submicron technologies tend to draw considerable amounts of power during operation. Sharp changes in power consumption are possible within only few clock cycles. For example, a microprocessor that, after being in the idle mode for several cycles, has to start complex calculations involving simultaneous use of multiple fixed-point and floating-point units may achieve a difference in power consumption of more than 100 W.

The term `power droop` is defined in [1] and describes the impact of power consumption transients on logic values of signal lines of a circuit and thus the correctness of the circuit's operation. There are two types of power droop, `low-frequency` and `high-frequency` power droop. In this work a method for testing for power droop is presented. The method applies a long sequence of input patterns that creates worst-case power droop conditions by combining the effects of low-frequency and high-frequency power droop. After presenting the method the work focuses on the necessary automatic test pattern generation (ATPG) algorithm which is based on the classical D-algorithm and is capable of generating new constraints on-the-fly based on previous assignments. The generated sequence can be used for evaluating early silicon for design flaws such as an insufficient sizing of the power grid (which may be a cause for high-frequency power droop, see Subsection 3.1.2) and in manufacturing test. Since power droop belongs to the class of `circuit marginalities` ([5]), some ICs may be affected stronger than others. By applying the generated sequence, the ICs which are vulnerable to power droop can be identified and either rejected or binned as low-performance parts.

The writing is organised as follows. The next chapter introduces some conventions for this work. In Chapter 3 the power droop phenomenon is introduced and the test method is presented. In Chapter 4 the ATPG problem is formulated formally. The following two chapters introduce the ATPG procedure and implementation issues. Experimental results are reported in Chapter 7. Chapter 8 concludes the work.

2

CONVENTIONS

In this chapter we fix the terminology used in this writing and, by means of a small example (see Figure 2.1), we show how circuits are represented graphically throughout this work.

Since the benchmarks we used to test the designed ATPG procedure are gate-level net-lists given in the format the circuit representation library `test_circ` understands, this library was used for basic circuit representation. The way `test_circ` manages the representation of a circuit's internal structure has been the basis for the following conventions which we observe throughout this work.

- Logic gates, input and output pins, and also secondary inputs and outputs are called `cells`. Each cell in the circuit is identified uniquely by an integer between 0 and $k - 1$, where the circuit contains k cells. That identification number is called `c_ord`. Instead of speaking about the cell with `c_ord` i , one can simply say cell i .
- Input pins are also simply called inputs, output pins outputs. The shorthands `P_IN`, `P_OUT`, `S_IN` and `S_OUT` may replace the terms primary input, primary output, secondary input and secondary output, respectively.
- Each cell not being an input or an output is of one of the following types: `AND2`, `AND3`, ..., `AND11`, `NAND2`, `NAND3`, ..., `NAND11`, `OR2`, `OR3`, ..., `OR11`, `NOR2`, `NOR3`, ..., `NOR11`, `INV` or `BUF`. Instead of talking about a cell having type `INV`, one can simply say the cell to be an `INV`.
- Lines are called `signals`. Like cells, signals are identified uniquely by an integer between 0 and $l - 1$, where the circuit contains l signals. That identification number is called `s_ord`. Instead of speaking about the signal with `s_ord` i , one can simply say signal i .
- For each fanout stem and all its corresponding fanout branches there is only one signal. That means, in the circuit there is only one signal object representing the stem and all branches, not an object for each of them. Fanout branches of a signal i are referred to as signal i 's 0-th branch, signal i 's 1-st branch and so on. (An example for this concept is signal 14 in Figure 2.1.)

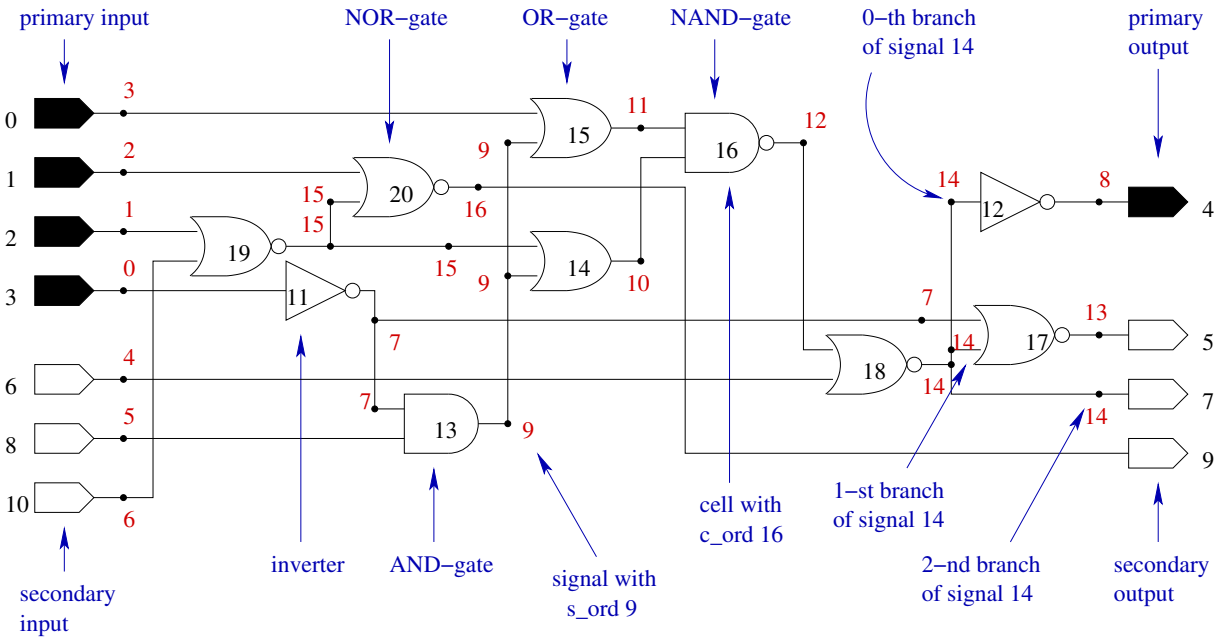


FIGURE 2.1: EXAMPLE CIRCUIT s00027

- Each signal has exactly one drain cell per branch and one source cell in all.
- Inputs have exactly one output signal and no input signals, outputs have exactly one input signal and no output signals. Cells not being inputs or outputs have one or more input signals and exactly one output signal.
- Cells not being P_OUTs have one or several **successors**, cells not being P_INs have one or several **predecessors**. Successors are the cell's output signal's drain cells. Predecessors are the cell's input signals' source cells.
- Since S_INs and S_OUTs are outputs and inputs of memory elements, respectively, there are always as many S_INs as S_OUTs. Each S_IN corresponds to exactly one S_OUT. Thus each S_OUT has got exactly one successor (the corresponding S_IN) and each S_IN has got exactly one predecessor (the corresponding S_OUT), although this is not visible in Figure 2.1.

Additionally, the following definitions are necessary:

- Performing automatic test pattern generation implies having to carry out logic value simulation in two different modes. Basically, first the simulation has to be performed assuming the circuit to be fault-free. Once the values thus obtained have been saved, simulation has to be performed assuming the circuit to have a certain fault f . Comparing the new obtained values to the ones saved before makes clear what signals are affected by the presence of the fault f .

The necessity of managing two different logic values for each signal can be avoided by using an extended range of logic values known as Roth's Algebra (see [2, Chapter 6]). This algebra consists of the values 0, 1, D, D' and X.

A signal is said to have the value	if its value in the fault-free circuit is	and its value in the faulty circuit is
1	1	1
0	0	0
D	1	0
D'	0	1
X	X	X

TABLE 2.1: ROTH'S ALGEBRA

The logic value X in the faulty and the fault-free circuits stands for **unspecified** or **don't-care**. D and D' are called **error values**.

- Let c_1 and c_2 be cells. Let s be any input signal of c_1 . If s is also an input signal of c_2 , we say that c_2 is a **sibling cell** of c_1 . For example, cells 15 and 20 are siblings of cell 14 in Figure 2.1.
- Let s be a signal and let c be its source cell. Then, the input signals of c are called **predecessor signals** of s . For example, signals 15 and 9 are predecessors of signal 10 in Figure 2.1.

3

POWER DROOP TESTING

3.1 On power droop

Power droop (defined in [1]) describes the impact of power consumption transients on logic values of signal lines and thus on the correctness of the circuit's operation. There is low-frequency and high-frequency power droop.

3.1.1 Low-frequency power droop

Low-frequency power droop occurs when the voltage regulator module (VRM) is unable to handle large transients in the power consumption of the whole device due to non-negligible inductance of the interconnect. During a very short period of time the VRM isn't able to deliver enough current, this circumstance resulting in power starvation (drop of voltage on certain lines). Because of capacitive effects it takes a number of clock cycles until low-frequency power droop creates the largest impact.

Figure 3.1 shows the circuit under test (CUT) connected to power supply (VRM). The parasitic inductance of the interconnect is denoted by L . The term $\frac{dI}{dt}$ event denotes a sudden increase in current I demanded per unit time t , which is equivalent to a sudden increase in power consumption. After a $\frac{dI}{dt}$ event, the CUT's power supply voltage V_{DD} is reduced by $L \cdot \frac{dI}{dt}$. For a current transient of 100 A (equivalent to a power transient of 100 W if $V_{DD} \approx 1$ V) taking place within 10^{-9} seconds (three cycles on a 3.3 GHz machine), this value is dramatic even for a small inductance $L < 1$ nH.

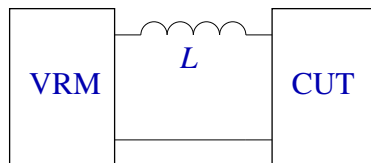


FIGURE 3.1: CUT CONNECTED TO VRM, AND PARASITIC INDUCTANCE

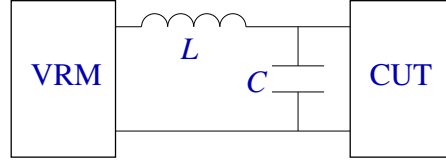


FIGURE 3.2: CUT, VRM AND CAPACITOR THAT DIMINISHES EFFECTS OF $\frac{dI}{dt}$ EVENT

This effect can be diminished by adding a capacitor C , as shown in Figure 3.2, to cover the CUT's short-term demand for current after a $\frac{dI}{dt}$ event. Since the inductance of the line between the capacitor and the CUT is much lower than L , the V_{DD} drop is less severe. However, if C is discharged before the VRM is ready to supply the full amount of needed current, a V_{DD} drop still takes place, albeit of a smaller extent and some time after the initial $\frac{dI}{dt}$ event. Figure 3.3 sketches this circumstance. The red dashed curve indicates V_{DD} 's development over time in absence of a capacitor, the blue solid curve in presence of a capacitor. The curves are for illustration only and have not been obtained by measurement or simulation.

Whether the V_{DD} drop is large enough to cause a logic or delay failure depends on the extent of the $\frac{dI}{dt}$ event, i.e. on how much more current is demanded over how small a period of time, and on the values of L and C as well as on the characteristics of the VRM. In any case, it is especially important to remark that the impact is most critical several clock cycles after the actual $\frac{dI}{dt}$ event.

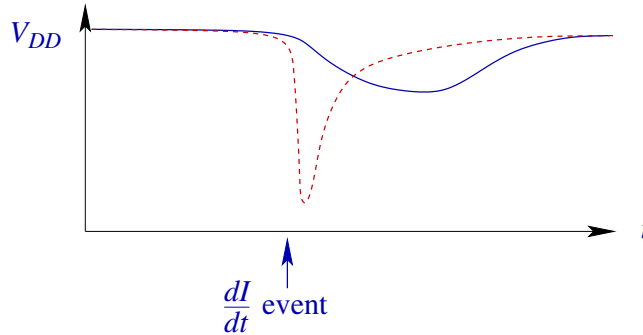


FIGURE 3.3: DEVELOPMENT OF V_{DD} AFTER A $\frac{dI}{dt}$ EVENT

3.1.2 High-frequency power droop

The reason for high-frequency power droop is the limited capability of the power distribution network on-chip to deliver power quickly. The power grid stretches over several metallisation layers which are connected by vertical vias. An example 4-layer grid is illustrated in Figure 3.4. The topmost layers are often reserved for power rails while lower

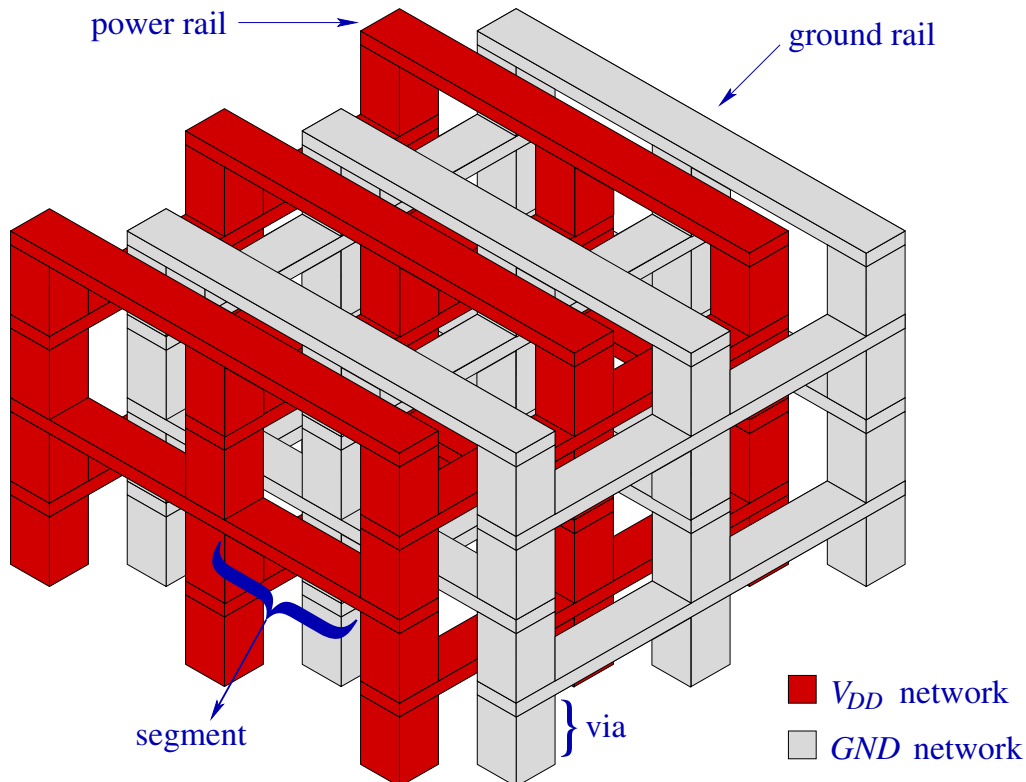


FIGURE 3.4: 4-LAYER POWER GRID

layers are shared with logic signal lines. In general, the power delivery capacity of a power rail is given by its width, which tends to decrease on lower layers. There is pressure to limit the width of power rails as the area consumed by them is not available to logic signal lines. The vias connecting power rails of different layers are relatively small and hence are an obvious bottleneck for power delivery. The layer on which the logic cells are attached to the power grid can be very low, which means that there are six or seven metallisation layers above and power must be delivered through a corresponding amount of vias. A part of power rail located between two vias is called *segment*.

High-frequency power droop occurs when multiple cells drawing current from the same power grid segment suddenly increase their current demand. If the current cannot be provided quickly enough from other parts of the chip, power starvation results in a voltage drop. In contrast to low-frequency power droop, this is a highly localised and transient phenomenon: one of the involved cells is slowed down for one clock cycle. Details on the electrical modelling of high-frequency power droop can be found in [1].

High-frequency power droop is very hard to debug or diagnose because it is nearly impossible to reproduce the error if the power droop conditions are not targeted explicitly. However, identified high-frequency power droop points to either a design issue (inadequate power rail sizing) or to a failed via of the power grid, i.e. a systematic manufacturing defect rather than random noise. Thus, it is important to target power droop systematically.

3.2 A test method for power droop

The test procedure is performed by applying a sequence of input patterns that imposes worst possible power droop and exposes its presence. First, low-frequency power droop is induced by creating a global $\frac{dI}{dt}$ event stretching over several cycles. When the voltage droop is most severe due to power starvation caused by the low-frequency power droop, high-frequency power droop is imposed on a victim signal. Combined power starvation due to both low-frequency and high-frequency power droop leads to an increased switching delay on the victim. Finally, the last input vector of the sequence must be able to sensitise a path from the victim to an observable output such that the faulty effect can be observed.

In order to create a global $\frac{dI}{dt}$ event, the amount of current the circuit draws from the VRM must change rapidly. This global current demand is a function of the inputs applied to the circuit. In deep-submicron CMOS, current is consumed for switching events of the gates and for leakage. However, the extent of pattern-dependent variation in leakage current is so small that we ignore it in this work and assume that the leakage current is a constant offset which cannot be influenced. In contrast, the switching current is completely determined by the input sequence, as it is given by the number of switching events in the circuit, i.e. signals changing their logic value from 0 to 1 or vice versa. The test sequence for imposing worst-case low-frequency power droop is composed of two sub-sequences $l_1 l_2 \dots l_M$ (called **low-activity sequence**) and $h_1 h_2 \dots h_N$ (called **high-activity sequence**). Applying the low-activity sequence should minimise the global switching activity while the application of the high-activity sequence should maximise it. The global switching activity between two cycles can be measured by $\frac{S}{l} \cdot 100\%$, where S is the number of switching events and l is the number of signals in the circuit. In general, switching events on different lines consume different amounts of power. This can be modelled by weighting the switching activity on a node, e.g. by the load it drives. In this work no such weighting is employed, but it can be easily integrated into the presented

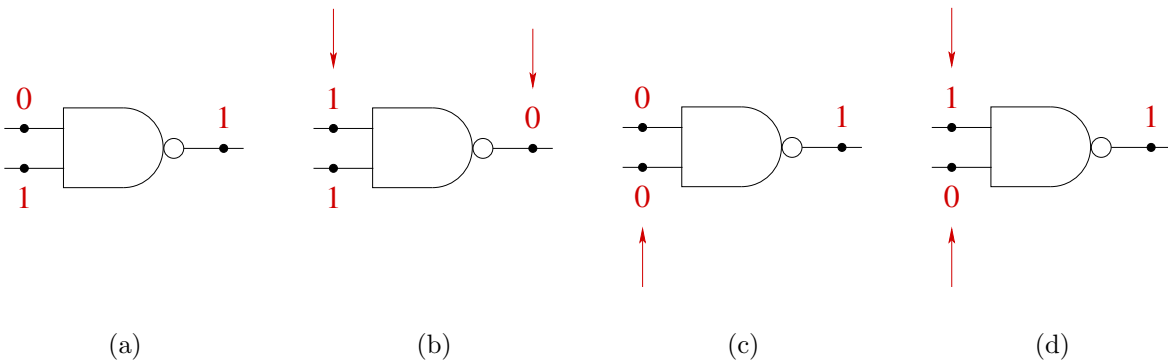


FIGURE 3.5: HIGHEST SWITCHING ACTIVITY CANNOT BE 100%

framework. Note that, in general, a global switching activity of 100% cannot be reached at all. Take for example the NAND2 gate illustrated in Figure 3.5. Figures 3.5(b), 3.5(c) and 3.5(d) show that, starting at the state illustrated in Figure 3.5(a), at most two of the three observed signals can switch in one cycle.

Worst-case high-frequency power droop will take place if the victim and several aggressors, which are signals driven by logic cells that draw power from the same segment of the power grid as the cell driving the victim, undergo the same transition simultaneously. Then, a significant amount of current must be transported to or from a single segment of the power grid through a series of resistive and inductive vias. How aggressors can be determined and chosen will be precised later in this work. In general, it may not be possible to impose the desired transition on all the aggressors simultaneously because of logic implications between them (e.g. if one of the aggressors feeds an inverter driving another aggressor). Hence, it is required that **as many aggressors as possible** switch in the same direction as the victim. High-frequency power droop leads to a delay fault on the victim which must be propagated to an observable point. Consequently, in order to test the high-frequency power droop, we require a test pair t_1t_2 that induces the desired transitions on victim and aggressors and that detects the transition fault on the victim.

The focus of this work is the generation of the test sequence. In the following chapter the ATPG problem is formalised. The next two chapters introduce the designed algorithm and implementation issues.

4

ATPG PROBLEM FORMULATION

An algorithm is to be designed and implemented that, being given the following arguments:

- a combinational or a full-scan sequential circuit C (circuit net-list on gate level) with $k_p \geq 1$ primary and $k_s \geq 0$ secondary inputs (cf. Figure 4.1),
- a victim signal and a set of aggressor signals,
- two integers $M \geq 1$ and $N \geq 1$,
- and the direction of a transition T (either $0 \rightarrow 1$ or $1 \rightarrow 0$);

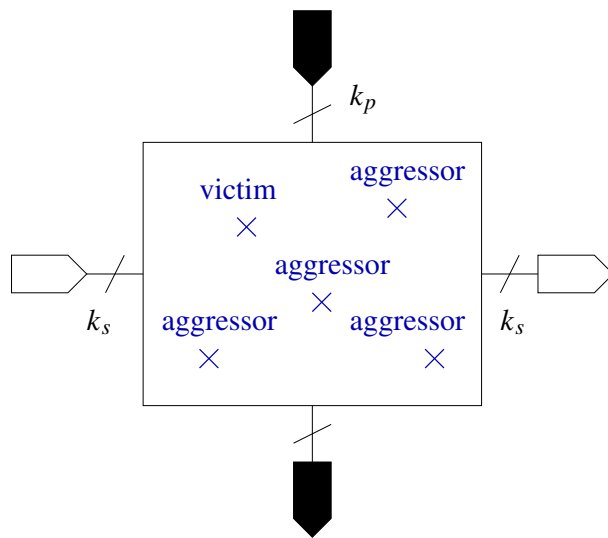


FIGURE 4.1: PROBLEM FORMULATION: GIVEN CIRCUIT

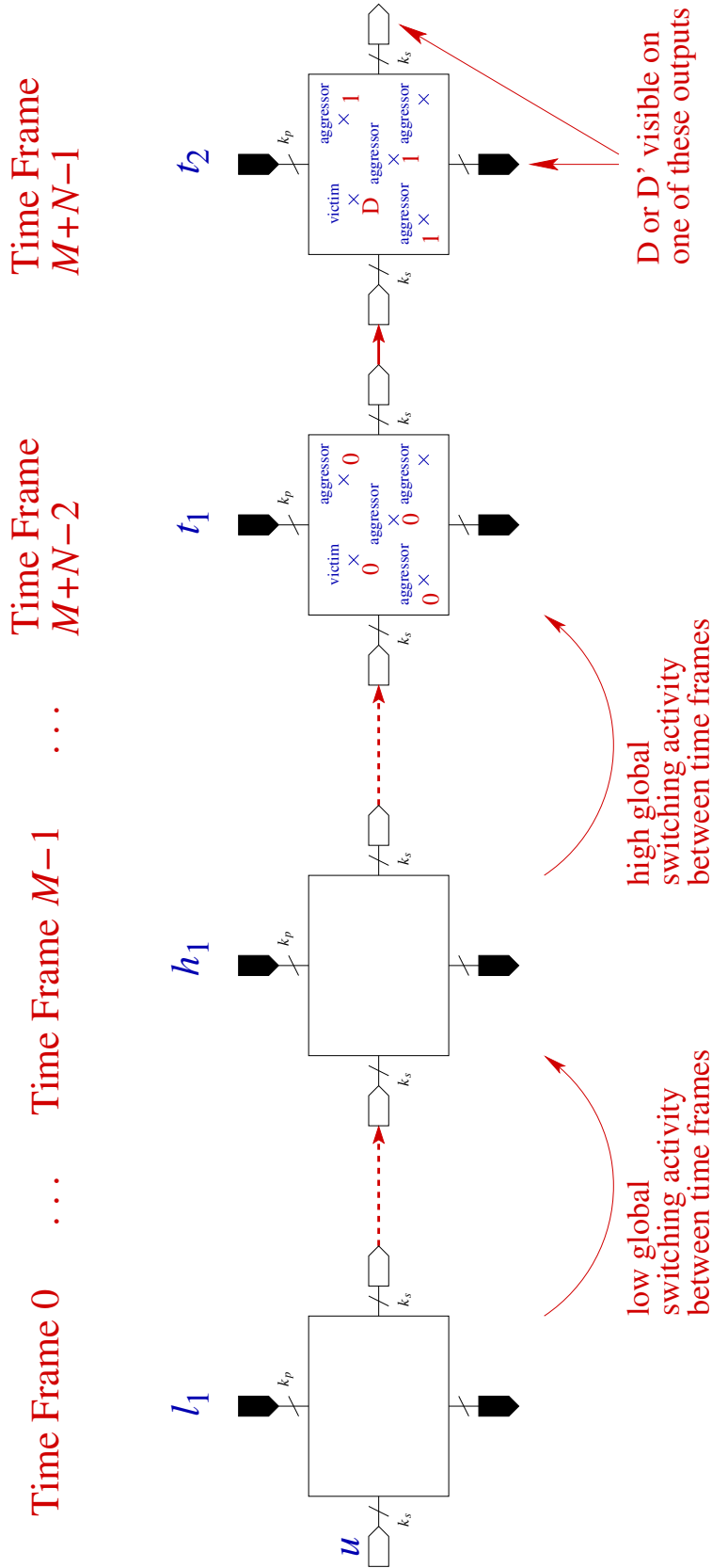


FIGURE 4.2: PROBLEM FORMULATION: CIRCUIT UNDER APPLICATION OF THE GENERATED TEST SEQUENCE, EXAMPLE FOR $T = 0 \rightarrow 1$

is able to generate a scan-in vector u of length k_s and a sequence of $M + N$ input vectors

$$\underbrace{l_1 \ l_2 \ l_3 \ \dots \ l_{M-1}}_{\text{low-activity sequence}} \underbrace{h_1 \ h_2 \ h_3 \ \dots \ h_{N-1}}_{\text{high-activity sequence}} \underbrace{t_1 \ t_2}_{\text{test pair}}$$

each one of them of length k_p , such that scanning in the generated vector u (if C is sequential) and applying the generated sequence to C for $M + N$ cycles brings forth the following behaviour (cf. Figure 4.2):

Constraint 1: the victim undergoes the transition T in the last two time frames (due to the application of the test pair);

Constraint 2: the error value (D if T is a rising transition, D' else) high-frequency power droop may cause on the victim in the last time frame is propagated to a P_OUT or an S_OUT in that time frame (due to the application of test pattern t_2);

Constraint 3: as many aggressors as possible undergo the transition T in the last two time frames (due to the application of the test pair);

Constraint 4: the global switching activity is as low as possible between time frame 0 and time frame $M - 1$ (i.e. during the application of the low-activity sequence);

Constraint 5: the global switching activity is as high as possible between time frame $M - 1$ and time frame $M + N - 2$ (i.e. during the application of the high-activity sequence).

Constraints 1 and 2 ensure that the victim undergoes the desired transition and that the delay imposed by power droop on the victim (transition fault) is detected. These two constraints must be satisfied exactly as they are formulated.

Constraint 3 creates worst-case high-frequency power droop. Constraints 4 and 5 help to induce the largest possible $\frac{dI}{dt}$ event required for worst-case low-frequency power droop. Fault detection takes place when the combined effects of low-frequency and high-frequency power droop impact the delay on the victim. These constraints are flexible, they only demand satisfaction in as many cases as possible. These constraints might also be conflicting in that an assignment necessary to maintain high switching activity (Constraint 5) may prevent aggressor signals from getting values desired for the satisfaction of Constraint 3. Consequently, the problem may have multiple solutions with different degrees of satisfaction of Constraints 3, 4 and 5.

5

AUTOMATIC TEST PATTERN GENERATION

The procedure that generates the test sequence for power droop is based on the D-algorithm, which is a well known ATPG method ([3], [2]). The main idea is that the D-algorithm is run in order to generate a test pair that satisfies Constraints 1 and 2. These two constraints have to be satisfied exactly as they are formulated on Page 23. The algorithm is capable of generating new assignment tasks on-the-fly depending on previous assignments. These additional dynamically generated assignments help satisfying Constraints 3, 4 and 5 in the best possible way. Due to its nature the algorithm has been called **dynamically constrained D-algorithm**.

5.1 The conventional D-algorithm

The conventional D-algorithm is used to generate tests for stuck-at faults in a combinational circuit with controllable primary inputs and observable primary outputs. The algorithm manages an `assignment queue`. Each element of the queue (`assignment`) is a task the algorithm has to perform. Before starting the algorithm, initial assignments are inserted into the queue. Then the algorithm removes the assignments from the queue one by one and processes them by assigning values to signals as directed by the assignments, and by calculating implications, which may lead to the insertion of new assignments into the queue. The algorithm is successful when the queue becomes empty, i.e. when all original tasks and all dynamically generated new tasks have been performed successfully.

An assignment is a triple (s, v, d) , where s is a signal, v a logic value (0, 1, D or D')¹ and d a propagation direction (\leftarrow or \rightarrow). For example, the assignment $(s, 1, \leftarrow)$ means that the algorithm has to `justify` the value 1 on signal s ; the assignment (s', D', \rightarrow) means that the algorithm has to `propagate` the value D' on signal s' towards a P_OUT. Assignments of the form (s, D, \leftarrow) and (s, D', \leftarrow) are invalid and therefore not accepted by the algorithm because it is neither possible nor required to justify error values.

¹For the meaning of the error values D and D', please refer to Chapter 2.

Justifying a signal s 's value v means setting the values of s 's predecessors (the input signals of the cell driving s) such that s gets the value v . The values s 's predecessors get have to be justified, too. This action is repeated until the values of one or several P_INs are set such that the justified values are guaranteed. Sometimes, choosing the values for s 's predecessors may require making a decision. For example, if s is driven by an AND n and the value 0 has to be justified, only one of the cell's n inputs needs to get the value 0. The algorithm has to decide which input to assign the value 0.

Propagating a signal s 's value v means setting s 's value to v and calculating all implications of that assignment. If v is an error value (D or D'), it is also necessary to sensitise a path from s to a P_OUT, such that an error value can be observed on that P_OUT. For example, if s feeds a cell c of type AND n , the value 1 has to be justified on all other inputs of c , such that only s 's value can influence c 's output value. If c 's output signal is a P_OUT, nothing has to be done any more. Otherwise, this procedure has to be repeated until a P_OUT is reached. Propagating a signal's value may also require making decisions. In general, only one path between the signal s and a primary output needs to be chosen, but if s feeds more than one cell, there is more than one path the algorithm can try to sensitise. The algorithm has to decide which drain cell of s to try first.

Whenever the algorithm makes a decision, the algorithm continues its job in the way described above, until either the assignment queue is empty and no further decisions are to be made, or until an inconsistency is identified. In the latter case, one of the earlier decisions is taken back and the search is continued after making a different decision (choosing a different cell input in the justification case, or a different drain cell in the propagation case). This behaviour is called *backtracking*. If no decisions are left to backtrack, at least one of the tasks the algorithm has to perform remains unachieved, which means that the set of initial conditions is contradictory. Therefore, the fault is proven to be untestable.

Testing for a stuck-at 0 fault on a signal s requires the creation of an input vector that justifies the logic value 1 on s and that propagates the faulty effect D on s to a primary output by sensitising a path. This means, the initial assignments $(s, 1, \leftarrow)$ and (s, D, \rightarrow) have to be inserted into the queue. Analogously, testing for a stuck-at 1 fault on s requires inserting the initial assignments $(s, 0, \leftarrow)$ and (s, D', \rightarrow) . After that, the algorithm processes the assignment queue as described above.

As was said at the beginning of this section, the conventional D-algorithm was designed for combinational circuits with controllable primary inputs and observable primary outputs. Nevertheless, the algorithm can also be applied to sequential circuits that have been designed using the `full-scan` DFT² methodology. Applying the scan methodology means that the flip-flops or latches are designed and connected in a manner that enables two different modes of operation. In the `normal` mode, the flip-flops are connected to the combinational logic block exactly as the design of the circuit requires it. In the `test` mode, the flip-flops are reconfigured such as to form one or more shift-registers called

²Design for Testability (DFT) is a name for design techniques that add certain features to the designed IC, the aim of this being making it easier to develop and to apply manufacturing tests. For more on this topic see, for example, [4, Chapter 11].

scan chains. During the normal mode, the response at the state outputs is captured in the flip-flops. These values can be observed by switching the circuit to the test mode by clocking the scan chains and observing the output of the last flip-flop in each chain. Furthermore, values to be applied at the state inputs in the subsequent test may be simultaneously shifted into the flip-flops. Hence, for the purpose of test development, the state or secondary inputs and outputs can be treated as being similar to primary inputs and outputs, respectively. Scan is called full-scan if all flip-flops in the circuit are accessible in this way. This makes it possible to apply the D-algorithm as explained in this section.

5.2 Dynamically constrained D-algorithm

The problem this work deals with is different from stuck-at test generation in combinational circuits, thus requiring modifications of the conventional D-algorithm. These are not drastic modifications since the problem can be basically solved by letting the D-algorithm process a set of assignments that are given as they were defined in Section 5.1. Thus, first of all, the D-algorithm has to be implemented such that it just processes a given assignment queue. Before starting the D-algorithm itself, the assignment queue can be initialised independently with any number of assignments and with any combination thereof. This provides the flexibility to handle a wide range of different fault models.

The circuit is sequential and a sequence of $M+N$ vectors is required for testing. Using an adequate representation of the $M+N$ time frames of the circuit also reduces the need for large modifications of the D-algorithm. This is handled by creating an $(M+N)$ -time-frame unfolding of the circuit. The primary inputs are all controllable in every time frame. Since full-scan is assumed but can be operated only for setting the initial state and observing the final state, secondary inputs (flip-flops) are controllable only in the first time frame and observable only in the last time frame. The fault effect emerges in the last time frame, so it can be observed on a primary or a secondary output of that time frame. Altogether, the $(M+N)$ -time-frame unfolding of the circuit can be handled as one big combinational circuit. The S_INs of the first time frame can be handled as P_INs of the unfolding, the S_OUTs of the last time frame can be handled as P_OUTs of the unfolding. Unfolding is the standard approach for sequential test generation ([2]). Please refer to Section 6.1 for implementation notes.

Finally, the D-algorithm has to be modified in a way that it can handle the five additional constraints listed on Page 23. Satisfying the first two constraints is easy and can be achieved without modifying the D-algorithm more than explained two paragraphs before (i.e. as to process a given assignment queue without questioning why the initial assignments are there). For example, refer to the problem depicted in Figure 4.2 (Page 22) where the victim's rising transition is tested. Constraints 1 and 2 can be satisfied by inserting only three assignments into the queue and letting the D-algorithm process them. The three assignments are:

- 1) (victim in time frame $M+N-2$, 0, \leftarrow),

- 2) (victim in time frame $M + N - 1$, 1, \leftarrow) and
- 3) (victim in time frame $M + N - 1$, D , \rightarrow).

If the algorithm fails to process these three assignments it can stop completely, since this failure means that either the tested transition cannot be imposed on the victim, or the faulty effect cannot be propagated. Thus, the conditions for testing high-frequency power droop are not given.

Constraints 3, 4 and 5 have to be handled differently as they are formulated in a flexible way (keywords “as many/low/high as possible”). The handling of these constraints is difficult because the achievable quality of a solution is not known. An assignment which is useful for satisfying a constraint may imply that other constraints are violated elsewhere in the circuit.

In theory, all these constraints can be approached by letting the D-algorithm process an adequate set of initial justification assignments:

Initial assignments for Constraint 3: (Referring to the example of Figure 4.2.) Insert, for each aggressor a , the assignments (a in time frame $M + N - 2$, 0, \leftarrow) and (a in time frame $M + N - 1$, 1, \leftarrow).

Initial assignments for Constraint 4: For each signal s in the circuit, insert the assignment (s in time frame i , v' , \leftarrow) if s has got the value v in time frame $i + 1$ and $i \in \{M - 1, M, \dots, M + N - 3\}$.

Initial assignments for Constraint 5: For each signal s in the circuit, insert the assignment (s in time frame i , v , \leftarrow) if s has got the value v in time frame $i + 1$ and $i \in \{0, 1, \dots, M - 2\}$.

However, the set of assignments for Constraint 3 is very large ($2 \cdot n_a$ assignments, where n_a is the number of aggressors). Besides, in most cases, the aggressors will all be in the physical neighbourhood of the victim which means that there is a possibly high dependence between their logic values. Thus, the probability that no solution for the problem of processing all these initial assignments doesn't exist at all, is very high. Concerning Constraints 4 and 5, the presented sets of initial constraints are not only extremely large ($M \cdot l$ assignments for Constraint 4 and $N \cdot l$ for Constraint 5, where l is the number of signals in the circuit). These cannot even be generated before launching the D-algorithm since the generation of the assignments for signals in a time frame i depends on the values those signals have got in time frame $i + 1$. Before launching the D-algorithm those values are still not defined.

Thus, the D-algorithm has to be modified once more. The approach starts with only few necessary initial assignments, thus increasing the probability that a solution for the problem exists. Then, the modified D-algorithm must be provided with a mechanism that permits on-the-fly generation of new assignments that should be performed, given the assignments that have been made before. However, these dynamically generated assignments don't need to be performed at any cost. Whenever an assignment is to be

inserted that either satisfies or violates a desired constraint, the satisfying assignment is tried first. Only if this assignment leads to an inconsistency will the ATPG procedure try the violating assignment. In particular, the following three rules have to be observed:

Rule 1:

- (a) Assume a rising transition on the victim. When a decision is made on an aggressor in time frame $M + N - 2$ (under vector t_1), assigning that aggressor the logic value 0 is always tried first. In time frame $M + N - 1$ (under vector t_2) logic 1 is always assigned first.
- (b) Assume a falling transition on the victim. When a decision is made on an aggressor in time frame $M + N - 2$ (under vector t_1), assigning that aggressor the logic value 1 is always tried first. In time frame $M + N - 1$ (under vector t_2) logic 0 is always assigned first.

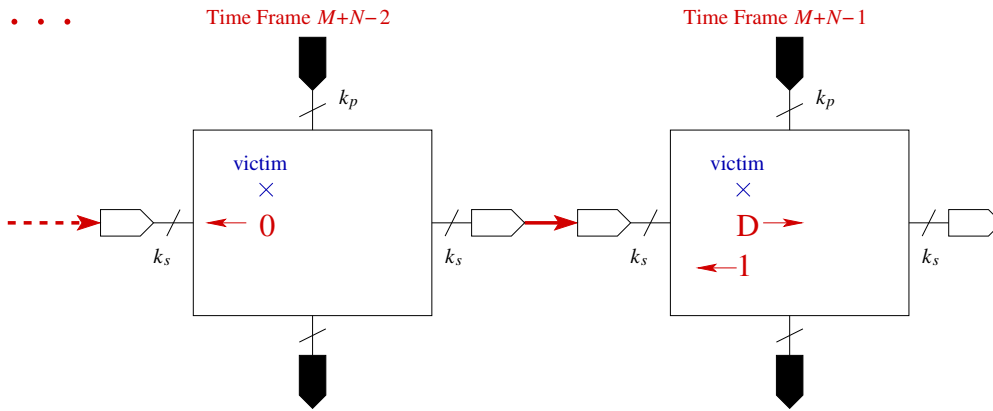
Rule 2: For justification: when selecting which signal to make a decision on, signals in later time frames are to be preferred.

Rule 3:

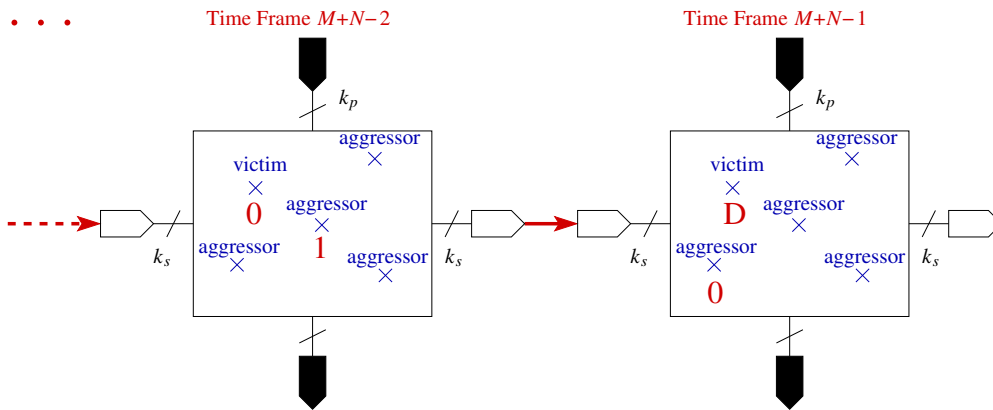
- (a) Suppose that the decision is made on a signal s in time frame i , $i \in \{M - 1, M, \dots, M + N - 3\}$. If s has already got the logic value v in time frame $i + 1$, insert the assignment (s in time frame i , v' , \leftarrow) into the queue. Only if this assignment leads to a conflict, try the assignment (s in time frame i , v , \leftarrow). If this leads to a conflict, too, leave signal s unassigned.
- (b) Suppose that the decision is made on a signal s in time frame i , $i \in \{0, 1, \dots, M - 2\}$. If s has already got the logic value v in time frame $i + 1$, insert the assignment (s in time frame i , v , \leftarrow) into the queue. Only if this assignment leads to a conflict, try the assignment (s in time frame i , v' , \leftarrow). If this leads to a conflict, too, leave signal s unassigned.

The rationale of Rule 1 is to satisfy Constraint 3 as well as possible. Rule 2's purpose is to facilitate the application of Rule 3. Rule 3 helps satisfying Constraints 4 and 5.

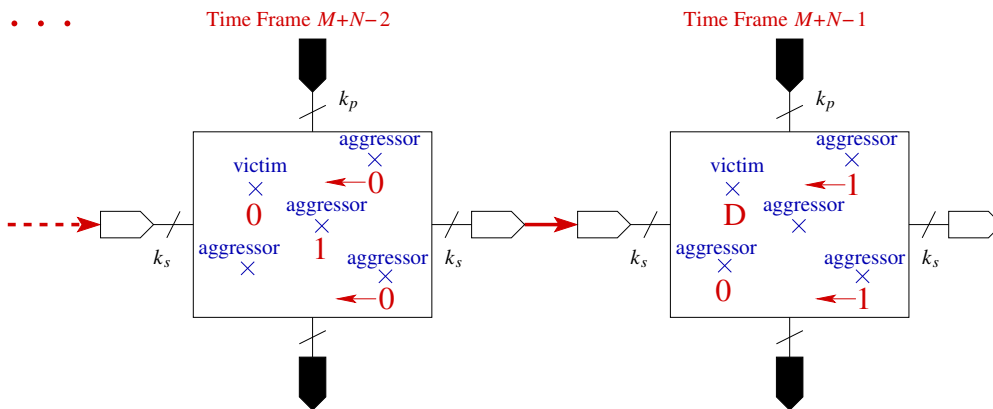
The proposed approach is heuristic, i.e. it may not yield the optimal result. However, it is not clear what the optimal solution is anyway, because Constraints 3 through 5 may be conflicting in that an assignment necessary to maintain high switching activity (Constraint 5) may prevent aggressor signals from getting values desired for the satisfaction of Constraint 3. Consequently, the problem may have multiple solutions with different degrees of satisfaction of Constraints 3, 4 and 5.



(a) FIRST STEP, SATISFYING CONSTRAINTS 1 AND 2



(b) AFTER THE FIRST STEP



(c) SECOND STEP, TRYING TO SATISFY CONSTRAINT 3

FIGURE 5.1: ATPG, AN EXAMPLE

5.3 Handling large M and N values

This section not only introduces a heuristic for the handling of large M and N values. It also explains in detail how the implemented program works.

Although the algorithm is adequate for mid-size circuits, its scalability may require an improvement to handling large circuits and large M and N values. There is a variety of improved techniques in the basic ATPG algorithm and add-ons such as diverse learning strategies. One further option is to consider subsequences of the global test sequence $l_1l_2\dots l_{M-1}h_1h_2\dots h_{N-1}t_1t_2$. For instance, let the sequence be $l_1l_2l_3l_4h_1h_2h_3h_4t_1t_2$. One could first generate a sequence for $l_1l_2l_3$ and the scan-in vector u taking all the relevant constraints into account. Let u_3 be the circuit state after three cycles. Then, test generation is run for subsequence $l_4h_1h_2$, with secondary inputs set to u_3 rather than being controllable. Let u_6 be the resulting state. Finally, test generation is run for $h_3h_4t_1t_2$ with secondary inputs set to s_6 . This technique reduces one ATPG run for a large number of time frames to several runs for shorter numbers of time frames. In general, this approach could fail to generate a sequence the originally proposed ATPG procedure could produce, or the quality of the obtained sequence may be worse. The reason for this is that, when the D-algorithm is working on a certain set of time frames, it cannot backtrack over previously processed sets of time frames.

A variation of this strategy was used when implementing the prototype for this work. By means of an example it is easier to explain how this procedure works. For instance, let us test the rising transition on the victim. The different steps are illustrated in Figures 5.1 and 5.2.

5.3.1 Step 1

The first step is illustrated in Figure 5.1(a). In this step the algorithm tries to satisfy Constraints 1 and 2. Only three initial assignments have to be inserted into the queue:

- 1) (victim in time frame $M + N - 2$, 0, \leftarrow),
- 2) (victim in time frame $M + N - 1$, 1, \leftarrow) and
- 3) (victim in time frame $M + N - 1$, D , \rightarrow).

After that the D-algorithm is run, but the algorithm only works on a two-time-frame unfolding of the circuit that corresponds to the last two time frames. The secondary inputs in time frame $M + N - 2$ are thus treated as primary inputs. Some aggressors might get, in one or both time frames, a value that prevents them from undergoing the same transition as the victim (see Figure 5.1(b)), but performing the three initial assignments listed above has got the higher priority. However, whenever possible, the algorithm will try to let the aggressors unassigned or to assign them a value according to Rule 1. In this step the algorithm also respects Rule 2.

If the algorithm isn't able to perform the three assignments mentioned above, it terminates unsuccessfully since the conditions for testing for high-frequency power droop are not given if the tested transition cannot be justified on the victim or if the error effect cannot be made visible.

5.3.2 Step 2

In the second step the D-algorithm is started again on the same two-time-frame unfolding as in the first step. The D-algorithm has been implemented such that the assigned values are managed by an instance that is independent of the D-algorithm. That means that the new D-algorithm instance will respect all specified signal values that were determined in the first step. This is equivalent to having one single instance of the D-algorithm performing both the first and the second step and being able to dynamically generate the start assignments for the second step. The implementation was done this way because having several short-running instances instead of one long-running instance of the D-algorithm permits a better management of memory resources. Besides, if working like this, the original D-algorithm has to be modified only slightly in order to make it capable of satisfying the additional constraints the problem this work deals with presents.

This step's job is trying to satisfy Constraint 3. The algorithm tries to justify the tested transition (in this example, the rising one) on all free aggressors. Free aggressors are those the first step hasn't assigned a value that prevents them from undergoing the tested transition (in this example, logic 1 in time frame $M + N - 2$ or 0, D or D' in time frame $M + N - 1$). For each free aggressor a , two assignments have to be inserted into the queue: (a in time frame $M + N - 2$, 0, \leftarrow) and (a in time frame $M + N - 1$, 1, \leftarrow). Figure 5.1(c) shows the set of all these assignments (referring to the current example). As was said before, Constraint 3 is flexible. If one or several of these assignments cannot be performed, the algorithm will still run and try to perform as many of them as possible.

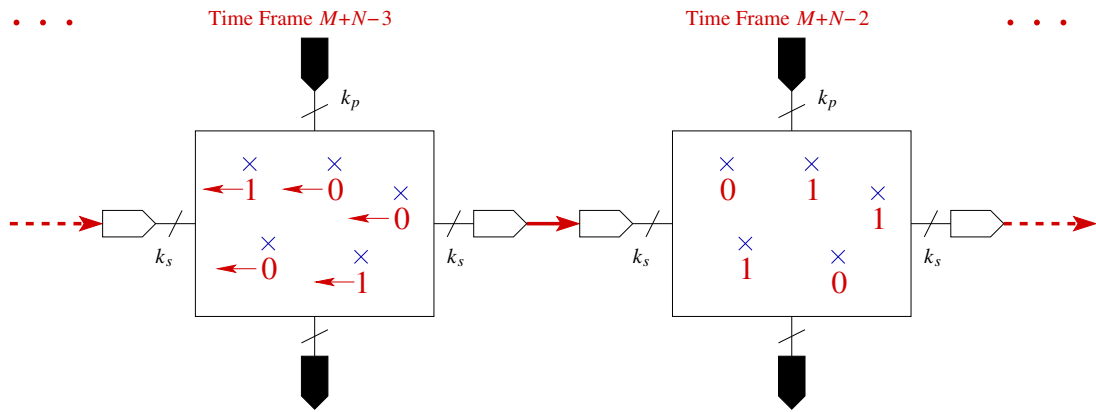
Not all these assignments can be inserted at once before starting the new D-algorithm instance. The problem could be overconstrained and get thus unsolvable. There is a wide range of approaches of what combinations of start assignments to try at all and what combinations to try first. It would even be possible to employ different learning techniques or a genetic algorithm. Here, since the overall test generation procedure has still a lot of work to do, a simple greedy heuristic has been used. The D-algorithm tries to justify the desired transition on only one free aggressor at a time. If the D-algorithm is successful for one aggressor, the new determined signal values are recorded in order to respect them. Then the D-algorithm is started again for the next free aggressor. If the D-algorithm fails to justify the desired transition on one aggressor, the old circuit's state is recovered and the D-algorithm is started again for the next free aggressor. This is repeated until all aggressors have been tried.

If the algorithm isn't able to justify the tested transition on any of the aggressors, it terminates unsuccessfully since the conditions for testing for high-frequency power droop are not given.

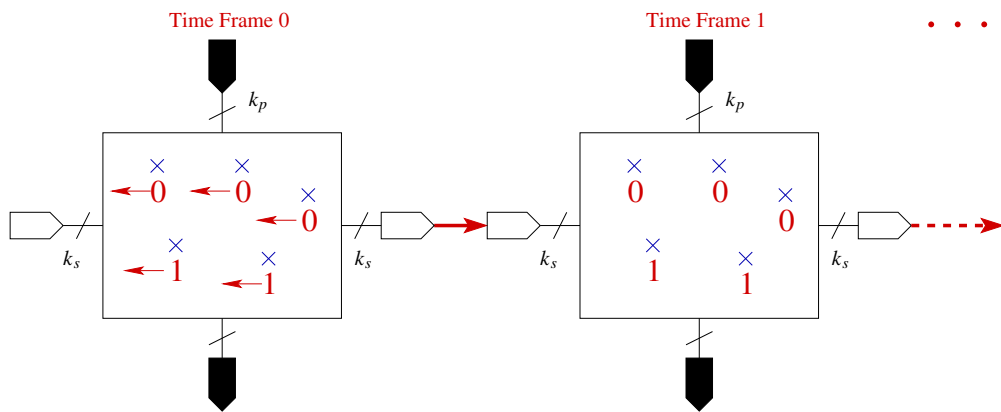
5.3.3 Step 3

In the third step the high-activity sequence is generated and the algorithm tries to satisfy Constraint 5. Since the sub-sequence generated in this step is intended for the test of low-frequency power droop, the algorithm doesn't need to treat the victim and the aggressors in a special way any more (which implies that Rule 1 does also not need to be observed any more).

During the generation of the high-activity sequence the algorithm never works on more than two time frames at a time. The algorithm begins working on a two-time-frame unfolding of the circuit representing time frame $M + N - 3$ and time frame $M + N - 2$ (see Figure 5.2(a)). Since the first two steps are applied only on time frame $M + N - 2$ and time frame $M + N - 1$, the signals in time frame $M + N - 3$ are all still unassigned.



(a) THIRD STEP, TRYING TO SATISFY CONSTRAINT 5



(b) LAST STEP, TRYING TO SATISFY CONSTRAINT 4

FIGURE 5.2: ATPG, AN EXAMPLE

But the signals in time frame $M + N - 2$ are initialised with the values that were recorded after the execution of the first two steps.

First, the algorithm tries to set all free signals in time frame $M + N - 2$ to 1 or 0. Free signals are those that haven't got logic 1 or logic 0 in that time frame. For each such signal the value v' is tried if the signal's got the value v in time frame $M + N - 1$ (if the signal is free in time frame $M + N - 1$, v is chosen at random). If performing that assignment is not possible, the value v is tried such that the majority of signals gets a specified value. If performing this second assignment is also not possible, the signal is left unassigned.

After this has been done, it can be assumed that as many signals as possible have got a specified value in time frame $M + N - 2$. Now, for each signal s that isn't free in time frame $M + N - 2$ (let v be the value of s in that time frame), the assignment (s in time frame $M + N - 3$, v' , \leftarrow) should be inserted into the queue before launching a new instance of the D-algorithm. Since there are too many assignments, the same greedy heuristic used in Step 2 is applied here: only one such assignment is tried at a time. While trying to justify all these assignments, the D-algorithm always observes Rules 2 and 3.

After this procedure has been applied to time frames $M + N - 3$ and $M + N - 2$, the algorithm repeats the job on a two-time-frame unfolding of the circuit representing time frame $M + N - 4$ and time frame $M + N - 3$; after that, on a two-time-frame unfolding representing time frame $M + N - 5$ and time frame $M + N - 4$. The algorithm goes on like this until the two-time-frame unfolding representing time frames $M - 1$ and M has been processed.

5.3.4 Step 4

In the last step the low-activity sequence is generated and the algorithm tries to satisfy Constraint 4. First, a two-time-frame unfolding of the circuit representing time frame $M - 2$ and time frame $M - 1$ is processed in a similar way as time frames $M + N - 3$ and $M + N - 2$ were processed in Step 3. Then the job is repeated for the two-time-frame unfolding of the circuit representing time frames $M - 3$ and $M - 2$. The algorithm goes on like this until the two-time-frame unfolding representing time frames 0 and 1 has been processed (see Figure 5.2(b)). There are three differences between the third step and this one.

- 1) In contrast to the third step, this step tries to assign a signal s in a time frame i ($i \in \{0, 1, \dots, M - 2\}$) the value v if the value of that signal is v in time frame $i + 1$.
- 2) If the given circuit is combinational, the algorithm processes the two-time-frame unfolding of the circuit representing time frame $M - 2$ and time frame $M - 1$. This is necessary in order to guarantee that as many signals as possible get a specified value in time frame $M - 1$. But, after that, no such processing is needed any more. All signal values in time frame $M - 2$ are simply copied without modification into

all time frames 0 through $M - 3$: the algorithm sets each signal's value to v in all time frames 0 through $M - 3$, where v is the value that signal has got in time frame $M - 2$. After doing so the algorithm can terminate.

- 3) If the given circuit is sequential, after a two-time-frame unfolding representing time frames i and $i + 1$ has been processed ($i \in \{0, 1, \dots, M - 2\}$), the algorithm checks if all signals have got the same value in time frame i and in time frame $i + 1$. If that is the case, all signal values in time frame i are simply copied without modification into all time frames 0 through $i - 1$, exactly as done for combinational circuits. After doing so the algorithm can terminate. If not all signals have got the same value in time frame i and in time frame $i + 1$, the algorithm goes on processing the unfolding composed of time frames $i - 1$ and i in the normal way.

6

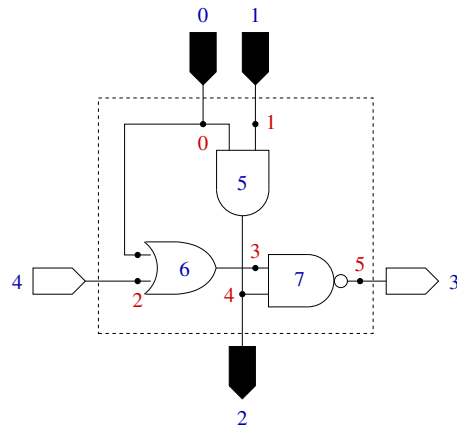
IMPLEMENTATION OF THE ATPG PROCEDURE

In order to obtain experimental results, the method proposed in Chapter 5 was implemented using C++ and the circuit representation library `test_circ`. It is not possible to write about all implementation details in this work, so we can introduce only the basic concepts of the two most important parts of the implementation. Those are the representation of unfolded circuits and the modifications of the D-algorithm. For more details, please refer to the source files which are delivered on the attached CD-ROM, they are full of extensive commentaries that will help understanding how the different classes' member functions work. The header files (also on the CD-ROM) contain complete user documentation.

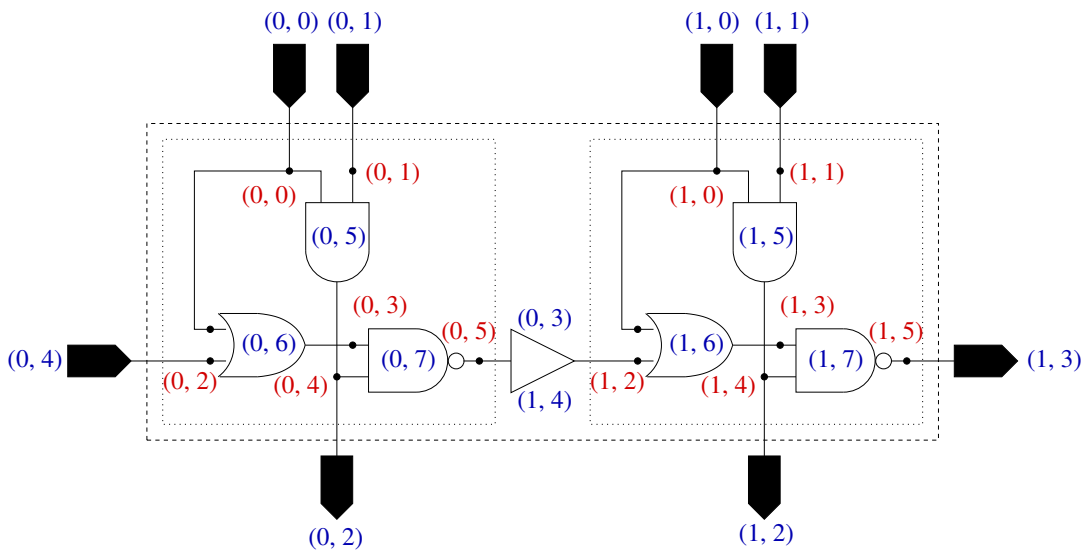
6.1 Unfolded circuits

As was already said in Chapter 5, the main difficulty of the test pattern generation is that not just one test pattern per fault has to be created, but a sequence of $M + N$ test patterns per fault. Generating only two input vectors is already a complicated task. In this work, where the number $M + N$ can easily become very large, the used approach is to unfold the circuit $M + N$ times and to perform well-known sequential ATPG while observing the additional constraints introduced on Page 23. Since the test generation is based on the D-algorithm which works on combinational circuits, there must be a way of representing unfolded circuits and treating them as big combinational circuits.

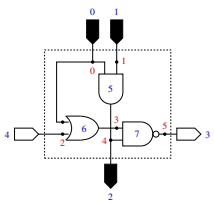
Since the benchmarks we used to test the ATPG algorithm are given in the format the circuit representation library `test_circ` understands, this library was used for basic circuit representation. An object of class `test_circ` represents a combinational or a sequential circuit. Chapter 2 contains a short explanation of how the library `test_circ` manages the internal representation of circuits. Recall that each cell is identified uniquely by an integer number called `c_ord` and that each signal is identified uniquely by an integer number called `s_ord`. The small fictive example circuit in Figure 6.1(a) illustrates this concept once more.



(a) test_circ CIRCUIT



(b) unfcirc CIRCUIT REPRESENTING THE CORRESPONDING 2-TIME-FRAME UNFOLDING

<p>Reference test_circ circuit C</p> 	<pre> int NUMBER_OF_INSTANCES = 2; sig_value SIGNAL_VALUES[6*2]; short SIGNAL_FLAGS[6*2]; short CELL_FLAGS[8*2]; </pre>
--	---

(c) INTERNAL REPRESENTATION OF THE CIRCUIT IN (b)

FIGURE 6.1: CIRCUIT REPRESENTATION WITH test_circ AND unfcirc

The class `test_circ`'s member functions can carry out a wide variety of different tasks. The most important functions are those which permit traversing the circuits by:

- returning a given cell's type, its number of inputs and outputs;
- returning a given signal's number of branches, its source cell and its drain cells;
- a given cell's input and output signals, its successors and its predecessors.

For the representation of unfolded circuits, instead of using the built-in circuit creation mechanisms of `test_circ`, we opted to develop a new circuit representation class called `unfcirc` which is based on that library. Working on F time frames of a `test_circ` circuit C implies loading F copies C_1, C_2, \dots, C_F of that circuit and trying to manage all of them separately and managing how they are related to each other. This process can be automated by working with the new class `unfcirc`. Besides, working with the new class also makes it possible to manage the same job without having to load more than one copy of C into the memory. Therefore, this solution is more memory efficient.

The class's constructor takes the `test_circ` circuit C to work on and the number F of time frames that are to be managed. The circuit C is called `reference circuit` and the time frames or copies of C are called `instances`. The created object manages the different instances and how they are related to each other. The main properties of the class are:

- The `unfcirc` object is a big combinational circuit that is made up of F instances of the reference circuit C . Figure 6.1(b) illustrates this for two instances of the reference circuit shown in Figure 6.1(a).
- Each cell in the `unfcirc` circuit corresponds to a cell in the reference circuit in a certain time frame. Thus cells can be identified uniquely by a pair of integers $(I, \text{c_ord})$, where I is the number of the instance (time frame) the cell belongs to and `c_ord` is the cell's identification number in the reference circuit. The F instances of the reference circuit are numbered from 0 to $F - 1$.
- Like cells, signals are identified uniquely by a pair of integers $(I, \text{s_ord})$, where `s_ord` is the signal's identification number in the reference circuit.
- The reference circuit's outer S_INs (those belonging to the 0-th instance) and outer S_OUTs (those belonging to the $(F - 1)$ -th instance) are presented as P_INs and P_OUTs of the `unfcirc` circuit, respectively. (examples: cells $(0, 4)$ and $(1, 3)$ in Figure 6.1(b)).
- The reference circuit's inner S_INs (all but those belonging to the 0-th instance) and inner S_OUTs (all but those belonging to the $(F - 1)$ -th instance) are presented as BUFs of the `unfcirc` circuit (examples: cell $(1, 4)/(0, 3)$ in Figure 6.1(b)).

In this case, an instance's S_OUT and its corresponding S_IN belonging to the next instance are both replaced by one single cell of type BUF which can be addressed using two different identification pairs. Identification pairs are only needed to create new cell objects while members processing cells work on real cell objects, not on identification pairs. So this circumstance doesn't represent an inconsistency or ambiguity in the definition of the class.

Now let us point out the major advantage of working on an `unfcirc` circuit. Like in class `test_circ`, there are member functions that provide information necessary to traverse the `unfcirc` circuit. The user can work directly on the `unfcirc` circuit over an interface similar to that offered by `test_circ`, without having to access the reference `test_circ` circuit.

Let us illustrate these ideas by means of a simple example. The example is based on the circuit depicted in Figure 6.1(b). If signal (0,5) has the logic value 0 (which is equivalent to signal 5 having the value 0 in the 0-th time frame) and signal (1,0) has the logic value 0, too, an appropriate `unfcirc` member function can be called to obtain the drain cell of signal (0,5). Cell (0,3) is returned. Requesting that cell's output signal will return signal (1,2). Since cell (0,3)'s type is BUF, signal (1,2) must be assigned the logic value of signal (0,5), i.e. logic 0. Going on like this it is possible to find out that signal (1,5) must have the value 1. Altogether the algorithm has found out how a signal's value relates to its own value in the following time frame, but the algorithm has not noticed that different time frames were involved. The example also illustrates why the signal and cell identification system introduced above has been chosen: It is possible to discern `unfcirc` signal (i, k) from `unfcirc` signal (j, k) for those are fully different objects; at the same time, it is easy to know that both `unfcirc` signals represent the same signal in the reference circuit (signal k), but in different time frames.

An `unfcirc` circuit does not only manage the reference circuit and its different instances. It also maintains an array of logic values (0, 1, D, D', X or U) that records the logic value of each `unfcirc` signal in the circuit. At the beginning of Subsection 5.3.2 it has been pointed out that it may be useful for certain algorithms to let the circuit manage its own signal values. An `unfcirc` circuit also maintains eight multi-purpose binary flags for each signal and for each cell. This system of flags is used by the D-algorithm in order to implement the application of Rules 1 and 3 (defined on Page 29).

Let us particularise how an `unfcirc` circuit is represented internally (cf. example in Figure 6.1(c)). The only data that are stored by the `unfcirc` object are:

- a private copy of the reference `test_circ` circuit C ,
- the number F of managed instances (var. `NUMBER_OF_INSTANCES` in the figure),
- and the arrays of logic values and flags.

A cell object is a pair composed of an integer (the number of instance the cell belongs to) and a pointer to the corresponding `test_circ` cell in the reference circuit. Analogously, a signal object is a pair composed of an integer (the number of instance the signal belongs to) and a pointer to the corresponding `test_circ` signal in the reference circuit.

The advantages of such an implementation are that memory is saved because only one copy of the reference circuit has to be loaded, and that the initialisation of a new `unfcirc` circuit is very fast.

6.2 Implementation of the D-algorithm

6.2.1 Basic implementation

The implementation of the D-algorithm made for this work is based on the version presented in [2, Chapter 6], but let us remark that this implementation works only on `unfcirc` circuits. Since the D-algorithm is well-known, in this section, only the modifications necessary to satisfy the constraints defined on Page 23 are explained in detail. Once more, the attached CD-ROM contains all source files which are full of implementation commentaries.

In Section 5.1 we have already explained in detail how the conventional D-algorithm works in order to generate tests for stuck-at faults in a combinational circuit with controllable primary inputs and observable primary outputs. The main differences between the classical D-algorithm and the implementation for this work are:

- In this implementation the assignment queue can be initialised independently with any number of assignments and with any combinations thereof. This provides the flexibility to handle a wide range of different fault models.
- The classical D-algorithm terminates unsuccessfully if no primary output gets an error value. In this implementation the D-algorithm can be started as:

D-algorithm: in this case it behaves as explained above; or as

Justification algorithm: in this case the algorithm performs justification assignments exactly in the way the D-algorithm does. But when processing forward propagation assignments, the algorithm only calculates implications that follow without making decisions. That means, the algorithm doesn't try to propagate error values using path sensitisation. The algorithm will be successful even if there is no primary output with an error value.

- This implementation of the D-algorithm observes the three rules defined on Page 29.

Figure 6.2 shows in detail how the implemented D-algorithm works. As was said before the algorithm works on an assignment queue. Assignments are processed one by one. Processing an assignment means assigning a signal a value (if possible) and calculating implications. This is done by sub-procedure `imply_and_check()` which is described in Figure 6.3 using pseudo-code. As was explained in Section 5.1, performing

```

1 D_algorithm() BEGIN
2     Make private backup copies of the assignment queue,
   of both frontiers and of variable ERROR_AT_P.OUT.
3     Initialise a private signal-recovery queue and
   store the s.ord of all signals with value X.
4     if imply_and_check() returns FAILURE ; then
5         set VICTORY = NO
6     else
7         if ERROR_AT_P.OUT equals NO ; then
8             if propagation() is SUCCESSFUL ; then
9                 set VICTORY = YES
10            else
11                set VICTORY = NO
12            fi
13        else
14            if justification() is SUCCESSFUL ; then
15                set VICTORY = YES
16            else
17                set VICTORY = NO
18            fi
19        fi
20    fi
21    if VICTORY equals NO ; then
22        Restore, according to the backup data, the
   assignment queue, both frontiers, variable
   ERROR_AT_P.OUT and the logic value of all
   signals that have been assigned a new value.
23    fi
24    Delete backup data.
25    if VICTORY equals YES ; then
26        return SUCCESS
27    else
28        return FAILURE
29    fi
30 END

```

FIGURE 6.2: RECURSIVE D-ALGORITHM

```
1 imply_and_check() BEGIN
2   if assignment queue is empty ; then
3     return SUCCESS
4   fi
5   repeat
6     Remove first assignment (SIG, VAL, DIR) from the queue.
7     if SIG's current logic value is not X and not VAL ; then
8       return FAILURE
9     fi
10    Set SIG's logic value to VAL.
11    if DIR equals <-- ; then
12      Let CELL be SIG's source cell.
13      Depending on CELL's type and on the current values
        of CELL's input signals and CELL's output signal,
        imply new assignments and insert them into the
        assignment queue. If necessary, remove CELL
        from the D- or the J-frontier, or add it to the
        D- or the J-frontier.
14    else (i.e. DIR equals -->)
15      for each drain cell CELL of SIG ; do
16        Depending on CELL's type and on the current values
          of CELL's input signals and CELL's output signal,
          imply new assignments and insert them into the
          assignment queue. If necessary, remove CELL
          from the D- or the J-frontier, or add it to the
          D- or the J-frontier.
17      done
18    fi
19  until assignment queue is empty.
20  return SUCCESS
21 END
```

FIGURE 6.3: D-ALGORITHM, SUB-PROCEDURE `imply_and_check()`

```

1 propagation() BEGIN
2   if D-frontier is empty ; then
3     return FAILURE
4   fi
5   repeat
6     Remove first cell CELL from the D-frontier.
7     for each input S of CELL with value X ; do
8       Insert the assignment
        (S, NON_CONTROLLING_VALUE_OF_CELL, <-->)
        into the assignment queue.
9     done
10    if D.algorithm() is SUCCESSFUL ; then
11      return SUCCESS
12    else
13      Remove from the assignment queue all
        assignments inserted in Lines 7 - 9.
14    fi
15  until D-frontier is empty.
16  return FAILURE
17 END

```

FIGURE 6.4: D-ALGORITHM, SUB-PROCEDURE propagation()

```

1 justification() BEGIN
2   if J-frontier is empty ; then
3     return SUCCESS
4   fi
5   Remove first cell CELL from the J-frontier.
6   Sort the inputs of CELL such that trying the different inputs
   in the resulting order guarantees that Rules 1, 2 and 3
   (see Page 29) are applied.
7   According to the sorting made in Line 6, for each input S of CELL with value X ; do
8     Insert the assignment (S, CONTROLLING_VALUE_OF_CELL, <-->)
       into the assignment queue.
9     if D.algorithm() is SUCCESSFUL ; then
10      return SUCCESS
11    else
12      Remove assignment (S, CONTROLLING_VALUE_OF_CELL, <-->)
        from the queue and insert the assignment
        (S, NON_CONTROLLING_VALUE_OF_CELL, <-->) instead.
13    fi
14  done
15  return FAILURE
16 END

```

FIGURE 6.5: D-ALGORITHM, SUB-PROCEDURE justification()

a justification assignment may require deciding which input signal of a cell c to assign c 's controlling value¹. But this decision is not made by `imply_and_check()`, it is made by sub-procedure `justification()` (Figure 6.5) after `imply_and_check()` is ready; `imply_and_check()` inserts cell c into a queue called J-frontier such that `justification()` knows what cells to perform decisions on. Analogously, performing propagation assignments may require deciding which path from an error signal to a P_IN to sensitise. The decision is not made by `imply_and_check()`, it is made by sub-procedure `propagation()` (Figure 6.4) after `imply_and_check()` is ready; `imply_and_check()` inserts all cells with an error input into a queue called D-frontier such that `propagation()` knows what cells can be made part of a sensitised path. The algorithm exits successfully if:

- the assignment queue becomes empty (which means that all assigning tasks have been performed); and
- the J-frontier is empty (which means that all justification tasks which require making decisions have been performed); and
- variable `ERROR_AT_P_OUT` is true (which means that an error value has been propagated to a primary output). This variable is set by `imply_and_check()` or by `propagation()` if a signal feeding a cell of type P_OUT is assigned an error value.

Please note that, while sub-procedure `imply_and_check()` never makes decisions, `propagation()` and `justification()` never modify signal values. These two sub-procedures only insert new assignments into the queue. These new assignments are processed by the `imply_and_check()` sub-procedure belonging to the recursively called D-algorithm instance (see Line 10 in Figure 6.4 and Line 9 in Figure 6.5).

An instance of the D-algorithm is composed of an execution of `imply_and_check()` (see Line 4 in Figure 6.2) and, if `imply_and_check()` is successful, either an execution of `propagation()` (Line 8) or an execution of `justification()` (Line 14). `propagation()` and `justification()` are never executed in the same instance of the D-algorithm. Which of both is executed depends on the value of variable `ERROR_AT_P_OUT` (Line 7). This makes the implementation of the justification algorithm defined on Page 41 very easy. It suffices to set the variable `ERROR_AT_P_OUT` to YES before launching the normal D-algorithm. Thus no processing of the D-frontier takes place.

Some final remarks must be made on the pseudo-code presented in Figures 6.2 through 6.5. On Line 3 in Figure 6.2, signal-recovery queues are mentioned. Such a queue is used simply to store the identification numbers of signals whose logic values may be modified by the call of `imply_and_check()` in Line 4. This is necessary because the old signal values must be restored if `imply_and_check()` or `propagation()`

¹The term *controlling value* denotes logic 0 if the cell in question is an AND n or a NAND n , logic 1 if the cell is an OR n or a NOR n .

or `justification()` are unsuccessful. Restoring the old state of the different data structures is part of undoing a decision (backtracking).

The other remarks concern Line 6 in Figure 6.3, Line 6 in Figure 6.4 and Line 5 in Figure 6.5, where an assignment is taken from the assignment queue and a cell is taken from the D-frontier and a cell is taken from the J-frontier, respectively. The assignment queue and the J-frontier may be implemented as FIFO queues. In which order the assignments are processed and in which order the cells are taken from the J-frontier will, in general, not influence the algorithm's output. All assignments in the queue and all cells in the J-frontier must eventually be processed. In contrast to this, in which order the cells are taken from the D-frontier does not only influence the output of the classical D-algorithm (since, in general, not all paths starting at the output of a cell which is in the D-frontier need to be sensitised in order to propagate the error value to a primary output). Particularly, in which order the D-frontier cells are tried influences the output of the dynamically constrained D-algorithm, in that choosing a certain path to sensitise when trying to satisfy Constraint 2 may impede the satisfaction of Constraint 3. The sub-procedure presented in Figure 6.4 may remain unchanged if the D-frontier is implemented as a priority queue. In the next subsection we will discuss how to define the priority of a D-frontier cell in an adequate manner.

6.2.2 Applying Rules 1, 2 and 3

Finally, it is necessary to explain how the D-algorithm was modified such that it observes Rules 1, 2 and 3 (defined on Page 29). Rule 2 dictates how to behave when having to decide what signal to make a decision on. Rules 1 and 3 dictate which value to try first when having to decide which value a certain signal must get. The D-algorithm only makes cell-selecting (when propagating an error value) and signal-selecting decisions (when justifying certain values). That means, the D-algorithm never reaches a situation in which it must decide which value to assign to a certain signal. Nevertheless, it is still possible to modify the D-algorithm such that it can observe Rules 1 and 3. In fact, only two simple modifications suffice to guarantee that all three rules are observed. The first modification is in sub-procedure `justification()` (Lines 6 and 7 in Figure 6.5).

There is only one kind of signal-selecting decision the D-algorithm ever makes. If the output signal of a cell c of type AND_n or OR_n has got c 's controlling value, or if c is of type $NAND_n$ or NOR_n and its output signal has got the negation of c 's controlling value, in order to justify c 's output value, at least one of c 's input signals needs to be assigned c 's controlling value. The algorithm has to decide which input signal of c gets that value. If that decision leads to a conflict, a different signal has to be tried. The classical D-algorithm is allowed to try the different signals in any order. This implementation tries the different input signals in a special order. Just by selecting the signals in this order the three rules are applied. Before introducing this ordering it is better to first understand how the three different rules can be applied.

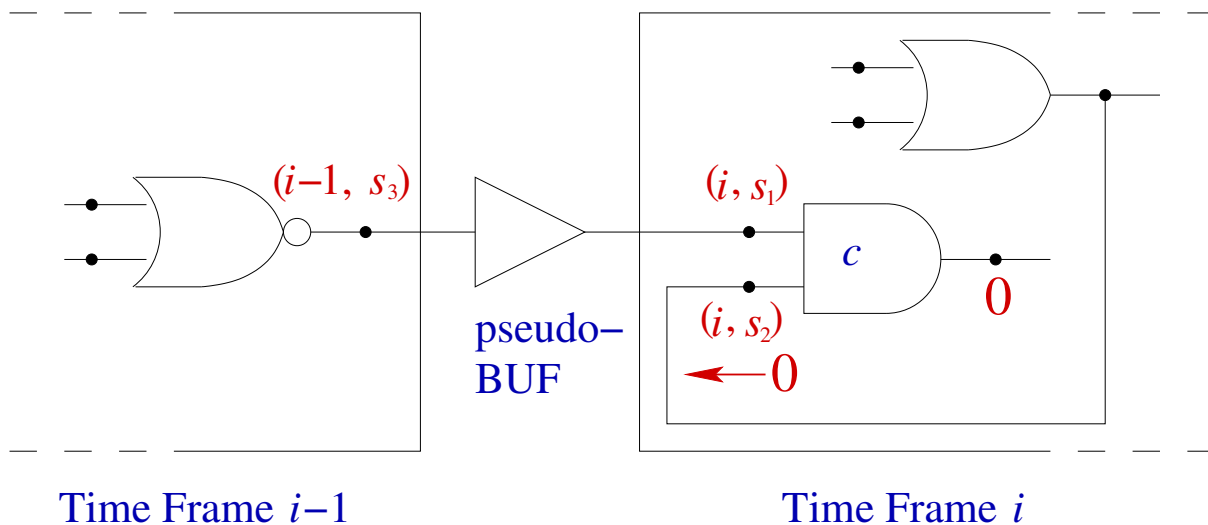


FIGURE 6.6: APPLICATION OF RULE 2

6.2.2.1 The application of Rule 2

Rule 2 dictates to prefer a signal in a later time frame when selecting which signal to make a decision on. As was explained above, the D-algorithm only selects signals when having to decide which input signal of a cell c to assign c 's controlling value. All input signals of a cell are always in the same time frame. However, there is a case in which Rule 2 can be applied. An example (Figure 6.6) illustrates this case. The output of cell c has got the value 0 and this value has to be justified. Hence one of both input signals of cell c must be assigned the value 0. Both signals are in the same time frame (as is always the case). However, since signal (i, s_1) and signal $(i - 1, s_3)$ are connected by a pseudo-BUF², the value of (i, s_1) depends exclusively on the value of $(i - 1, s_3)$ which is not in the same time frame as c . Thus, selecting signal (i, s_1) instead of (i, s_2) would be equivalent to making a decision on signal $(i - 1, s_3)$. If the algorithm observes Rule 2, it must select signal (i, s_2) because, when comparing $(i - 1, s_3)$ and (i, s_2) , (i, s_2) is in the later time frame.

Altogether, in order to respect Rule 2, it suffices to try signals which are not driven by a pseudo-BUF before signals which are driven by one.

²Recall that this implementation of the D-algorithm works only on `unfcirc` circuits. Recall also that class `unfcirc` presents the reference circuit's inner S_INs and inner S_OUTs as BUFs. These `unfcirc` cells are called here `pseudo-BUFs` for they do not correspond to `test_circ` BUFs in the reference circuit. Checking if an `unfcirc` cell of type BUF is a pseudo-BUF is easy: it is one if and only if its input signal (i_1, j_1) and its output signal (i_2, j_2) belong to different instances, i.e. if $i_1 \neq i_2$.

6.2.2.2 The application of Rules 1 and 3

Rule 1 states which logic value to try first on the aggressors in the last two time frames when having to decide which value to assign to them. Rule 3 states exactly the same, but it refers to all signals in the circuit and to all but the last two time frames. Therefore, Rules 1 and 3 do never need to be applied at the same time. Since both rules are similar in what they dictate one single modification of the D-algorithm suffices to guarantee that each one of these rules is applied whenever appropriate. As was said before, the D-algorithm never reaches a situation in which it must decide which value to assign to a certain signal. Despite this, it is possible to guarantee the application of Rules 1 and 3.

Let us explain by means of an example how Rule 1 can be applied. Suppose the algorithm is working on the last two time frames of the unfolding and that it has to select one of the input signals of a cell c and assign c 's controlling value to that signal. Let s_0, s_1, \dots, s_k be c 's input signals. For instance, let us assume that only s_0 and s_k are aggressors and that Rule 1 dictates that the value v should be preferred for those aggressors. If c 's controlling value is the value v (which means that at least one of the signals s_0, s_1, \dots, s_k must be assigned the value v), Rule 1 is applied by selecting signals s_0 and s_k before signals s_1, \dots, s_{k-1} . On the other hand, if c 's controlling value is the value v' (which means that at least one of the signals s_0, s_1, \dots, s_k must be assigned the value v'), Rule 1 is applied by selecting signals s_1, \dots, s_{k-1} before signals s_0 and s_k .

Rule 3 can be applied in exactly the same way as Rule 1. Suppose the algorithm is working on a time frame i and has to select one of the input signals of a cell c and assign c 's controlling value to that signal. Let s_0, s_1, \dots, s_k be c 's input signals. For instance, let us assume that Rule 3 dictates that the value v should be preferred for signal s_0 and that the value v' should be preferred for signal s_k . Additionally, let us assume that signals s_1, \dots, s_{k-1} haven't got a specified value in time frame $i + 1$ which means that no value has to be preferred for these signals in time frame i . If c 's controlling value is the value v (which means that at least one of the signals s_0, s_1, \dots, s_k must be assigned the value v), Rule 3 is applied by selecting signal s_0 before signals s_1, \dots, s_{k-1} (for s_0 should preferably get the value v) and by selecting signals s_1, \dots, s_{k-1} before signal s_k (for s_k should preferably not get the value v). On the other hand, if c 's controlling value is the value v' (which means that at least one of the signals s_0, s_1, \dots, s_k must be assigned the value v'), Rule 3 is applied by selecting signal s_k before signals s_1, \dots, s_{k-1} and these signals before signal s_0 .

Altogether, Rules 1 and 3 can be applied by letting the signal-selecting procedure select signals in an appropriate order. The ordering can be made according to a pair of flags that is recorded for each signal (and for each time frame) in the circuit. Each signal may be marked with up to one of two flags called `logic_1_preferred`, meaning that Rule 1 (or Rule 3) dictates that on the marked signal logic 1 should be tried first, and `logic_0_preferred`. The different time frames of the circuit are processed in a reverse order, i.e. beginning with the last one and ending with the first time frame. Before launching the D-algorithm on a certain time-frame, the overall ATPG procedure sets these flags for all signals in that time frame according to Rules 1 and 3.

6.2.2.3 Signal ordering for the signal-selecting procedure

Let v be a cell c 's controlling value. When deciding which input signal of c should get the value v , c 's input signals are tried in the following order:

- 1) Signals which are not driven by a pseudo-BUF and that are marked with flag `logic_v_preferred`.
- 2) Signals which are not driven by a pseudo-BUF and that are not marked with any flag.
- 3) Signals which are not driven by a pseudo-BUF and that are marked with flag `logic_v'_preferred`.
- 4) Signals which are driven by a pseudo-BUF and that are marked with flag `logic_v_preferred`.
- 5) Signals which are driven by a pseudo-BUF and that are not marked with any flag.
- 6) Signals which are driven by a pseudo-BUF and that are marked with flag `logic_v'_preferred`.

In this listing the `v` and `v'` in the flag names are metasyntactic variables. That means, `logic_v_preferred` stands for `logic_1_preferred` and `logic_v'_preferred` stands for `logic_0_preferred` if c 's controlling value is logic 1; or for `logic_0_preferred` and `logic_1_preferred`, resp., if c 's controlling value is logic 0. This ordering has been constructed according to the observations made in 6.2.2.1 and 6.2.2.2. It gives Rule 2 precedence over Rules 1 and 3. This is acceptable since Rule 2's purpose is to facilitate the application of Rule 3, as was explained in the end of Section 5.2.

6.2.2.4 The cell-selecting procedure and the application of Rule 1

So far, we have discussed how the `justification()` sub-procedure has to be modified such that all three Rules can be applied. But the `propagation()` sub-procedure does also make decisions. As was already explained in Section 5.1, propagating the error value of a signal s requires making a decision if s feeds more than one cell. `imply_and_check()` inserts all these drain cells into the D-frontier and `propagation()` decides which path to try first by choosing the cells from the D-frontier in a certain order.

At the end of Subsection 6.2.1 we pointed out that choosing the cells from the D-frontier in a certain order may affect the task of satisfying Constraint 3. Let us illustrate this with an example (see Figure 6.7). Suppose the ATPG procedure must try to impose the falling transition on as many aggressors as possible and suppose that there are only two cells c_1 and c_2 in the D-frontier.

- Let both cells be of type AND2 and let their output signals feed primary outputs of the last time frame.
- Let $s_{1,0}$ and $s_{1,1}$ be the input signals of c_1 , $s_{2,0}$ and $s_{2,1}$ be the input signals of c_2 .
- Let $s_{1,0}$ and $s_{2,0}$ have an error value, let $s_{1,1}$ be an aggressor and let $s_{2,1}$ be a non-aggressor signal.
- Suppose that both assignments $(s_{1,1}, 1, \leftarrow)$ and $(s_{2,1}, 1, \leftarrow)$ can be performed successfully.

If `propagation()` tries cell c_1 first, the D-algorithm performs the assignment $(s_{1,1}, 1, \leftarrow)$. Thus, c_1 's output gets an error value which means that cell c_2 doesn't need to be tried any more. However, $s_{1,1}$ has got the logic value 1, and this is bad because the generated test sequence should cause as many aggressors as possible to undergo the falling transition in the last two time frames. If `propagation()` chooses cell c_2 instead, it is signal $s_{2,1}$ and not signal $s_{1,1}$ that gets the logic 1. Since $s_{2,1}$ is not an aggressor, this assignment doesn't affect the task of satisfying Constraint 3. Furthermore, since c_2 's output signal also feeds a primary output, the error input of c_1 doesn't need to be propagated any more and the ATPG procedure is free of trying to assign the aggressor $s_{1,1}$ the desired value 0. Altogether, just by trying c_2 before c_1 , Rule 1 is being applied implicitly.

Hence, if `propagation()` is provided with a mechanism that causes that adequate cells are tried first, Rule 1 is respected. Instead of modifying the classical `propagation()` sub-procedure, it suffices to implement the D-frontier as a priority queue. The priority of a cell c with controlling value v (Recall that `propagation()` will try to assign the value v' to all free inputs of c .) may be defined as $n_+ - n_-$, where n_+ is the number of free inputs of c marked with flag `logic_v_preferred` and n_- is the number of free inputs of c marked with flag `logic_v_preferred`. If `propagation()` tries cells with higher priority first, Rule 1 is respected.

Last but one time frame

Last time frame

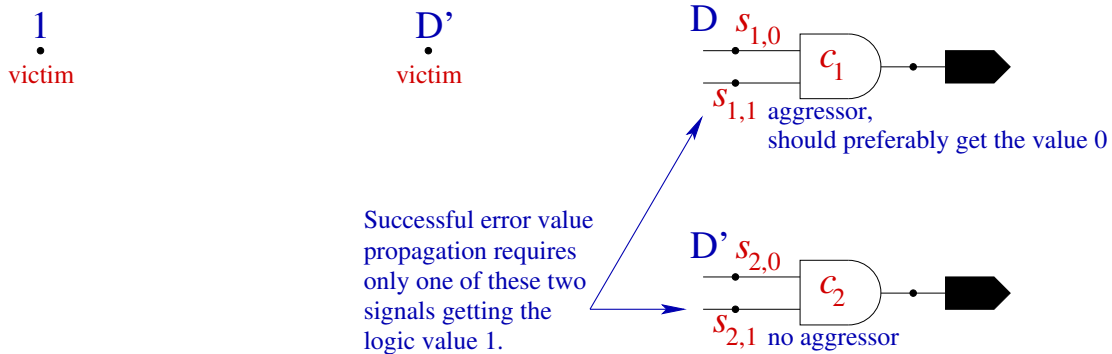


FIGURE 6.7: THE CELL-SELECTING PROCEDURE AND THE APPLICATION OF RULE 1

Note that the application of Rules 2 and 3 is not affected by the cell selection since Rule 2 only applies to the justification of signal values and since error value propagation only takes place in the last time frame, which Rule 3 doesn't apply to.

7

EXPERIMENTAL RESULTS

7.1 The benchmarks

In order to obtain experimental results, the method described in Chapter 5 was applied to ISCAS 85 and ISCAS 89 benchmark circuits. Table 7.1 shows the combinational circuits that were used. The second and third columns contain the number of primary inputs and primary outputs of the circuits, respectively. The fourth column lists the number of signals. In all tables of this chapter the circuits have been ordered by their number of signals which is a direct measure of the test generation's complexity. Finally, the last two columns show the `s_ord` of the signal the test sequence was generated for and the number of branches that signal has got.

Circuit	P_INs	P_OUTs	Signals	Victim	Branches
c0017	5	2	11	2	2
c0095	5	7	32	19	5
c0880	60	26	443	52	8
c1908	33	25	913	300	16
c3540	50	22	1719	226	16
c6288	32	32	2448	7	16
c5315	178	123	2485	1429	13
c7552	207	108	3719	1623	15

TABLE 7.1: COMBINATIONAL ISCAS 85 CIRCUITS

Table 7.2 shows the sequential circuits that were used. As in Table 7.1, the second and third columns contain the number of primary inputs and outputs. Additionally, the fourth column lists the number of secondary inputs of the circuits. The last three columns contain the same data as the last three columns in Table 7.1.

Circuit	P_INs	P_OUTs	S_INs	Signals	Victim	Branches
s00027	4	1	3	17	7	2
s00208	10	1	8	122	13	6
s00298	3	6	14	136	4	12
s00386	7	7	6	172	20	19
s499	1	22	22	175	5	6
s00382	3	6	21	182	15	10
s00344	9	11	15	184	39	8
s00349	9	11	15	185	57	8
s00400	3	6	21	186	15	10
s00444	3	6	21	205	13	10
s00526	3	6	21	218	5	12
s00510	19	7	6	236	24	12
s00420	18	1	16	252	29	6
s00832	18	19	5	310	47	36
s00820	18	19	5	312	20	35
s635	2	1	32	320	8	3
s00641	35	24	19	433	209	9
s00953	16	23	29	440	20	10
s00713	35	23	19	447	208	9
s00838	34	1	32	512	61	6
s938	34	1	32	512	61	6
s01238	14	14	18	540	4	18
s01196	14	14	18	561	2	16
s01494	8	19	6	661	51	54
s01488	8	19	6	667	22	53
s01423	17	5	74	748	554	12
s1512	29	21	57	866	329	25
s3271	26	14	116	1714	276	24
s3384	43	26	183	1911	590	24
s3330	40	73	132	1961	783	19
s4863	49	16	104	2495	4	16
s05378	35	49	179	2993	970	9
s6669	83	55	239	3402	544	34
s09234	36	39	211	5844	780	16

TABLE 7.2: SEQUENTIAL ISCAS 89 CIRCUITS

7.2 Settings

Determining an appropriate victim signal and appropriate aggressors is out of the scope of this work. The victim can be obtained by analysis presented in [1]¹. The aggressors can be determined by the analysis of the power grid: the segment which supplies the victim's driver with current is identified. Aggressors are driven by cells powered by the same segment. M and N should be chosen such as to maximise the voltage drop and can be derived analytically from the electrical parameters of the circuit, the VRM, the capacitor, etc.; or they can be obtained by measurement. Since the used benchmarks contain no layout information or technology data, it was not possible to perform such analysis to obtain the victim node and the aggressors. Since the aim of these experiments was not testing the efficiency of the proposed test method but testing the automatic test pattern generation procedure, it was sufficient to let the algorithm generate only one test sequence per circuit. Thus only one victim was chosen for each circuit. As victim we chose the signal with the largest number of branches as this node is likely to have the largest load. If there were several such signals, one of these was selected at random. We worked with five aggressors which were selected at random either from anywhere in the circuit or from among the victim's neighbours.

7.2.1 Choosing the aggressors

The implemented program can choose 5 aggressors at random from anywhere in the circuit or from among the victim's neighbours.

If the signals chosen as aggressors draw power from sources the victim is not connected to, high-frequency power droop will most probably not take place because of lack of power starvation. That is why, in general, power droop cannot be tested using randomly chosen aggressors. Nevertheless, for the test of the ATPG procedure random aggressors can be seen as optimal because this provides the possibility of testing the procedure under a great variation of the input parameters. Since a signal's `s_ord` lies between 0 and $l - 1$, where l is the number of signals, choosing 5 aggressors at random is reduced to repeatedly generating random integers between 0 and $l - 1$ and putting them into an array of length 5 until the array is full. A new `s_ord` gets into the array only if it is not already there and if it doesn't equal the victim's `s_ord`.

In order to test how well the test sequences generated by the designed ATPG procedure satisfy Constraint 3 (see Page 23), it is helpful to test the ATPG procedure with aggressors chosen from among the victim's neighbours since there is a higher dependence between logic values of neighbouring signals. Furthermore, there is a high probability that aggressors chosen from among the neighbours coincide with the aggressors won by the analysis of the power grid. Since there was no layout information on the circuits an

¹An interesting finding of [1] is that the number of possible sites for power droop is very limited: less than 100 for a microprocessor of 128 000 standard cells.

approximating approach had to be used in order to determine suitable neighbours. If a victim had more than 5 neighbours, 5 were chosen at random to become aggressors.

The set of `direct neighbours` is defined as the set of input signals of the victim's drain cells (with exception of the victim itself). These are adequate neighbours since these signals may be physically close to the victim and because their logic values and the victim's value will be rather independent from each other. However, if all aggressors are direct neighbours at the same time, during the propagation of the victim's error value, many of them might be assigned a value that affects the satisfaction of Constraint 3. In order to dispose of additional differently natured neighbours to choose the aggressors from, we define the set of `indirect neighbours` as the set of output signals of c 's sibling cells (see Page 13 for this term's definition) where c is the cell driving the victim. Finally, the set of suitable neighbours (see Figure 7.1) consists of the set of direct neighbours together with the set of indirect neighbours.

Other signals in the victim's vicinity are not suitable. Choosing signals whose logic values depend on the victim's value or vice versa could affect the task of satisfying Constraint 3.

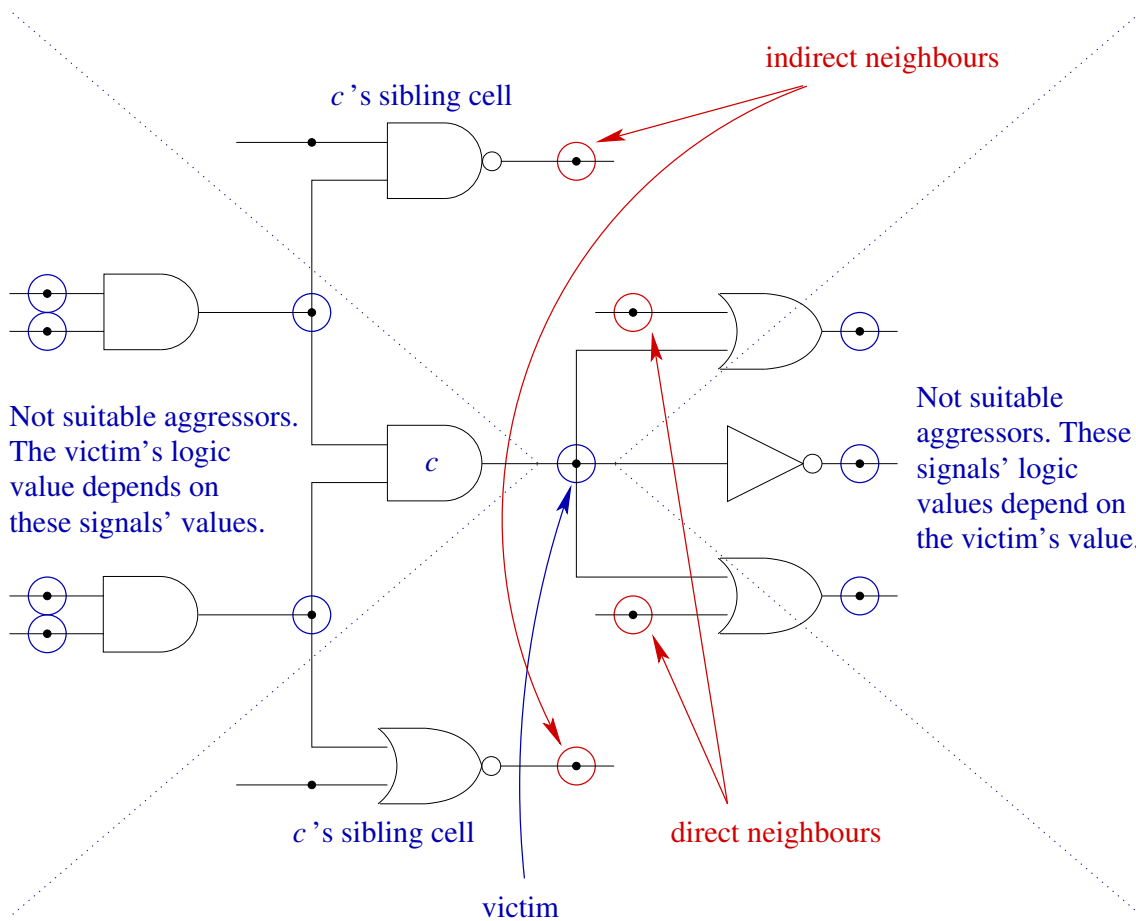


FIGURE 7.1: THE VICTIM'S SUITABLE NEIGHBOURS

7.3 Results

All measurements were performed on a 2×1280 MHz UltraSPARC-IIIi machine with 6 GB RAM running SunOS. The method was applied to all mentioned circuits several times trying different parameter combinations, where the varying parameters are M , N , the tested transition (rising or falling) and whether the aggressors were generated randomly or chosen from among the neighbours. An overview of the shown data follows:

Column Heading “Ag.”: This column is only present if the program was requested to choose the aggressors from among the neighbours. Then it shows how many aggressors were chosen.

Column Heading “Low”: This column shows the percentage of signals that switch in the low-activity part. Let S_i be the number of signals that switch between time frame i and $i + 1$. Then, the average number of signals switching in the low-activity part is

$$S_l := \frac{\sum_{i=0}^{M-2} S_i}{M-1}.$$

The listed number is $\frac{S_l}{l} \cdot 100\%$, where l is the total number of signals in the circuit. Here lesser figures represent better results.

Column Heading “High”: This column shows the percentage of signals that switch in the high-activity part. The average number of signals switching in the high-activity part is

$$S_h := \frac{\sum_{i=M-1}^{M+N-3} S_i}{N-1}.$$

The listed number is $\frac{S_h}{l} \cdot 100\%$. Here larger figures represent better results.

Column Heading “Tr.”: This column shows how many aggressors (max. 5) undergo the same transition as the victim in the last two time frames. Here larger figures represent better results.

Column Heading “Time”: This column shows the time that was needed for test generation on the machine mentioned above.

Sometimes it is not possible to induce the tested transition on any of the neighbours. As was explained in Subsection 5.3.2, in such a case the program quits without trying to generate the low-activity and the high-activity sequences. This means that no test sequence exists for the victim together with the chosen set of aggressors. But there might exist test sequences for a different set of aggressors. In order to gain the experimental results, the program was started, with fixed parameter constellation (i.e. fixed victim,

fixed M and N and fixed decision whether the aggressors are chosen from among the neighbours or from anywhere in the circuit), up to 13 times for each circuit until the test sequence generation was successful. If the program was unsuccessful more than 13 times, no data were recorded and the job was resumed on the next circuit. In such cases the tables show a row of hyphens.

The impact of a $\frac{dI}{dt}$ event is most critical several clock cycles after the $\frac{dI}{dt}$ event has taken place (see Subsection 3.1.1). Therefore, there was a particular interest in finding out how efficient the generation of the low-activity and the high-activity sequence is for large M and N values. We started with $M = N = 10$. Test sequences were generated for both the rising and the falling transitions; and both random and neighbouring aggressors were tried. Detailed results are listed in Tables 7.4 and 7.5, their average figures are summarised in Table 7.3.

For $M = N = 10$, the program was started several times in the random-aggressors-mode. The results from those different starts are not listed, but the obtained figures for one specific circuit were nearly the same in all cases. The figures listed in Tables 7.4 and 7.5 for the random-aggressors-mode are the best results that were achieved for each circuit. Looking at Table 7.3 it is remarkable that the achieved results are nearly the same in both the neighbouring-aggressors and random-aggressors cases. Altogether, we can say that the ATPG procedure seems to be stable with respect to variations of the set of aggressors. Analysing the figures in Tables 7.4 and 7.5 brings forth the following observations:

- Let us first try to evaluate how well the sub-routine that generates the test pair has done its job. The application of the test pair should cause as much aggressors as possible to undergo the same transition as the victim in the last two time frames. There are only three circuits whose chosen victim has less than 5 neighbours. In all the other circuits there are at least five neighbours so we can base our observations on the simplifying assumption that there are always 5 aggressors. We see that the generated test pair has caused, in average, half of the aggressors to undergo the desired transition. For several circuits the transition was induced on all aggressors. It can also be observed that the results are slightly better if the aggressors are chosen at random. This was more or less expected since many neighbours (especially the direct ones, cf. Subsection 7.2.1) are not controllable any more after having propagated the victim's error and after having justified the victim's transition.

Settings	Low	High	Tr.	Time
rising transition, neighbour aggressors	7.92%	41.64%	2.28	0:03:43
rising transition, random aggressors	6.44%	42.15%	2.37	0:03:53
falling transition, neighbour aggressors	6.88%	40.19%	2.19	0:01:41
falling transition, random aggressors	8.32%	42.63%	2.32	0:03:17

TABLE 7.3: EXPERIMENTAL RESULTS: $M = N = 10$, AVERAGE FIGURES

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	2	0%	87%	2	0:00:00	0%	87%	3	0:00:00
c0095	-	-	-	-	-	0%	71%	3	0:00:00
c0880	5	0%	51%	5	0:00:02	0%	50%	3	0:00:01
c1908	5	0%	59%	4	0:00:02	0%	57%	4	0:00:01
c3540	2	0%	42%	1	0:00:08	0%	46%	3	0:00:05
c6288	5	0%	43%	5	0:00:05	0%	47%	3	0:00:39
c5315	5	0%	54%	3	0:00:14	0%	53%	4	0:00:14
c7552	5	0%	52%	4	0:00:42	0%	52%	5	0:00:42
s00027	-	-	-	-	-	11%	69%	2	0:00:00
s00208	-	-	-	-	-	0%	36%	1	0:00:00
s00298	-	-	-	-	-	10%	50%	1	0:00:01
s00386	5	0%	37%	2	0:00:02	4%	30%	2	0:00:02
s499	5	4%	18%	1	0:00:02	-	-	-	-
s00382	5	12%	42%	1	0:00:03	4%	32%	2	0:00:02
s00344	5	14%	49%	5	0:00:01	16%	47%	3	0:00:02
s00349	5	17%	49%	1	0:00:02	3%	40%	1	0:00:01
s00400	5	6%	39%	1	0:00:04	5%	37%	2	0:00:02
s00444	5	11%	43%	1	0:00:09	0%	39%	1	0:00:07
s00526	5	13%	29%	2	0:00:02	3%	24%	1	0:00:01
s00510	5	14%	37%	1	0:00:04	0%	29%	1	0:00:04
s00420	5	3%	31%	1	0:00:03	3%	26%	2	0:00:02
s00832	5	3%	39%	3	0:00:09	0%	37%	3	0:00:04
s00820	5	5%	40%	1	0:00:03	0%	43%	2	0:00:01
s635	2	0%	12%	1	0:00:02	0%	13%	1	0:00:02
s00641	5	2%	49%	3	0:00:27	0%	50%	4	0:00:06
s00953	5	0%	21%	1	0:04:30	1%	19%	1	0:05:06
s00713	5	2%	45%	2	0:00:18	2%	49%	2	0:00:15
s00838	5	1%	30%	1	0:00:07	0%	27%	3	0:00:04
s938	5	3%	30%	1	0:00:10	0%	28%	1	0:00:05
s01238	5	10%	34%	2	0:00:18	13%	36%	1	0:00:23
s01196	5	6%	34%	1	0:00:17	18%	38%	3	0:00:21
s01494	5	23%	33%	2	0:00:40	29%	34%	1	0:00:32
s01488	5	3%	35%	1	0:00:23	9%	33%	1	0:00:56
s01423	5	15%	32%	1	0:00:51	5%	26%	1	0:00:46
s1512	-	-	-	-	-	0%	29%	1	0:00:15
s3271	5	29%	57%	2	0:01:01	32%	55%	5	0:00:55
s3384	5	23%	59%	5	0:02:22	23%	58%	4	0:01:39
s3330	5	11%	61%	4	0:00:57	9%	52%	3	0:01:06
s4863	5	20%	41%	5	0:23:31	24%	40%	3	1:34:17
s05378	-	-	-	-	-	6%	53%	2	0:08:38
s6669	5	23%	46%	4	0:13:03	18%	45%	4	0:06:25
s09234	5	12%	39%	2	1:23:11	16%	41%	4	0:35:08
AVG:	4.75	7.92%	41.64%	2.28	0:03:43	6.44%	42.15%	2.37	0:03:53

TABLE 7.4: EXPERIMENTAL RESULTS: $M = N = 10$, RISING TRANSITION

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	-	-	-	-	-	-	-	-	-
c0095	-	-	-	-	-	0%	65%	1	0:00:00
c0880	-	-	-	-	-	0%	56%	1	0:00:01
c1908	5	0%	59%	2	0:00:02	0%	61%	4	0:00:02
c3540	2	0%	40%	1	0:00:04	0%	38%	2	0:00:04
c6288	-	-	-	-	-	0%	44%	2	0:00:08
c5315	5	0%	54%	1	0:00:11	0%	53%	3	0:00:10
c7552	5	0%	53%	2	0:00:24	0%	52%	5	0:00:25
s00027	2	9%	73%	1	0:00:00	1%	73%	3	0:00:00
s00208	5	0%	38%	3	0:00:00	5%	38%	1	0:00:01
s00298	5	10%	35%	2	0:00:01	12%	43%	2	0:00:00
s00386	5	15%	41%	2	0:00:01	-	-	-	-
s499	5	1%	22%	2	0:00:03	2%	29%	1	0:00:05
s00382	5	6%	42%	2	0:00:03	10%	33%	2	0:00:03
s00344	5	11%	39%	1	0:00:01	0%	45%	1	0:00:01
s00349	5	5%	42%	1	0:00:01	7%	42%	1	0:00:01
s00400	5	4%	38%	2	0:00:02	10%	42%	3	0:00:02
s00444	5	4%	42%	3	0:00:03	9%	36%	1	0:00:04
s00526	5	5%	33%	3	0:00:02	3%	25%	2	0:00:02
s00510	5	11%	34%	1	0:00:04	18%	30%	1	0:00:06
s00420	5	3%	33%	4	0:00:02	2%	29%	4	0:00:02
s00832	5	4%	35%	1	0:00:08	11%	43%	1	0:00:04
s00820	5	0%	39%	2	0:00:04	20%	40%	1	0:00:12
s635	-	-	-	-	-	-	-	-	-
s00641	-	-	-	-	-	4%	47%	1	0:00:34
s00953	5	1%	25%	1	0:05:54	2%	30%	1	0:05:11
s00713	-	-	-	-	-	4%	45%	1	0:00:17
s00838	5	1%	26%	4	0:00:06	0%	28%	2	0:00:06
s938	5	1%	28%	4	0:00:06	1%	28%	1	0:00:05
s01238	5	18%	37%	4	0:00:22	17%	35%	3	0:00:26
s01196	5	15%	40%	3	0:00:19	15%	33%	2	0:00:21
s01494	5	15%	36%	2	0:00:43	20%	42%	1	0:00:24
s01488	5	17%	38%	1	0:00:32	11%	43%	2	0:00:31
s01423	5	3%	34%	2	0:00:48	8%	30%	4	0:00:54
s1512	5	2%	31%	1	0:00:16	-	-	-	-
s3271	5	30%	56%	4	0:00:53	33%	58%	4	0:00:47
s3384	-	-	-	-	-	25%	58%	4	0:02:13
s3330	5	10%	51%	2	0:01:18	11%	55%	4	0:01:28
s4863	-	-	-	-	-	12%	35%	4	1:53:41
s05378	5	9%	52%	5	0:10:47	7%	51%	4	0:10:23
s6669	-	-	-	-	-	22%	44%	5	0:07:42
s09234	5	10%	40%	1	0:30:18	14%	41%	3	0:41:40
AVG:	4.81	6.88%	40.19%	2.19	0:01:41	8.32%	42.63%	2.32	0:03:17

TABLE 7.5: EXPERIMENTAL RESULTS: $M = N = 10$, FALLING TRANSITION

- The switching activity in the low-activity part is satisfyingly low. 0% of a combinational circuit's signals switching in the low-activity part is an expected result. But there are also some sequential circuits for which the same result has been achieved. There are only very few cases in which the percentage of switching signals is greater than 10%. Nevertheless, the average percentage of switching signals lies between 6% and 8% which is very good considering that the ATPG procedure tries to assign as much signals as possible a specified logic value.
- Before evaluating the switching activity in the high-activity sequence, recall that a global switching activity of 100% cannot be reached in general (see Section 3.2). Table 7.6, in which ARS stands for "Average Random Switching", shows an estimate of the average switching activity in the circuits under normal working conditions. These data were gained by applying 5000 random input patterns to the circuits and measuring the amount of signals switching between each pair of cycles. The values don't vary much between different circuits, the average value is of about 30%. After comparing this value to the achieved high-activity average values of about 41% (note that there are several values greater than 50% and even 60%) it can be said that the high-activity sequence has managed to increase the average switching activity by about 33%. It is not feasible to deterministically find out the highest reachable switching activity. However, an overall high switching average of more than 50% or 60% may be considered impossible to achieve if taking into account the following facts:
 - ◊ The algorithm tries to leave as less signals as possible having an unspecified logic value.
 - ◊ These circuits contain many complex cells (AND, OR, NAND, NOR gates with more than five and even up to nine inputs) which means that there is a high local dependence between signal values.
 - ◊ Many of the used sequential circuits have got many more secondary than primary inputs. Hence, there is a high degree of reconvergence in these circuits.
 - ◊ Only the primary inputs are controllable for so many subsequent cycles.
- Finally, let us analyse the needed times. Average times of about three minutes are reasonable. It is interesting that the needed time can vary acutely for one circuit (e.g. s4863, s09234). The hardness of the problem seems to depend on the specific problem instance (controllability and observability of aggressors and victim, etc.) rather than simply on M and N . This dependency seems to show up quite randomly. Sometimes, the justification algorithm in charge of performing the assignments intended for the satisfaction of the flexible constraints, may become very deep before realising that the assignment leads to a conflict. This behaviour is neither perfectly predictable nor controllable, it is unavoidable given the heuristic procedure that is used for the generation of the low-activity and high-activity sequences.

Circuit	ARS	Circuit	ARS	Circuit	ARS	Circuit	ARS
c0017	46.72%	s00386	35.58%	s00820	30.10%	s01423	32.26%
c0095	39.33%	s499	27.01%	s635	10.08%	s1512	24.07%
c0880	34.09%	s00382	34.31%	s00641	30.66%	s3271	49.18%
c1908	39.73%	s00344	34.97%	s00953	17.08%	s3384	45.72%
c3540	33.07%	s00349	34.87%	s00713	29.49%	s3330	36.33%
c6288	38.01%	s00400	33.68%	s00838	10.25%	s4863	42.45%
c5315	39.91%	s00444	31.45%	s938	10.23%	s05378	35.00%
c7552	40.86%	s00526	28.35%	s01238	25.65%	s6669	38.39%
s00027	37.19%	s00510	23.98%	s01196	26.77%	s09234	28.67%
s00208	19.25%	s00420	13.20%	s01494	30.77%	AVG:	31.33%
s00298	36.43%	s00832	29.62%	s01488	30.94%		

TABLE 7.6: AVERAGE SWITCHING ACTIVITY, 5000 CYCLES, RANDOM INPUT PATTERNS

After having applied the method to the benchmarks for $M = N = 10$, we tried $M = N = 30$ (Table 7.8), $M = N = 50$ (Table 7.9), $M = N = 100$ (Table 7.10) and $M = N = 150$ (Table 7.11). We generated test sequences only for the rising transition (falling transition in the case $M = N = 150$) since we already found out that the results don't vary much when switching from rising to falling transitions. Also, after realising that the results we are interested in aren't affected too much by the choice of the aggressors, it wasn't necessary to start the program more than once in the random-aggressors-mode for each circuit and for each parameter combination.

M	N	Aggressors	Low	High	Tr.	Time
10	10	neighbours	7.92%	41.64%	2.28	0:03:43
		random	6.44%	42.15%	2.37	0:03:53
30	30	neighbours	5.81%	41.06%	2.25	0:12:27
		random	5.6%	43%	2.38	0:13:11
50	50	neighbours	5.91%	40.97%	2.16	0:11:33
		random	4.43%	43.24%	2.27	0:10:26
100	100	neighbours	4.72%	42.91%	2.06	0:05:47
		random	4.56%	43.03%	2.28	0:17:13
150	150	neighbours	5.06%	42.44%	2.06	0:15:25
		random	5.61%	41.79%	2.32	0:22:08

TABLE 7.7: EXPERIMENTAL RESULTS: AVERAGE FIGURES

Table 7.7 summarises the average results for different M and N values, Figure 7.2 contains the same data again, but normalised to the average data for $M = N = 10$ and rising transition and in graph form. As was already said before, we had particular interest in finding out how efficient the generation of the low-activity and the high-activity sequence is for large M and N values. According to Prof. Sandip Kundu it may be necessary, in many cases, to have hundreds of cycles of electrical noise before low-frequency power droop can develop fully. The average figures largely remain as good as they were for $M = N = 10$. The switching activity in the low-activity part even appears to become better (it decreases) with increasing M and N . The time requirements are higher but, if we compare the average time needed for $M = N = 10$ with the average time needed for $M = N = 150$, we see that the time has increased by a factor of only 6 or 7. The time increase seems to be sublinear in M and N . The solution quality seems to be relatively stable with respect to M and N and some of the statistical noise is attributed to the random sets of aggressors being not the same throughout the experiments. Since the algorithm does never work on more than two frames at a time, if memory freeing is carefully implemented, memory requirements only increase if the circuit's size does so, not if M and N become larger.

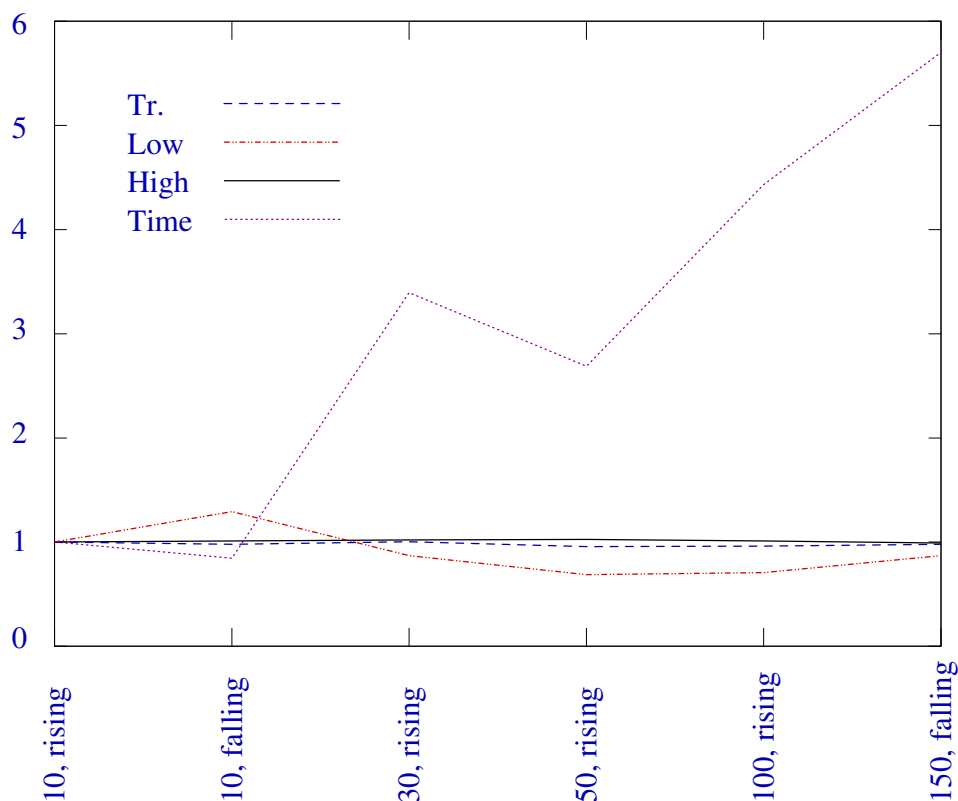


FIGURE 7.2: EXPERIMENTAL RESULTS: NORMALISED AVERAGE RESULTS

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	2	0%	90%	2	0:00:00	0%	90%	3	0:00:00
c0095	-	-	-	-	-	0%	71%	1	0:00:00
c0880	5	0%	42%	5	0:00:02	0%	46%	4	0:00:02
c1908	5	0%	58%	3	0:00:03	0%	63%	3	0:00:03
c3540	2	0%	43%	1	0:00:09	0%	42%	3	0:00:23
c6288	5	0%	45%	5	0:00:12	0%	41%	1	0:01:07
c5315	5	0%	54%	4	0:00:32	0%	55%	3	0:00:32
c7552	5	0%	49%	3	0:01:13	0%	50%	3	0:01:14
s00027	-	-	-	-	-	3%	68%	2	0:00:00
s00208	-	-	-	-	-	1%	38%	2	0:00:01
s00298	-	-	-	-	-	6%	47%	1	0:00:02
s00386	5	1%	34%	2	0:00:06	16%	34%	1	0:00:07
s499	5	0%	20%	1	0:00:06	-	-	-	-
s00382	5	7%	30%	1	0:00:10	4%	27%	1	0:00:09
s00344	5	12%	43%	5	0:00:05	4%	43%	2	0:00:05
s00349	5	3%	44%	1	0:00:03	0%	28%	1	0:00:02
s00400	5	4%	33%	1	0:00:08	7%	34%	2	0:00:09
s00444	-	-	-	-	-	4%	36%	4	0:00:14
s00526	5	3%	29%	1	0:00:05	5%	32%	1	0:00:06
s00510	-	-	-	-	-	4%	35%	2	0:00:08
s00420	5	0%	27%	1	0:00:03	1%	32%	2	0:00:04
s00832	5	0%	37%	1	0:00:17	0%	41%	2	0:00:28
s00820	5	4%	40%	2	0:00:13	0%	41%	1	0:00:10
s635	2	0%	17%	1	0:00:06	-	-	-	-
s00641	5	0%	49%	2	0:00:23	0%	50%	1	0:01:30
s00953	5	1%	29%	1	0:15:34	0%	24%	2	0:00:59
s00713	5	1%	48%	1	0:00:22	0%	49%	5	0:00:17
s00838	5	0%	25%	1	0:00:11	0%	26%	1	0:00:12
s938	5	0%	25%	1	0:00:13	0%	26%	3	0:00:12
s01238	5	13%	36%	2	0:01:22	11%	31%	3	0:01:37
s01196	5	11%	39%	2	0:00:58	13%	37%	2	0:01:14
s01494	5	23%	37%	1	0:01:47	20%	33%	2	0:02:29
s01488	5	12%	38%	1	0:02:08	17%	39%	1	0:01:46
s01423	-	-	-	-	-	11%	34%	1	0:02:16
s1512	-	-	-	-	-	2%	31%	2	0:00:56
s3271	5	31%	57%	5	0:02:45	30%	58%	4	0:02:27
s3384	5	22%	58%	5	0:06:20	23%	60%	5	0:07:08
s3330	5	6%	52%	4	0:04:31	12%	58%	5	0:03:24
s4863	-	-	-	-	-	7%	41%	3	2:04:33
s05378	-	-	-	-	-	3%	50%	5	0:30:36
s6669	5	20%	47%	5	0:41:29	7%	38%	4	3:25:10
s09234	5	12%	39%	1	5:16:51	13%	41%	1	2:15:13
AVG:	4.72	5.81%	41.06%	2.25	0:12:27	5.6%	43%	2.38	0:13:11

TABLE 7.8: EXPERIMENTAL RESULTS: $M = N = 30$, RISING TRANSITION

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	2	0%	72%	2	0:00:00	0%	90%	2	0:00:00
c0095	-	-	-	-	-	0%	71%	1	0:00:00
c0880	5	0%	42%	5	0:00:04	0%	46%	3	0:00:04
c1908	5	0%	61%	2	0:00:05	0%	60%	4	0:00:05
c3540	2	0%	43%	1	0:00:18	0%	41%	3	0:00:17
c6288	5	0%	46%	5	0:00:17	0%	41%	2	0:00:27
c5315	5	0%	54%	3	0:00:50	0%	52%	3	0:00:50
c7552	5	0%	49%	3	0:01:46	0%	53%	5	0:01:45
s00027	-	-	-	-	-	0%	69%	1	0:00:00
s00208	-	-	-	-	-	0%	35%	1	0:00:01
s00298	-	-	-	-	-	-	-	-	-
s00386	5	5%	29%	2	0:00:07	0%	28%	1	0:00:12
s499	-	-	-	-	-	-	-	-	-
s00382	5	8%	33%	1	0:00:12	4%	39%	3	0:00:13
s00344	5	0%	46%	5	0:00:04	4%	46%	4	0:00:06
s00349	5	10%	47%	1	0:00:09	0%	45%	2	0:00:06
s00400	5	7%	29%	1	0:00:16	7%	29%	1	0:00:14
s00444	-	-	-	-	-	2%	35%	2	0:00:28
s00526	5	6%	31%	1	0:00:09	3%	31%	1	0:00:08
s00510	5	5%	37%	1	0:00:23	1%	35%	2	0:00:19
s00420	5	0%	29%	1	0:00:06	0%	27%	3	0:00:07
s00832	5	1%	42%	3	0:00:09	9%	46%	1	0:00:17
s00820	5	3%	43%	1	0:00:14	0%	45%	2	0:00:06
s635	2	0%	21%	1	0:00:14	2%	19%	2	0:00:18
s00641	5	0%	49%	2	0:00:32	0%	48%	1	0:00:29
s00953	5	1%	25%	1	0:28:36	-	-	-	-
s00713	5	0%	50%	3	0:00:30	0%	52%	2	0:00:37
s00838	5	0%	25%	1	0:00:18	0%	26%	2	0:00:20
s938	5	0%	27%	1	0:00:21	0%	26%	2	0:00:19
s01238	5	16%	35%	2	0:01:46	15%	35%	2	0:02:08
s01196	5	16%	36%	1	0:02:12	12%	40%	3	0:01:41
s01494	5	18%	39%	3	0:02:49	16%	40%	1	0:02:26
s01488	5	11%	35%	1	0:04:32	8%	37%	2	0:04:36
s01423	5	11%	29%	2	0:05:16	0%	33%	2	0:02:43
s1512	-	-	-	-	-	0%	27%	1	0:01:20
s3271	5	31%	59%	3	0:04:18	31%	58%	2	0:04:33
s3384	5	21%	57%	5	0:13:19	24%	57%	3	0:12:46
s3330	5	5%	50%	4	0:05:19	9%	49%	5	0:05:59
s4863	-	-	-	-	-	-	-	-	-
s05378	-	-	-	-	-	4%	49%	4	1:48:33
s6669	-	-	-	-	-	-	-	-	-
s09234	5	14%	41%	1	4:54:10	13%	40%	3	3:51:18
AVG:	4.72	5.91%	40.97%	2.16	0:11:33	4.43%	43.24%	2.27	0:10:26

TABLE 7.9: EXPERIMENTAL RESULTS: $M = N = 50$, RISING TRANSITION

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	2	0%	81%	2	0:00:00	0%	72%	2	0:00:00
c0095	-	-	-	-	-	0%	71%	1	0:00:00
c0880	5	0%	58%	5	0:00:08	0%	41%	2	0:00:07
c1908	5	0%	59%	3	0:00:09	0%	61%	4	0:00:09
c3540	2	0%	41%	1	0:00:37	0%	46%	1	0:00:32
c6288	5	0%	43%	5	0:00:33	0%	46%	3	0:00:40
c5315	5	0%	51%	3	0:01:35	0%	51%	5	0:01:36
c7552	5	0%	51%	3	0:03:04	0%	51%	5	0:03:06
s00027	-	-	-	-	-	1%	69%	1	0:00:00
s00208	-	-	-	-	-	-	-	-	-
s00298	5	5%	40%	1	0:00:06	5%	42%	3	0:00:07
s00386	5	0%	32%	3	0:00:11	-	-	-	-
s499	-	-	-	-	-	-	-	-	-
s00382	5	4%	30%	1	0:00:25	6%	28%	2	0:00:29
s00344	5	0%	45%	5	0:00:12	0%	47%	3	0:00:09
s00349	5	3%	47%	1	0:00:13	1%	43%	2	0:00:10
s00400	5	6%	28%	1	0:00:34	4%	29%	1	0:00:28
s00444	5	2%	38%	1	0:01:02	2%	38%	2	0:01:22
s00526	-	-	-	-	-	6%	31%	1	0:00:19
s00510	5	2%	36%	1	0:00:27	1%	34%	1	0:00:32
s00420	5	0%	29%	1	0:00:11	0%	28%	1	0:00:11
s00832	5	0%	40%	2	0:00:30	1%	41%	3	0:00:34
s00820	5	0%	44%	2	0:00:23	0%	43%	1	0:00:17
s635	2	1%	20%	1	0:00:30	0%	20%	1	0:00:26
s00641	5	0%	55%	2	0:00:54	0%	48%	2	0:01:01
s00953	-	-	-	-	-	1%	22%	1	1:11:26
s00713	5	0%	50%	2	0:01:18	0%	48%	1	0:01:19
s00838	5	0%	26%	1	0:00:40	0%	25%	2	0:00:37
s938	5	0%	26%	1	0:00:37	0%	25%	1	0:00:38
s01238	5	15%	39%	1	0:03:27	13%	38%	2	0:03:24
s01196	5	15%	37%	2	0:03:20	16%	37%	2	0:04:01
s01494	5	2%	37%	1	0:03:20	12%	40%	1	0:03:54
s01488	5	13%	43%	1	0:06:14	16%	39%	3	0:05:51
s01423	5	8%	32%	1	0:06:29	0%	33%	2	0:05:16
s1512	-	-	-	-	-	0%	29%	1	0:02:45
s3271	5	33%	59%	1	0:09:24	33%	60%	3	0:09:47
s3384	5	22%	58%	4	0:28:45	21%	58%	4	0:27:51
s3330	5	6%	50%	3	0:10:44	8%	56%	3	0:12:56
s4863	-	-	-	-	-	7%	41%	3	2:00:00
s05378	-	-	-	-	-	1%	56%	5	1:30:57
s6669	5	14%	48%	4	1:38:54	14%	49%	5	2:20:56
s09234	-	-	-	-	-	9%	42%	3	2:37:47
AVG:	4.72	4.72%	42.91%	2.06	0:05:47	4.56%	43.03%	2.28	0:17:13

TABLE 7.10: EXPERIMENTAL RESULTS: $M = N = 100$, RISING TRANSITION

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
c0017	-	-	-	-	-	-	-	-	-
c0095	5	0%	71%	1	0:00:01	0%	71%	1	0:00:00
c0880	-	-	-	-	-	0%	48%	2	0:00:11
c1908	5	0%	59%	1	0:00:14	0%	56%	3	0:00:14
c3540	2	0%	44%	1	0:00:49	0%	39%	3	0:00:50
c6288	-	-	-	-	-	0%	48%	1	0:01:34
c5315	5	0%	53%	2	0:02:17	0%	52%	5	0:02:16
c7552	5	0%	54%	2	0:04:05	0%	53%	4	0:04:05
s00027	2	0%	70%	1	0:00:00	0%	70%	4	0:00:00
s00208	5	0%	35%	3	0:00:05	0%	37%	3	0:00:04
s00298	5	6%	40%	3	0:00:10	7%	40%	1	0:00:10
s00386	5	0%	39%	1	0:00:11	-	-	-	-
s499	5	0%	27%	3	0:00:43	0%	25%	2	0:00:40
s00382	5	4%	34%	2	0:00:37	4%	27%	1	0:00:43
s00344	5	0%	46%	1	0:00:15	0%	45%	3	0:00:16
s00349	5	0%	41%	1	0:00:14	9%	45%	3	0:00:21
s00400	5	5%	34%	2	0:00:37	8%	30%	3	0:00:58
s00444	-	-	-	-	-	2%	39%	1	0:01:19
s00526	5	5%	30%	4	0:00:29	6%	31%	2	0:00:26
s00510	5	3%	36%	2	0:00:50	-	-	-	-
s00420	5	0%	29%	1	0:00:16	0%	30%	1	0:00:15
s00832	5	1%	39%	2	0:01:22	0%	42%	3	0:00:28
s00820	5	0%	40%	1	0:01:32	0%	41%	1	0:00:49
s635	-	-	-	-	-	0%	15%	1	0:00:28
s00641	5	0%	49%	1	0:01:32	-	-	-	-
s00953	5	3%	22%	1	0:35:21	1%	30%	1	1:37:22
s00713	5	0%	46%	1	0:01:17	0%	45%	1	0:02:12
s00838	5	0%	26%	4	0:00:58	0%	26%	1	0:00:54
s938	5	0%	26%	4	0:00:59	0%	25%	1	0:00:57
s01238	5	16%	35%	3	0:06:39	15%	38%	2	0:06:29
s01196	5	16%	39%	4	0:06:20	12%	39%	2	0:05:30
s01494	5	17%	42%	1	0:05:52	20%	41%	1	0:07:29
s01488	5	13%	43%	2	0:08:32	15%	39%	1	0:10:37
s01423	5	2%	32%	1	0:10:37	8%	31%	3	0:08:59
s1512	-	-	-	-	-	0%	28%	1	0:03:55
s3271	5	32%	58%	4	0:14:28	33%	59%	4	0:15:35
s3384	5	20%	59%	1	0:37:19	21%	60%	3	0:36:25
s3330	5	6%	48%	3	0:24:21	10%	59%	5	0:17:28
s4863	-	-	-	-	-	7%	43%	4	1:47:47
s05378	5	5%	52%	5	3:09:31	3%	51%	4	2:34:22
s6669	5	18%	45%	1	2:45:29	22%	48%	4	2:47:12
s09234	-	-	-	-	-	10%	42%	2	3:01:47
AVG:	4.82	5.06%	42.44%	2.06	0:15:25	5.61%	41.79%	2.32	0:22:08

TABLE 7.11: EXPERIMENTAL RESULTS: $M = N = 150$, FALLING TRANSITION

7.4 Larger sequential circuits

We also worked on a few other larger ISCAS 89 sequential circuits which haven't been mentioned before in this chapter. Their data are displayed in Table 7.12. This table's last column shows the average switching activity under normal working conditions. The figures were gained by applying 5000 random input patterns. Due to the high amount of memory the program required when applied to these sequential circuits, the measurements for these circuits were performed on a 2600 MHz AMD Opteron machine with 16 GB RAM running Debian Linux. The results obtained for $M = N = 10$ are depicted in Table 7.13.

Unfortunately, the shown execution times are not necessarily correct since the mentioned machine was used for other measurements simultaneously which may have lead to inaccurate time measurements. The switching activity in the high-activity part is as good as it was for the other circuits, the number of aggressors undergoing the same transition as the victim in the last two time frames even better. The average switching activity in the low-activity part is slightly higher than it was for the smaller circuits. However, it is only the results for s35932 that push up the average value this badly (s35932 possesses the highest amount of S_INs and only very few P_INs.) The results obtained for the other circuits are in the same range as the results obtained for the small circuits.

Circuit	P_INs	P_OUTs	S_INs	Signals	Victim	Br.	Avg Rnd Swt
s13207	62	152	638	8651	5989	30	26.72%
s15850	77	150	534	10383	3513	33	25.48%
s35932	35	320	1728	17828	5095	128	42.23%
s38584	38	304	1426	20717	12356	88	32.34%
s38417	28	106	1636	23843	1726	49	25.89%
							AVG: 30.53%

TABLE 7.12: LARGER SEQUENTIAL ISCAS 89 CIRCUITS

Circuit	Aggressors: up to 5 neighbours					Aggressors: 5 at random			
	Ag.	Low	High	Tr.	Time	Low	High	Tr.	Time
s13207	5	11%	43%	3	1:11:36	9%	42%	3	2:18:01
s15850	5	9%	35%	5	7:21:58	10%	34%	3	1:28:18
s35932	5	25%	48%	3	18:34:25	25%	48%	3	20:02:28
s38584	-	-	-	-	-	19%	46%	2	6:01:27
s38417	5	13%	31%	2	2:31:34	12%	32%	5	2:01:55
AVG:	5	14.5%	39.25%	3.25	7:24:53	15%	40.4%	3.2	6:22:25

TABLE 7.13: EXPERIMENTAL RESULTS: LARGER SEQUENTIAL CIRCUITS, $M = N = 10$, RISING TRANSITION

8

CONCLUSIONS

Power droop is a power integrity phenomenon which arises under specific conditions and leads to power starvation and erroneous circuit operation. If power droop is not targeted directly, its effects could be confused with that of random noise, leading to potentially counterproductive mitigation strategies. In contrast, if power droop is identified as such, the parts of the design which need improvements are determined easily.

We presented a test method consisting of the application of a long sequence of input patterns which creates worst-case power starvation by maximising the effects of low-frequency and high-frequency power droop. Furthermore, we presented an automatic test pattern generation method which has to apply sequential test generation even for circuits with scan. The classical D-algorithm is enhanced by a dynamic constraint generation technique in order to produce the test sequence while satisfying non-trivial constraints for power droop maximisation. A prototype was implemented using C++ and the circuit representation library `test_circ`. Basic implementation issues have been introduced and explained in this writing, while all the sources are delivered on a CD-ROM. Finally, experiments were executed on ISCAS 85 and ISCAS 89 circuits. The results have been presented and their quality has been discussed extensively.

While the proposed implementation is adequate for mid-size blocks and clearly demonstrates the feasibility of the approach in this context, scalability may be limited for larger devices in combination with longer test sequences. Possible solutions include the use of a basic algorithm more appropriate for sequential test generation such as PODEM, in connection with advanced techniques such as static and dynamic learning.

Appendix A

CONTENTS OF THE ATTACHED CD-ROM

Directory tree:

studienarbeit

```
|-- bin
|-- etc
|-- include
|-- lib
|-- share
    |-- Benchmarks
    |-- experiments
        |-- program_outputs
    |-- number_of_signals
    |-- report
        |-- RawData
            |-- graphics
                |-- RawData
    |-- signals_and_branches
    |-- test_inputs
    |-- test_logs
|-- src
    |-- apps
    |-- libczu
    |-- tests
```

Contents of directory:

Makefile and executable files
configuration files needed for compilation
header files and documentation on
 implemented classes
compiled library and object files
miscellaneous files like lists of
 benchmark circuits, chosen victims
 and official proposal for Studienarbeit
benchmark files
experimental results in tabular form
generated test sequences and process logs
lists of the number of signals of
 each circuit
written work
pdfTeX sources
included pdf-graphics
fig and eps sources
lists of signals and their number of
 branches
inputs for test routines for libraries
outputs of test routines for libraries
source files
ATPG procedure and help routines
libraries
test routines for libraries

BIBLIOGRAPHY

- [1] *C. Tirumurti, S. Kundu, S. Sur-Kolay, Y.S. Chang.* A MODELING APPROACH FOR ADDRESSING POWER SUPPLY SWITCHING NOISE RELATED FAILURES OF INTEGRATED CIRCUITS. Design, Automation and Test in Europe, 2004.
- [2] *M. Abramovici, M.A. Breuer, A.D. Friedman.* DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN. Computer Science Press, 1990.
- [3] *J.P. Roth.* DIAGNOSIS OF AUTOMATA FAILURES: A CALCULUS AND A METHOD. IBM J. Res. Dev., 10:278–281, 1966.
- [4] *N. Jha and S. Gupta.* TESTING OF DIGITAL SYSTEMS. Cambridge University Press, 2003.
- [5] *S. Kundu.* TESTING FOR CIRCUIT MARGINALITIES. Int'l Test Synthesis Workshop, 2004.