Design Strategy for Testing Sequential Logic Gates Based on Signal Pattern Generator

Pablo Rafael Bodmann, Renato Perez Ribas

Abstract—**The validation of standard cell libraries used on digital integrated circuit design is a crucial task. However, the test of sequential logic gates is quite complex due to the inherent memory effect. In this work, it is proposed a universal signal pattern generator to be applied in a novel test circuit strategy. To model the problem our approach creates a pattern sequence over structures like graph and tree structures. As proof-of-concept, a Java application has been developed. Experimental results have shown that such a strategy has attained a high coverage of logic faults.**

Index Terms—**digital circuit, standard cell library, logic gate, sequential cell, test.**

I. Introduction

THE design of integrated circuits (IC) comprises many tasks. In order to reduce design costs and time-to-market, standard cells design methodology has been widely adopted. This methodology is based on the reuse of blocks and small circuits (named as cells) that implement logic functions. These pre-designed cells are available in a library and must be pre-evaluated and pre-validated before using in ASIC design.

Since a library usually comprises a large number of logic gates and all of them must be validated, efficient test setups are essential to reduce design costs. For combinational gates, we consider that the solution proposed in [1] is quite simple and effective. However, the test of sequential logic gates is more complex than the combinational ones due to the inherent memory behavior, *i.e.*, the current output signals depend not only on the current input variables but also on the previous sequence of these ones. Another difficulty is the presence of asynchronous signals that have priority in the sequential behavior.

Several attempts have been proposed related to the testing of sequential logic gates. In [2], the authors propose the use of Boolean equation describing the gate and create a corresponding state table in order to calculate the minimum input sequence to cover all possible defects usually observed in the registers. Despite of having high coverage, this solution is not general, *i.e.*, the sequence depends on the specific circuit behavior targeted. This is a huge problem when testing a large set of gates with different behaviors because for each group of gates a single generator must be created so increasing the complexity of the test. Another approach evaluates latches and flip-flops using shift-register and counter circuitries, respectively [3]. However, it is not shown the way to create a new shift-register for testing other sequential logic gates different from the ones treated in that work. Therefore, this solution is not general yet. An approach based on finite state machine, for describing the sequential gate behavior, is found in [4]. However, as this approach also represents particular solutions for specific sequential gates, a large circuit area overhead is expected.

Pablo Rafael Bodmann and Renato Perez Ribas are with the Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), 9500 Porto Alegre, Brazil (e-mail: {prbodmann, rpribas}@inf.ufrgs.br).

Research partially funded by CNPq Brazilian agency

In this work, we proposed a universal logic vector generator for testing sequential circuits. The main idea is that the generator provides one signal transition per cycle, covering all possible steady states and signal transitions. Repeated output signal transitions do not occur.

This article is organized as follows. Section II discusses the possible static states, expected and unexpected transitions of sequential logic gates. Section III analyzes some previous approaches related to the testing of sequential cells. Section IV presents the proposed approach. Section V shows the circuit generator implementation. Section VI shows and discusses some experimental results. The conclusions are outlined in Section VII.

II. Preliminaries

This section presents the logic behavior observed in sequential logic gates such as the steady states and dynamic states (expected and unexpected transitions). Three basic gates are taken into account to illustrate these situations: C-element (Mller cell), D-type latch and D-type flip-flop, both with asynchronous reset signal. In the C-element circuit, when both inputs are equal the same logic value is presented at the output, and when the inputs are different the gate output keeps its previous state [5]. In the case of D-type latch with asynchronous reset, it is a level sensitive logic gate, *i.e.*, when the enable signal is high the value at the data input is transmitted to the output, and when the enable input is low the previous output is maintained. The D-type flip-flop, on the other hand, has a similar behavior to the latch but it is border sensitive circuit, *i.e.*, the value at the data input is only transmitted to the output when the clock input rises.

A. Steady States

In order to have the maximum test coverage, the analysis of the steady states of the circuit is a crucial task. The steady states are the combination of inputs and output values. Table I shows the steady states of the C-element, Table II presents the steady states of the D-type latch with asynchronous reset signal, and Table III shows the steady states of the D-type flip-flop with asynchronous reset.

It is worth to note that some input combinations can have two possible outputs. The reason is the memory effect of the

TABLE I
C-ELEMENT STEADY STATES

A	B	Previous Q	expected Q
0	0	X	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	X	1

TABLE II
D-TYPE LATCH WITH ASYNCHRONOUS RESET STEADY STATES

R	D	E	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	1	0	1
0	1	1	1
1	X	X	0

TABLE III
D-TYPE FLIP-FLOP WITH ASYNCHRONOUS RESET STEADY STATES

R	D	CLK	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	X	X	0

latch. In the flip-flop, it still is more significant. The reason for that is the fact that the latch is a level sensitive gate whereas the flip-flop is a border sensitive circuit. Therefore, the flip-flop presents more states to be covered. In order to cover these states, it is necessary to pass through a specific transition. The states with reset with the ”1” logic value were omitted because the output is stuck at the 0 logical value. The C-element has an interesting behavior compared with latch and flip-flop gates. Instead of having the memory behavior controlled by a single input, in this case, it is controlled by both of them. This behavior is desired in asynchronous circuit, where there is no global clock signal [5].

B. Dynamic States: Expected Transitions

Another important aspect of test coverage is the expected transitions, *i.e.*, when an input changes there is a transition at the output. Table IV shows the expected transitions of the C-element gate, Table V presents the expected transitions of the D-type latch, and Table VI shows the expected transitions of the D-type flip-flop.

Again, the difference among these circuits can be observed. The transition in the data input is only propagated to the output when the enable input is high or when the enable input rises the data input and this one differs from the current state of the latch. This propagation only occurs in the flip-flop when the clock signal rises and the data input differs from the current

TABLE IV
C-ELEMENT EXPECTED TRANSITIONS

A	B	Previous Q	expected Q
1	↑	0	↑
↑	1	0	↑
↓	0	1	↓
0	↓	1	↓

TABLE V
D-TYPE LATCH WITH ASYNCHRONOUS RESET EXPECTED TRANSITIONS

R	D	E	Previous Q	Q
0	0	↑	1	↓
0	↑	1	0	↑
0	↓	1	1	↓
↑	1	1	1	↓
↑	1	0	1	↓
↓	1	1	0	↑

TABLE VI
D-TYPE FLIP-FLOP WITH ASYNCHRONOUS RESET EXPECTED TRANSITIONS

R	D	CLK	Previous Q	Q
0	1	↑	0	↑
0	0	↑	1	↓
↑	X	X	1	↓

state of the flip-flop. The propagation of transitions in the C-element occurs when one input transitions to a value equal to the other, and this new input value differs from the current C-element state.

C. Dynamic States: Non-Expected Transitions

Besides testing expected transitions, it is important to test the non-expected transitions. These transitions occur when the input is transitioned but the output must stay stable. Table VII shows the non-expected transitions for C-element, Table VIII presents the non-expected transitions for latch, and Table IX shows the ones for the flip-flop.

The states show the memory effect of these circuits, when a transition does not propagate to the output. These transitions must be covered in order to completely test whether the gate is holding the state and not transitioning.

III. RELATED WORKS

As mentioned in the Introduction section, there are proposed works which show some approaches to test sequential cells, especially D-type latches with asynchronous set and reset and D-type flip-flop with asynchronous set and reset.

At [2], it is proposed a set of necessary conditions in order to fully test latches. Using a logical equation to describe the possible states of the gate, the paper delimiters some essential

TABLE VII
C-ELEMENT NON-EXPECTED TRANSITIONS

A	B	Previous Q	expected Q
↑,↓	0	0	0
↑,↓	1	1	1
0	↑,↓	0	0
1	↑,↓	1	1

TABLE VIII
D-TYPE LATCH WITH ASYNCHRONOUS RESET NON-EXPECTED TRANSITIONS

R	D	E	Previous Q	expected Q
0	↑,↓	0	X	X
1	↑,↓	X	X	X
1	X	↑,↓	X	X
↑,↓	X	X	0	0
0	0	↑,↓	0	0
0	1	↑,↓	1	1

TABLE IX
D-TYPE FLIP-FLOP WITH ASYNCHRONOUS RESET NON-EXPECTED TRANSITIONS

R	D	CLK	Previous Q	expected Q
0	↑,↓	X	X	X
0	X	↓	X	X
↑,↓	X	X	0	0

sequences that must be part of the checking experiment in order to cover all possible steady states. Despite detecting all faults of a logic gate, this solution misses some possible transitions, as well as it is not generic and it is not cyclic, requiring a reset signal.

At [3], the testing of D-type latch with asynchronous set and reset is done by instantiating a 12-bit shift-register with the same gate under test, as shown in Fig. 1. However, some latches have the set and reset signals behavior determined by the current overall state of the register, and the enable polarity signal is inverted at every couple of latches. The steady state coverage is almost 100% but the state where both set and reset are on is not covered. Moreover, some possible and unexpected output transitions are not covered by this approach.

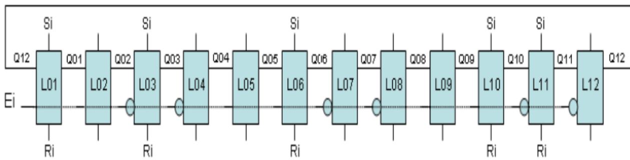

Fig. 1. Shift register setup described at [3]

For testing the D-type flip-flop with asynchronous set and reset, a modified 5-bit counter is created where each bit is the same gate under test, as shown in Fig. 2. Similar to the test of the latch, the set and reset signals of several bits are dependent of different states of the counter and are calculated by a handshake circuit. The test covers all steady states excluding the ones with both set and reset signals activated. The transition coverage is 50% of the unexpected transitions. Both solutions are specific either latch or to flip-flop, and it is not shown the way to create a shift-register to test other kind of sequential gates.

Another approach is defined at [4], where finite-state machine (FSM) is created using the circuit behavior description. The FSM passes through all steady states and signal transitions in order to create an input pattern sequence that has 100% of coverage. Despite being similar to this work, the mentioned

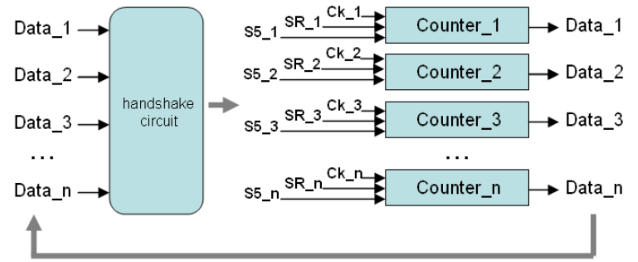

Fig. 2. Modified counter described at [3]

generator is dependent of the gate behavior and the proposed one is dependent only to the number of signals. Another problem is that the length of the sequence varies with the initial state. This solution, despite of having 100% of coverage, is not general aw well as it is not cyclic requiring a reset signal.

IV. Proposed Self-Check Setup

In this paper, it is proposed a universal generator for testing sequential logic gates, *i.e.*, a generator independent from the circuit behavior. The generator provides signals in a cyclic sequence, *i.e.*, a sequence that starts and terminates at the same input vector. This is desired because the generator can be left running to stops only when an error occurs. It covers also all input states and signal transitions. The signal transitions occur by changing one bit per step. Such a characteristic is important because it avoids timing race conditions that can cause meta-stability and raise false-positive errors. Another reason is that it is easier to debug when an error occurs.

This work is very similar to [2] and [4] due to the use of FSM strategy. The difference is that we disregard the gates behavior and only focus the number of inputs that the circuits have. Another difference between these two previous works and the one proposed is that this generator is cyclic and can run multiple times without external signals if necessary.

The proposed generator is part of a larger test bench. Fig. 3 shows the proposed generator and the test bench. Using the concept of self-checking, *i.e.*, the test bench does not need external clock signal to run. Instead, it creates its own temporizing signal. When using this principle, the generator must also provide a template that is compared with the gate output. Since the template signal is usually faster than the circuit output, the checker provides a 0 logic value. If the output is correct, then the checker provides 1 logic value, creating a rising border at the internal clock signal. Thus, making the generator provides the next states. If an error occurs, this rising edge does not occur, locking the generator at the current state.

V. Generator Model

A. Modeling

Since the proposed generator is universal, its behavior must be independent from the gate under test. Therefore, we ignore the gate memory effect and treat it as a black box. One

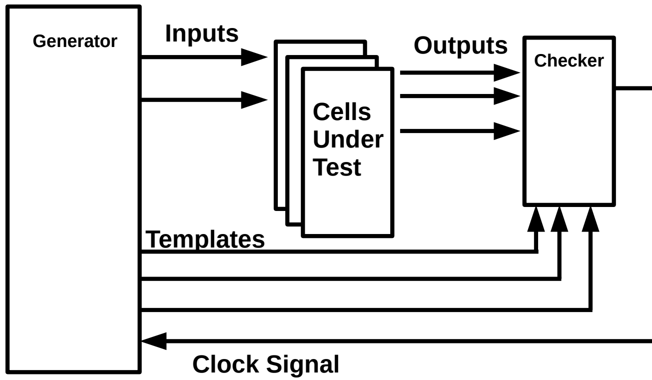

Fig. 3. The self-check setup

candidate would be the Gray code but it does not cover all possible states, as shown in Table X. Thus, a better model is necessary in order to solve such a deficiency.

TABLE X
3-BIT GRAY CODE AND ITS MISSING TRANSITIONS.

Gray Code	Missing Transitions
000	000 → 010 and 000 → 100
001	001 → 101 and 001 → 000
011	011 → 001 and 011 → 111
010	010 → 011 and 010 → 000
110	110 → 010 and 110 → 100
111	111 → 110 and 111 → 011
101	101 → 111 and 111 → 001
100	100 → 101 and 100 → 110

Using the content in Table X, a graph can be built. The nodes represent the possible states and the edges of the transitions. Fig. 4 shows the resulting graph of a 3-input gate. This kind of graph is called an n-cube graph or a hypercube. Since it is interesting to cover both rising transition, when a bit goes from logic value 0 to value 1, and falling transition, when a bit goes from logic value 1 to value 0, the graph must be bidirectional.

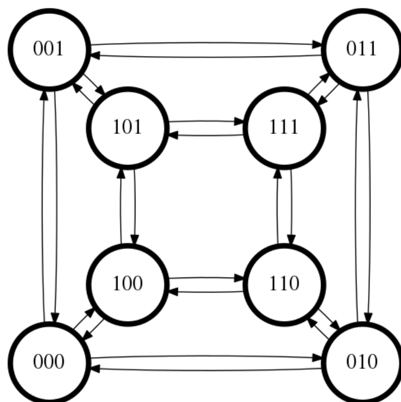

Fig. 4. 3-Dimensional Hypercube representing a 3 bit Gray Code

Once having built the graph, it is only necessary to find out an Euler cycle, *i.e.*, a cycle that passes through all edges exactly once, beginning and ending at the same node. Due to the form of the graph, it has many different Euler cycles and to find one, it is proposed a simple solution. The graph is transformed in a tree. Each node can be interpreted as a number and each node must have their sons with larger values and their father with smaller value. The root will have the lowest value possible, the state with every input at the 0 logic value. In order to preserve all possible transitions, some nodes must be repeated. Fig. 5 shows the resulting tree for the graph in Fig. 4. Once having the tree, it is only necessary to make a depth first search (DFS) in order to generate an Euler cycle.

B. Implementation

During the implementation, some shortcuts can be used in order to speed up the process. The graph building phase can be skipped and the tree can be represented with a table, such as shown in Table XI. The columns are the nodes and each row has the possible next node. The creation of this table can be done by flipping the bits with 0 in order to save only the numbers larger than the current. Fig. 6 shows the pseudocode for the creation of data shown in Table XI. The DFS can be performed by saving the current column, jumping to the first son and marking it as visited. If jumping to a column with no sons or with all sons visited, the algorithm jumps back to the previous node from which it came. The algorithm stops when all sons from the 0 state are visited. Fig. 7 shows the pseudo algorithm for the creation of the sequence.

C. Complexity

With a gate with N inputs, the possible states are 2^N different states. In order to calculate the larger values, each bit up to the *Nth* bit must be tested and inverted if necessary. Thus, the complexity for creation of data in Table is $N * 2^N$. The creation of the sequence is also $N * 2^N$. That is, in order to represent all possible transitions, it is necessary to storage and pass through $N * 2^N + 1$ states. Since the last state is the 0 state, it can be ignored and, in order to maintain this last transition, it is only necessary to return to the beginning of the list.

VI. RESULTS

In order to validate the proposed universal signal generator, a Java application has been created. In this application, the behavior of the circuit under test was described and the corresponding steady states and signal transitions calculated automatically. Afterwards, the behavioral model of the logic gate was submitted to the sequence stimuli transitions and the test coverage was evaluated. The first steady state of the generator output is set at the logic value 0. The gates

TABLE XI
TABLE CONTAINING EACH NODE ADJACENCY LIST REPRESENTING THE TREE IN FIG. 5

Nodes	000	001	010	011	100	101	110	111
Next Nodes	001	011	011	111	101	111	111	
	010	101	110		110			
	100							

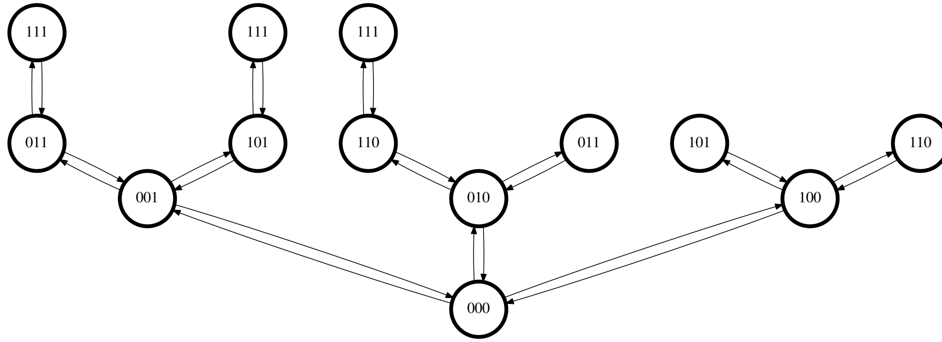

Fig. 5. The resulting tree from the graph in Fig 4

1: **Input:** The number of inputs of the CUT
2: **procedure** $Create_Table(N)$
3: $adjacency_table = [[] * N]$
4: **for** $i = 0; i < 2^N; i++$ **do**
5: $mask = 1$
6: **for** $j = 0; j < N; j++$ **do**
7: **if** $temp \wedge mask == 0$ **then**
8: $temp = i \oplus mask$
9: $adjacency_table[i].append(temp)$
10: **end if**
11: $mask \ll 1$
12: **end for**
13: **end for**
14: **return** $adjacency_table$
15: **end procedure**

Fig. 6. Pseudocode for creating Table of adjacency lists

1: Global pattern_list
2: Global adjacency_table
3: **procedure** $Create_pattern(node)$
4: pattern_list.append(node)
5: **for each** element **in** adjacency_table[node] **do**
6: $Create_pattern$(element)
7: pattern_list.append(node)
8: **end for**
9: mark_visited(node)
10: **end procedure**

Fig. 7. Pseudocode for creating the pattern sequence

used as case studies for such a validation was the D-type latch, D-type latch with asynchronous set, D-type latch with asynchronous reset, D-type latch with asynchronous set and reset, D-type flip-flop, D-type flip-flop with asynchronous set, D-type flip-flop with asynchronous reset, D-type flip-flop with asynchronous set and reset, and a Mller cell (or C-element). The generator outputs ordering of signal connections used was: output 0 is enable (E) or clock (Ck) inputs, output 1 is data (D) input, output 2 is reset (R) or set (S) input, and output 3 is set when reset signal is also available. Since there are vectors where set and reset are turned on, it was supposed that reset has priority over set, it means, when both set and reset asynchronous signals are activates the gate output goes down (value 0). Table XII shows the test coverage when considering a single instance of each sequential gate connected to the proposed generator.

TABLE XII
COVERAGE USING A SINGLE CELL

Cell	Steady State Coverage	Dynamic States Coverage
D-Latch	100%	75%
SD-Latch	100%	83.34%
RD-Latch	90%	80%
RSD-Latch	100%	90.27%
D-FF	87.5%	56.25%
SD-FF	91.67%	69.44%
RD-FF	83.34%	66.66%
RSD-Ff	100%	81.25%
C-element	100%	66.66%

Using a single instantiation of the circuit under test, 100% of coverage was not attained in some cases, neither in steady state coverage nor in dynamic coverage (transitions). It is resulting from the memory effect of sequential circuit. Some input to output signal transitions and states are hidden from the stimulus sequence of the generator. Fig. VI shows the modelling of a D-type flip-flop with asynchronous reset. The borderless green circle represents the possible input vectors that the corresponding output is expected to be 1, and the red border circle represents the input vector with the output in 0. When compared to the modelling in Fig. 4, it can be seen that some input vectors are repeated and there are more transitions, being some of them one-direction only.

Since it is desired to maintain the universal characteristics of the generator, the direct use of the circuit under test behavior cannot be used to create a specific sequence with higher coverage. Instead, the circuit behavior, the vector sequence and the multiple instantiations of the same gate can be previously calculated. This multiple instances can have none, one or more negated inputs, that means, the connection of the signal from the generator to the circuit is negated before arriving in the circuit input. Moreover, such a connection can be permuted, *i.e.*, not following the previous connection ordering. Table XIII shows the dynamic coverage using such a strategy. The number

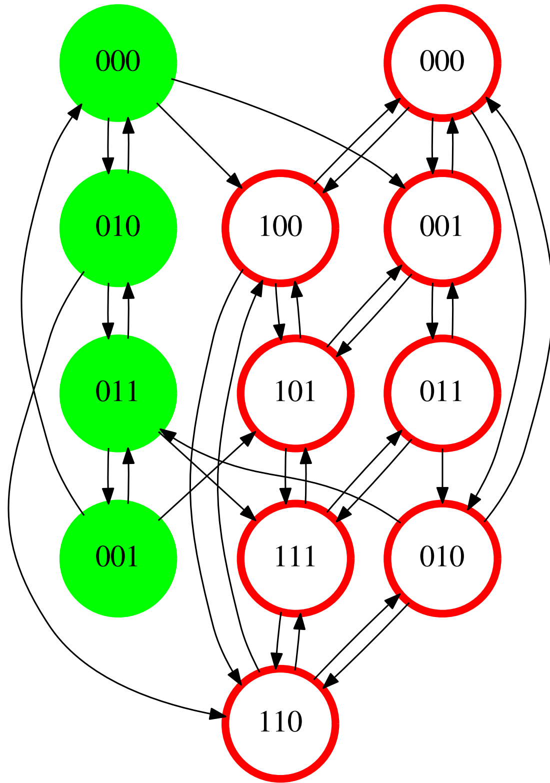

Fig. 8. D-type flip-flop with asynchronous reset model

of instances of the gate under test is the minimum necessary to achieve the maximum test coverage. By making so, all steady state coverage attained (100%), therefore it was omitted. As can be seen for almost all gates under test, the dynamic test coverage is 100%. It is possible because when permuting or negating the stimulus signals, the circuit is actually starting at a different state and passing through a different sequences without modifying the generator. In the case of the flip-flops without asynchronous signals (D-FF) and with reset signal (RD-FF), where the complete test coverage is not achieved, only a single dynamic state has not been observed, when clock signal is rising, data signal is at high value, reset is at low value, and the current output state is high and does not change.

In order to evaluate the scaling factor of the proposed approach in terms of circuit area, it is considered two possible designs for the generator: the first one by synthesizing the circuit from a Verilog description of the universal generator; the second one by applying a ROM block with the planned signal sequence. In the synthesis solution, the logic gates used were been restricted to only 2-input NOR (NOR2), inverter and flip-flop in order to estimate the resulting circuit size in equivalent gate metric, defined as a NOR2-based circuit area. Fig. 9 shows the number of logic gates used in the resulting map per number of outputs of the generator. As can be seen,

TABLE XIII
COVERAGE USING MULTIPLE INSTANCES

Cell	# of cells used	Dynamic State Coverage
D-Latch	2	100%
SD-Latch	3	100%
RD-Latch	4	100%
RSD-Latch	2	100%
D-FF	3	93.75%
SD-FF	4	100%
RD-FF	5	97.23%
RSD-Ff	3	100%
C-element	2	100%

the number of NOR2 and inverters grows exponentially with the number of bits at the output signal vector. The number of flip-flops grows linearly because they are only required in the counter circuit and in the output register.

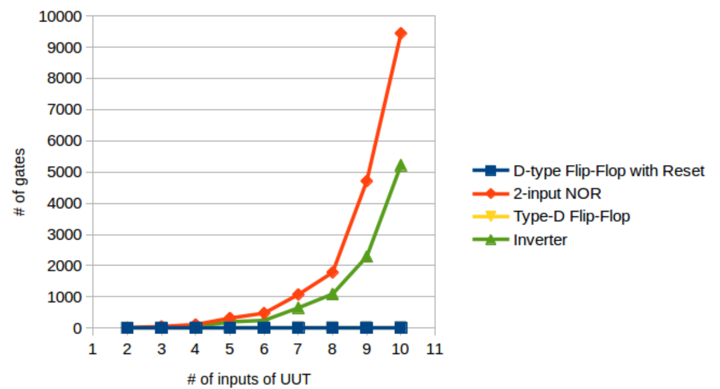

Fig. 9. Graph showing the numbers of gates per generator

The other possible physical implementation is the use of a read-only-memory (ROM) block. Since each signal state (output of generator) uses one line of the ROM, the resulting memory must be capable of mapping at least $N{*}2^N$ addresses. In order to profit of the full capacity of the memory, the sequence can be parted in N banks of memory with capacity for 2^N addresses. As in the previous design solution, through standard cells synthesis, this one grows exponentially as well.

VII. Conclusion

In this paper was proposed a general signal generator for testing standard cell libraries, in particular sequential logic gates (latches and flip-flops). Due to the inherent memory effect of these gates, even providing all possible single signal transition as stimuli, it was proven to be not sufficient in some cases to attain 100% of test coverage. On the other hand, the same generator can be applied to test several logic gates in parallel, reducing the area overhead. The generator was implemented in Java language, as proof-of-concept. The physical implementation of corresponding circuit is on progress.

References

[1] R. Ribas, S. Bavaresco, N. Schuch, V. Callegaro, M. Lubaszewski, and A. Reis, “Contributions to the evaluation of ensembles of combinational

logic gates," *Microelectronics Journal*, vol. 42, no. 2, pp. 371 – 381, 2011.

[2] S. R. Makar and E. J. McCluskey, "Checking experiments to test latches," pp. 196–201, Apr 1995.

[3] R. P. Ribas, Y. Sun, A. I. Reis, and A. Ivanov, "Self-checking test circuits for latches and flip-flops," in *2011 IEEE 17th International On-Line Testing Symposium*, July 2011, pp. 210–213.

[4] H. H. Avelar, P. F. Butzen, and R. P. Ribas, "Automatic circuit generation for sequential logic debug," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, Dec 2015, pp. 141–144.

[5] J. Spars and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*, 1st ed. Springer Publishing Company, Incorporated, 2010.