

Hidden Markov and Independence Models with Patterns for Sequential BIST

Laurent Bréhélin Olivier Gascuel Gilles Caraux Patrick Girard Christian Landrault

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

161 rue Ada, 34392 Montpellier Cedex 5, France.

E-mail: <brehelin, gascuel, caraux, girard, landrault>@lirmm.fr

Abstract

We propose a novel BIST technique for non-scan sequential circuits which does not modify the circuit under test. It uses a learning algorithm to build a hardware test sequence generator capable of reproducing the essential features of a set of precomputed deterministic test sequences. We use for this purpose two new models called Hidden Markov Model with Patterns and Independence Model with Patterns. Compared to existing methods, the proposed technique exhibits a very high fault coverage, including performance testing, at the expense of a low silicon area overhead.

1. Introduction

With the ever increasing complexity and density of present day integrated circuits, testing costs have become a significant part of overall chip costs. The Built-In Self Test (BIST) has been proposed as a powerful technique [1]. BIST design includes on-chip circuitry to provide test patterns and analyze output response. It performs the test within the chip so that the need for complex external testing equipment is greatly reduced. Many traditional testing problems (e.g., low accessibility of internal nodes) are overcome by using BIST.

A large number of techniques for BIST have been developed for combinational and full-scan sequential circuits [12, 11, 13]. However, these techniques are not directly applicable to non-scan sequential circuits. BIST techniques are introduced for synchronous sequential circuits because many performance-driven circuits and embedded-core systems do not use full-scan design strategies. These strategies require heavy circuit transformations (all or most flip-flops have to be transformed into multi-function cells), thus involving significant performance degradation and preventing at-speed testing of the circuit under normal operation conditions.

BIST requires both a compact Test Pattern Generator (TPG) for generating an effective sequence of test patterns,

and a compact response analyzer for compressing the output response of the system under test. The main goal of this paper is to focus on the former issue while proposing an efficient BIST TPG for synchronous sequential circuits.

Several solutions exist to generate test patterns during BIST of sequential circuits. One solution involves synthesizing a Finite State Machine (FSM) starting from its state transition diagram derived from the desired output sequence. The main applicability drawback of this approach lies in the limitations of current logic synthesis tools, that fail to fully optimize FSMs with more than some hundreds of states [4]. Another solution consists of adopting a deterministic TPG that autonomously generates a predetermined test sequence calculated from a sequential ATPG tool. In [8], such a deterministic BIST approach is presented, in which the TPG is composed of a sequence generator (e.g. a ROM that stores the compressed deterministic test sequence) and a decoder circuit. Although this approach provides high fault coverage with short test application time, the area overhead penalty limits its applicability to circuits containing a large number of flip-flops and few primary inputs.

In fact, test pattern generation during a BIST session traditionally relies on pseudo-random generators, either uniform or weighted, based on Linear Feedback Shift Registers (LFSRs) or Cellular Automata (CA) [1]. In [9], a parameterized TPG structure composed of comparison units driven by a k-bit counter is used to produce pseudo-random patterns in which selected bit values are fixed at 0 or 1 for several time units. However, this technique does not always yield the same fault coverage as deterministic sequences and requires excessively high test application times [8]. In [5], an original BIST architecture for FSMs that exploits CA as both pattern generator and signature analyzer is proposed. To the authors' knowledge, this approach is the most efficient solution proposed so far with respect to fault coverage and area overhead.

In this paper, we present a novel biased pseudo-random BIST technique for test pattern generation in non-scan sequential circuits. The key idea in this technique is to use a

set of deterministic test pattern sequences provided by a sequential ATPG program to build up a biased pseudo-random process capable of reproducing some essential features of these ATPG test sequences. The main advantage of this approach with respect to the area cost of the TPG is that it is not necessary to implement the complete set of deterministic test sequences but only a subset of rules to which these test sequences comply. These rules are expressed as test patterns that appear several times or as basic pattern series that are often used in ATPG sequences. Thus the proposed method generates sequences similar (or even identical) to those of the ATPG. We shall see (Section 6) that, at the expense of larger but reasonable test sequence sets, such a process effectively generates sequences with high fault coverage.

We defined two classes of stochastic processes in order to set up this biased pseudo-random process. The first one is called *Hidden Markov Model with Patterns (HMMP)* and is close to the classical *Hidden Markov Model (HMM)* introduced by Baum *et al.* in 1960-70 [2]. It offers the advantage of partially representing the pattern order in a test sequence while being simple enough to allow efficient hardware implementation. The second is a simplification of the first one and is called *Independence Model with Patterns (IMP)*. It is less “expressive” than HMMP since it cannot represent the pattern order in the sequences, but surprisingly it also achieves high fault coverage for several circuits and has the advantage of being implementable using less area overhead. The construction of an HMMP or IMP capable of mimicking the features of sequential ATPG test sequences is carried out by a learning algorithm.

The rest of the paper is organized as follows. In the next section, we present the features and properties of HMMPs. In Section 3, we explain how HMMP can be simplified into IMP and the main differences between both models. In Section 4, we describe the learning algorithm used to build up either an HMMP or an IMP from ATPG sequences. The complete hardware structures for both models are presented in Section 5. Experiments performed on ISCAS benchmark circuits are presented and discussed in Section 6. Concluding remarks are given in Section 7.

2. HMMP

2.1. Presentation

A Hidden Markov Model with Patterns is defined by its *structure* that is made up of *states* and *transitions*, and by probability distributions over the transitions; moreover, each state is labeled by a *pattern*. A pattern is a vector of length k that belongs to $\{0, 1, *\}^k$. k is the size of the test vectors, *i.e.* the number of PIs of the CUT plus some eventual bits that control the eventual set and/or reset lines avail-

able on the flip-flops of the circuit. The pattern associated with the state s is denoted as p_s . One or several transitions, labeled by their probability of being run over, start from each state and link it to other states. States and transitions define the *structure* of the HMMP. The set of states is denoted as S and $P(s \rightarrow t)$ is the transition probability from state s to state t . Two HMMP states are special: the *start* and the *end*. The *start* is used to initiate a new test sequence. The *end* indicates the end of the sequences. Both states are the only ones with no associated pattern. Figure 1.(a) represents an example of HMMP with five states and seven transitions.

2.2. Test sequence generation

The procedure consists in beginning on the *start* state, running over the transitions and generating a test vector for each state encountered by using the pattern associated with that state. The test vectors generated by a given pattern are consistent with this pattern and, moreover, the $*$ has equal probability of generating a 0 and a 1. For example, the pattern $*1*$ generates with probability $1/4$ each of the 4 vectors 010, 011, 110 and 111. Once the test vector has been generated, we choose (according to the associated probabilities) one of the transitions starting from the current state, and then go to the targeted state. This procedure is continued until the *end* is reached. A sequence of test vectors has then been generated, and another sequence can eventually be generated by going onto the *start* again.

2.3. Sequence generation probability

Pattern p is said to be *compatible* with pattern p_s if its fixed bits (those with value 0 or 1) have the same value or the value $*$ in p_s . For example, $p = 11*$ is compatible with $p_s = 1*0$, while $p = 11*$ is not compatible with $p_s = 100$. The probability is zero for the state s to generate a pattern p which is not compatible with p_s . The generation probability by a state s of a pattern p compatible with p_s depends on the number of bits which are fixed in p but have the value $*$ in p_s . Let $*_{p_s}^p$ denote this number. For example, if $p_s = 1***$ and $p = 10**$ then $*_{p_s}^p = 1$. Since $*$ has equiprobability of generating a 0 or a 1, the probability of generating the pattern p on the state s is given by the formula:

$$P(p|s) = \left(\frac{1}{2}\right)^{*_{p_s}^p}. \quad (1)$$

For example, if $p = 1*01*$ and $p_s = 1*010$, then $P(p|s) = 1$. On the other hand, if $p = 1*010*$ and $p_s = ***10*$, then $P(p|s) = \frac{1}{4}$.

Let $x = p_1 p_2 \dots p_l$ be a sequence of patterns. A common method for computing the generation probability of x

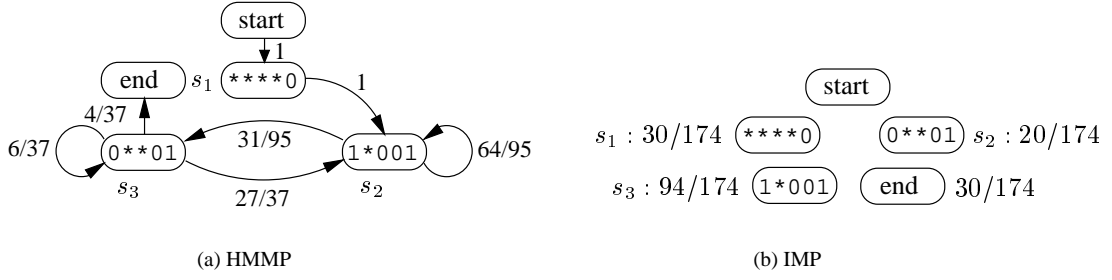


Figure 1. An example of HMMP (a) and IMP (b). Each state is labeled by its name and its associated pattern; HMMP transitions are labelled by their probability of being run over while the states of the IMP are labeled by their *a priori* probability.

by an HMMP H is to make the *Viterbi* assumption [7] that x can only be generated by a unique path (or sequence of states) through H . In other words, all paths except the most likely are assumed to have a negligible (or null) probability of generating x . This path is called the *Viterbi path* of x . For example, let us consider the following set of 3 sequences:

Sequence 1	Sequence 2	Sequence 3
****0	****0	****0
1*001	1*001	1*001
0*101	0*101	0*001
10*01	11*01	1*001
0**01		

The Viterbi path of the first sequence in the HMMP of Figure 1.(a) is $start - s_1 - s_2 - s_3 - s_2 - s_3 - end$. Moreover, this is the only path that can generate this sequence and the Viterbi assumption holds in this case.

Let $V_x = v_{p_0} \dots v_{p_{l+1}}$ (with $v_{p_0} = start$ and $v_{p_{l+1}} = end$) be the Viterbi path of the sequence $x = p_1 \dots p_l$. Then, under the Viterbi assumption, the generation probability of x by H is:

$$P(x|H) = P(v_{p_0} \rightarrow v_{p_1}) \prod_{i=1}^l P(p_i|v_{p_i})P(v_{p_i} \rightarrow v_{p_{i+1}}).$$

For Sequence 1 above, we have: $P(start \rightarrow s_1) = 1$, $P(p_1|s_1) = 1$, $P(s_1 \rightarrow s_2) = 1$, $P(p_2|s_2) = 1$, $P(s_2 \rightarrow s_3) = 31/95$, $P(p_3|s_3) = 1/2$, $P(s_3 \rightarrow s_2) = 27/37$, $P(p_4|s_2) = 1/2$, $P(s_2 \rightarrow s_3) = 31/95$, $P(p_5|s_3) = 1$, $P(s_3 \rightarrow end) = 4/37$. It follows that the probability for the HMMP in Figure 1.(a) of generating this sequence is: $31/95 \times 1/2 \times 27/37 \times 1/2 \times 31/95 \times 4/37 \approx 0.002$.

Let X be a set of sequences and $V = \{V_x, x \in X\}$ be the set of Viterbi paths associated with the sequences of X . The probability $P(X|H)$ of generating, with $|X|$ trials, the set X using H , is obtained (under the same assumption) by

the following formula:

$$P(X|H) = |X|! \prod_{x \in X} P(x|H). \quad (2)$$

3. IMPs

IMPs are defined from HMMPs by stating that the probability of running over a state does not depend on the actual state but only on an *a priori* probability associated with the state. In other words, we do not define transition probabilities between states but only a set of $|S|$ *a priori* probabilities associated with each state. The *a priori* probability associated with the state s is denoted $P(s)$. The *start* state is only used to begin a sequence and has no associated *a priori* probability. Figure 1.(b) represents an IMP with five states.

The procedure for generating a test sequence with an IMP is similar to that used for HMMPs and consists of: beginning on the *start* state, running over the states according to their *a priori* probability and generating a test vector for each encountered state until the *end* is reached.

As for HMMPs, for a given IMP H , we can associate with a sequence of patterns $x = p_1 p_2 \dots p_l$ the Viterbi path $V_x = v_{p_0} \dots v_{p_{l+1}}$ (with $v_{p_0} = start$ and $v_{p_{l+1}} = end$) of x in H . Under the Viterbi assumption, the generation probability of x by H is

$$P(x|H) = \left(\prod_{i=1}^l P(v_{p_i})P(p_i|v_{p_i}) \right) P(v_{p_{l+1}}).$$

For example, the Viterbi path of Sequence 1 of Section 2.3 in the IMP of Figure 1.(b) is $start - s_1 - s_3 - s_2 - s_3 - end$ and its generation probability is $30/174 \times 94/174 \times 20/174 \times 1/2 \times 94/174 \times 1/2 \times 20/174 \times 30/174 \approx 2.8 \cdot 10^{-5}$.

For a set of sequences X and a given IMP H , $P(X|H)$ is also defined by Formula (2).

4. Learning HMMP or IMP

Our aim is to build an HMMP or an IMP that generates sequences as similar as possible to those of the ATPG. We thus use a very classical principle in Statistical Modeling and Machine Learning. We consider that the higher the probability that an HMMP/IMP generates the ATPG sequences, the more *actually* generated sequences are similar to these sequences, and then the higher their fault coverage. We designed a learning algorithm based on this principle and on the state merging method which is a natural and efficient way to learn automata and HMMs [10].

The main algorithm of the learning procedure is the same for both HMMPs and IMPs. It involves building a large initial model that represents ATPG sequences and then compressing this model by iteratively merging "similar" states, until a sufficiently small HMMP/IMP is obtained. Sections 4.1 and 4.2 describe the initial model building procedure while Sections 4.3 and 4.4 describe the state merging procedure. Section 4.5 describes the criterion used to select the state pair to be merged. Figure 2 details the steps of the algorithm when applied to the 3 sequences of Section 2.3 for both HMMPs and IMPs.

4.1. Building the initial HMMP

The initial HMMP H_0 accurately represents the sequences from X and is obtained by building the *prefix tree* of X . A prefix tree is a tree for which the common prefixes are not repeated but represented by a unique path. The root of the tree is the *start* state and each path from this state to the leaves describes a sequence of X .

Next, we attach to each state s its pattern p_s and the parameter values associated with the Viterbi paths. In H_0 , Viterbi paths are naturally described by the sequences and the Viterbi assumption holds. Therefore, n_s is equal to the number of leaves of the sub-tree with root s , and, in the same way, $n_{s \rightarrow t}$ is equal to the number of leaves of the sub-tree with root t .

According to the maximum likelihood principle, the values of probabilities associated with H_0 transitions are estimated using the formula:

$$P(s \rightarrow t) = \frac{n_{s \rightarrow t}}{n_s}. \quad (3)$$

Finally, we create the *end* state to which every leaf is linked with transition probability 1. The HMMP H_0 of Figure 2 is the initial HMMP obtained from the sequence set of Section 2.3.

4.2. Building the initial IMP

The initial IMP is obtained by creating one state for each different pattern of ATPG sequences. Then we associate

with each state s its pattern p_s and the number of times n_s this pattern is used in ATPG sequences. We create the *start* and *end* states and associate with the latter the number of times it is used, *i.e.* the number of sequences provided by the ATPG. Let n_{total} be the total number of patterns in ATPG sequences. Then according to the maximum likelihood principle, the values of the *a priori* probabilities associated with the states of H_0 are estimated using the formula:

$$P(s) = \frac{n_s}{n_{total}}. \quad (4)$$

The IMP H_0 of Figure 2 is the initial IMP obtained from the sequence set of Section 2.3.

4.3. State merging in HMMPs

When two states have been selected (the criterion used is described in Section 4.5), they are merged in a very natural way. See for example the grey states s and s' of H_0 in Figure 2. s and s' are deleted and replaced by a new state u in H_1 . The *in* ($t \rightarrow s$ and $t' \rightarrow s'$) and *out* ($s \rightarrow end$ and $s' \rightarrow end$) transitions to and from the deleted states are connected to the new state, then the resulting potential double transitions ($u \rightarrow end$) are also merged.

As the HMMP structure is modified, its parameters have to be updated. We assume (as usual [10]) that Viterbi paths are not altered by merging, and that the new Viterbi paths are inferred from the previous ones by replacing s and s' by u . This assumption provides an efficient way to update the parameters. In H_1 (Figure 2) we have: $n_u = n_s + n_{s'}$, $n_{t \rightarrow u} = n_{t \rightarrow s}$, $n_{t' \rightarrow u} = n_{t' \rightarrow s'}$ and $n_{u \rightarrow end} = n_{s \rightarrow end} + n_{s' \rightarrow end}$. Finally, probabilities attached to the updated transitions are computed using Formula (3). Note that only parameters associated with the new state and its adjacent transitions need to be updated; the merging has no effect on the other states and transitions. This fact provides the basis for an efficient implementation of the learning algorithm described in [3].

γ	1	0	*	$\bar{*}$
1	1	$\bar{*}$	1	$\bar{*}$
0	$\bar{*}$	0	0	$\bar{*}$
*	1	0	*	$\bar{*}$
$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$	$\bar{*}$

Table 1. Response table of the bit merging operator.

The pattern associated with the new state is computed by merging bit after bit the patterns attached to both previous states. Let γ denote the bit merging operator. The character $*$ means that the value of the bit is not important. Therefore, $\gamma(*, 0) = 0$ and $\gamma(*, 1) = 1$. But two patterns are not always compatible. In this case, some of their bits differ and the merged pattern must generate both 0 and 1 on these

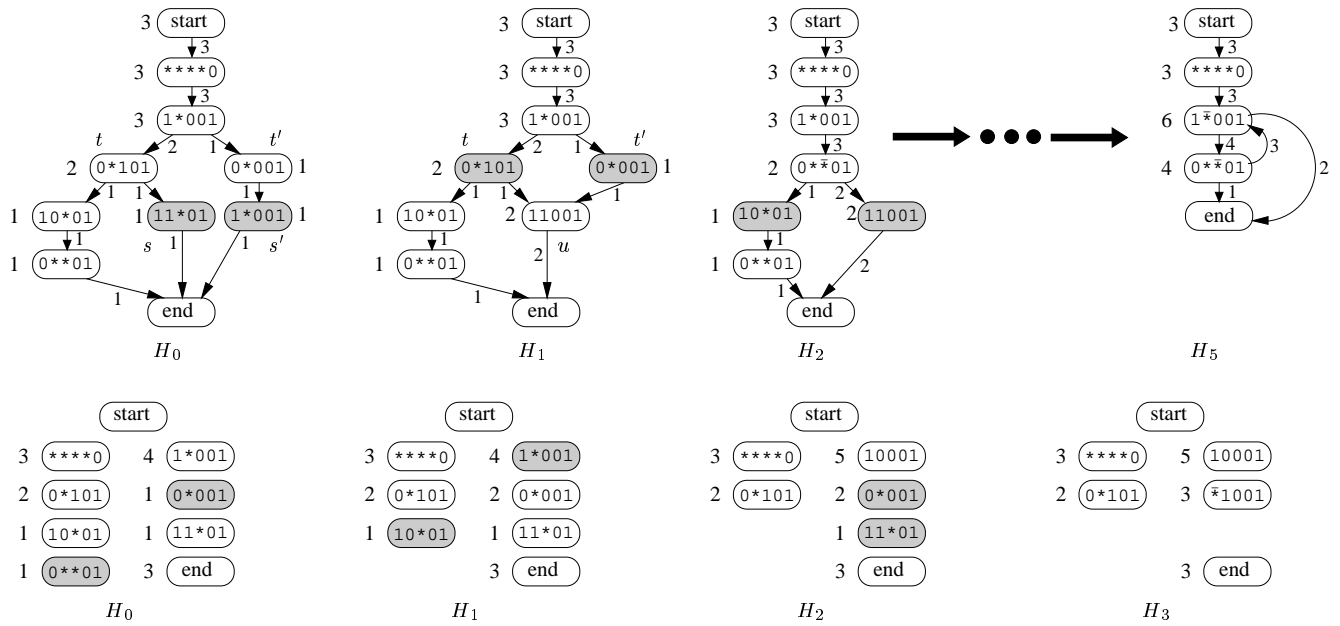


Figure 2. Structure compression achieved by state mergings for HMMPs and IMPs. The HMMPs/IMP are obtained by merging the grey states. Each state is labeled by n_s . For HMMPs, each transition $s \rightarrow t$ is labeled by $n_{s \rightarrow t}$ and $P(s \rightarrow t) = n_{s \rightarrow t}/n_s$ (c.f. Formula (3)). For IMPs, $P(s) = n_s/n_{total}$ (c.f. Formula (4)) with $n_{total} = 16$.

bits. These bits take the value * in the merged pattern and are marked to retain that this * results from the merging of a 0 and a 1. Let $\bar{*}$ denote the marked *. During further steps of the learning algorithm, this information is needed to avoid setting this bit to 0 or 1 value. The merging operator takes this into account, and we have: $\gamma(0, 1) = \bar{*}$, and after, $\gamma(\bar{*}, 0) = \bar{*}$, $\gamma(\bar{*}, 1) = \bar{*}$ and $\gamma(\bar{*}, \bar{*}) = \bar{*}$. Table 1 summarizes the response table of the γ operator.

4.4. Merging in IMPs

The procedure is similar to that employed for HMMPs. The chosen states s and s' are deleted and replaced by a new state u . Using the same hypothesis on the conservation of Viterbi paths, we have $n_u = n_s + n_{s'}$, $p_u = \gamma(p_s, p_{s'})$ and the *a priori* probability of the new state is computed using Formula (4). Note that parameters associated with the other states do not have to be updated.

4.5. Selection of the best state pair

The procedure is the same for both HMMPs and IMPs. The aim of the learning algorithm is to reduce the HMMP/IMP structure while keeping as high as possible the probability (given by Formula (2)) of generating the ATPG sequences. So at each step the algorithm chooses the state pair which, when merged, keeps highest this probability. Nevertheless, sometimes (especially at the beginning of the

process) many pairs agree with this criterion. Then among these pairs we choose that for which pattern merging involves fixing the lowest number of bits. The number of bits fixed by merging the two patterns p and p' is obtained by the formula:

$$\varphi(p, p') = \min(*_p^{\gamma(p, p')}, *_p'^{\gamma(p, p')}). \quad (5)$$

For example, if $p = 1*11*$ and $p' = *0\bar{*}**$, then $\gamma(p, p') = 10\bar{*}1*$; we have $*_p^{\gamma(p, p')} = 1$, $*_{p'}^{\gamma(p, p')} = 2$ and then $\varphi(p, p') = 1$. Using Formula (5) has two justifications. First, at the beginning of the learning procedure, this avoids fixing bits in the patterns too soon, which would make further mergings more difficult (in terms of generation probability). Secondly, we can reasonably assume that similar patterns quite likely play the same part in sequences. Such an argument is very classical in Machine Learning.

5. Hardware implementation

After the HMMP or the IMP for a given circuit has been determined and the generated test set meets the fault coverage requirement, an implementation of the BIST TPG can be performed. As in any BIST hardware implementation, the main objective is to implement this sequential test generator with a low area overhead cost.

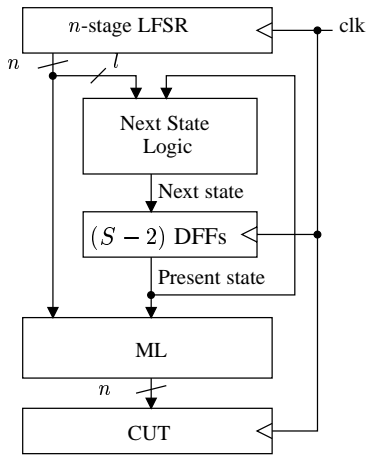


Figure 3. Block diagram of the BIST TPG architecture.

5.1. HMMP hardware implementation

The HMMP is implemented by a Mealy finite state machine fed by the outputs of a n -stage primitive LFSR (see Figure 3). The state coding in the Mealy machine is a 1-out-of- n coding, *i.e.* $(S - 2)$ D-type flip-flops are required for an S state HMMP (*start* and *end* states are not implemented) and only one DFF is set at 1 at a given time period. Outputs of the LFSR considered as random inputs are used:

- first, to implement the weighted transitions of the finite state machine by way of the next state function of the Mealy machine;
- secondly, to produce the don't care values (denoted * and $\bar{*}$ in the previous sections) of test patterns at the output of the Mapping Logic ML (the output function of the Mealy machine), the constant part of these test patterns being determined by using of the present state value.

The LFSR, Mealy machine and CUT are supposed to be synchronized and use the same clock.

Let us now describe the way to design each block of the BIST architecture from a given HMMP. This can be better explained with an example. Let us assume the HMMP example depicted in Figure 1.(a) and determined according to the learning process described in the previous section. The corresponding BIST architecture is depicted in Figure 4.(a). As we said, the *start* and *end* states are not implemented, so since we want to use our generator in continuous way, the incoming states to the *end* are directly linked to the outgoing states from the *start* with probability equal to the product of both probabilities. In our example, the only incoming state to the *end* (s_3) is linked to the only outgoing state from the *start* (s_1) and this transition is labeled

with probability $4/37 \times 1 = 4/37$. The interconnections between DFFs are determined from the transitions in the HMMP representation. For example, three transitions lead to state s_2 in Figure 1.(a), so three lines are ORed to the inputs of the corresponding DFF (Dff_2) in Figure 4.(a). Probabilities associated with the transitions are discretized to be as close as possible to the values calculated during the learning process (discretization of probabilities is carried out through an algorithm which is not described in this paper for space consideration) and implemented using demultiplexers, NOR gates and output lines from the LFSR. For example, transition $s_3 \rightarrow s_1$ has probability $4/37$ (see above) and is approximated by the probability $1/4$ (NOR output) $\times 1/2$ (LFSR output) = $1/8$ in the hardware implementation.

The mapping logic ML is designed as follows. Considering the HMMP example given in Figure 1.(a), three test patterns ($****0$, $1*001$ and $0**01$) have to be produced by ML. As mentioned earlier, unspecified bits (* or $\bar{*}$) are produced by LFSR outputs and constant values (0 or 1) are set by using the present state information. Patterns are implemented through logic synthesis. For example, when test pattern p_{s_1} is selected, only the last bit has to be set at 0. Other bits remain unspecified. Since this bit position is set at 1 for the two other patterns, only an inverter at the corresponding input of the CUT is needed to implement this test pattern. Now, if test pattern p_{s_3} is selected in a subsequent clock cycle (this depends on the path actually followed in the HMMP), two bits have to be set at a specified value: the first and fourth bits must take the value 0. This is accomplished by placing two NOR gates with an inverting input on the first and fourth bit positions in ML. The internal structure of the mapping logic for this example shows that a small number of gates is needed for the ML implementation. Note that for circuits with bigger Selecting Logic and ML blocks, a logic synthesis tool can be used to minimize the silicon area of the TPG design.

5.2. IMP hardware implementation

As suspected, and according to the description in Section 3, the hardware implementation of an IMP for a given CUT will be less expensive than that obtained from an HMMP. The way to design each block of the BIST architecture from a given IMP is very similar to that presented for an HMMP. In fact, the main difference between the two resulting implementation states is the absence of DFFs in the IMP architecture, which corresponds to the missing transitions in the IMP representation. For this reason, we do not describe the way to design an IMP BIST architecture. However, for the reader's information, we show in Figure 4.(b) the resulting BIST TPG architecture for the IMP example depicted in Figure 1.(b).

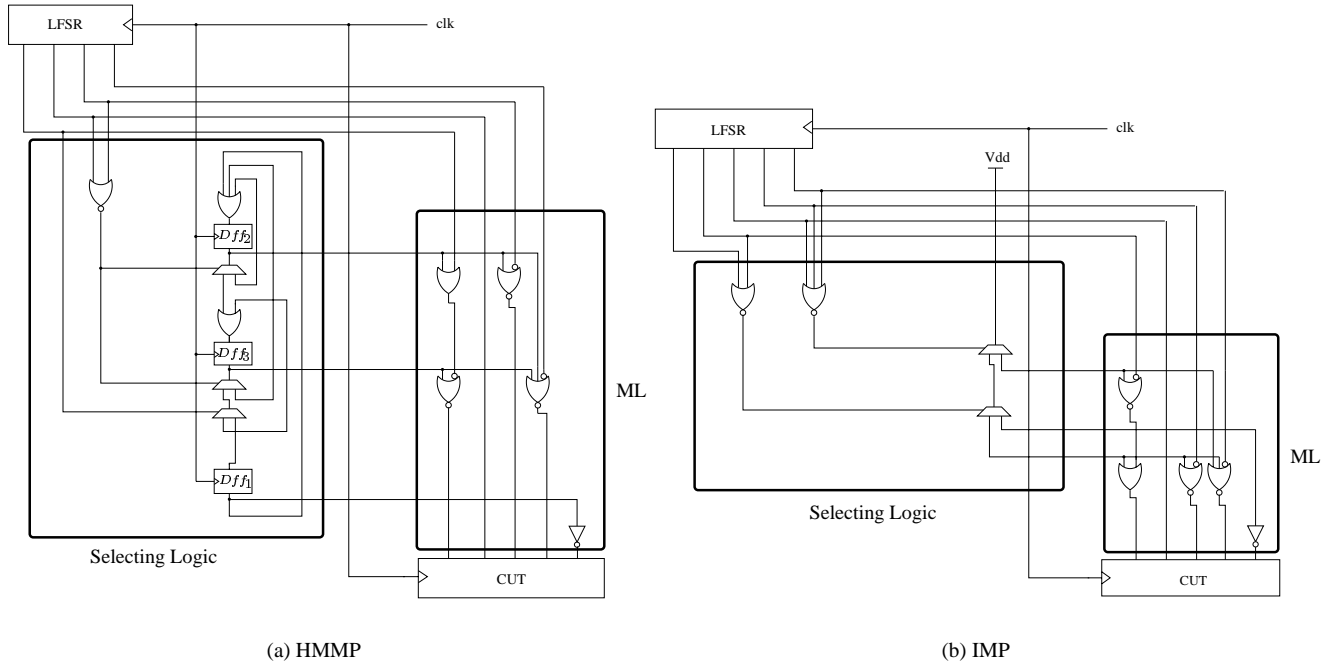


Figure 4. BIST architecture.

Name	CUT			ATPG		LFSR		HMMP					IMP			
	#PI	#FF	#F	Length	%Cover	Length	%Cover	Length	%Cover	#State	#Edge	%GE	Time	%Cover	%GE	Time
s298	3	14	596	2472	89.9%	1000	63.8%	1000	89.9%	3	6	13.2%	5.5	89.3%	5.7%	0.01
s344	9	15	670	414	97.6%	1000	93.9%	1000	97.6%	3	7	18.3%	0.3	97.6%	9.0%	0.02
s382	3	21	764	13930	94.0%	1000	14.4%	5000	94.5%	4	9	15.7%	327.2	89.8%	6.9%	0.02
s386	7	6	772	736	90.2%	1000	50.3%	2000	89.6%	3	7	11.7%	0.6	87.0%	3.5%	0.03
s420	18	16	916	8434	64.8%	5000	69.4%	1000	74.3%	3	5	8.2%	17.1	73.1%	2.9%	0.01
s444	3	21	888	15510	91.8%	1000	13.4%	5000	93.4%	4	11	15.9%	322.8	88.6%	6.4%	0.01
s526	3	21	1052	3032	56.8%	1000	10.8%	5000	85.5%	5	14	14.9%	62.5	83.8%	5.6%	0.01
s526n	3	21	1052	3706	59.9%	1000	10.8%	5000	84.5%	4	9	10.3%	65.4	84.8%	4.3%	0.01
s641	35	19	1278	1194	88.1%	2000	87.1%	2000	88.1%	3	6	10.7%	3.7	88.0%	7.4%	0.07
s713	35	19	1426	1250	83.9%	2000	82.9%	1000	83.9%	4	9	14.1%	4.7	83.9%	9.3%	0.09
s820	18	5	1640	4540	96.3%	1000	39.8%	28000	84.0%	6	17	18.0%	25.0	72.1%	9.8%	1.12
s832	18	5	1664	4626	95.3%	1000	40.4%	28000	82.3%	6	17	17.7%	24.5	72.0%	10.0%	1.10
s1196	14	18	2392	926	99.9%	10000	96.8%	1000	98.2%	6	20	15.7%	1.0	98.0%	8.0%	0.14
s1238	14	18	2476	988	96.4%	5000	90.6%	1000	93.7%	6	22	15.5%	0.3	93.5%	8.6%	0.06
s1423	17	74	2846	1538	64.3%	2000	38.4%	5000	89.2%	6	24	10.0%	22.2	77.3%	5.8%	0.30
s1488	8	6	2976	5680	98.6%	1000	37.3%	10000	97.2%	7	17	8.2%	9.3	96.8%	4.8%	0.70
s1494	8	6	2988	5258	98.1%	1000	36.8%	10000	96.8%	7	16	8.2%	20.6	93.6%	4.7%	0.68
s3330	40	132	6660	5958	79.5%	2000	68.3%	5000	78.2%	8	43	6.9%	628.8	77.2%	4.1%	15.73
s5378	35	179	10590	1602	65.0%	1000	65.0%	5000	69.6%	4	10	2.4%	1.4	72.2%	1.4%	0.39
s13207	62	638	26358	3018	31.6%	1000	31.9%	1000	37.6%	4	7	0.7%	5.8	37.3%	0.8%	0.06
Mean:					82.1%		52.1%		85.4%			11.8%		82.8%	5.9%	

Table 2. Fault coverage achieved by ATPG, LFSR sequences, LFSR+HMMPs and LFSR+IMPs.

6. Experimental results

The performances of our method have been evaluated on the ISCAS'89 benchmark circuits. Tests were carried out for every circuit in the following manner. Ten pseudo-random sequences generated with an LFSR using various seeds were simulated, and that with the highest fault coverage was retained. Such pseudo-random sequences detect numerous (easy) faults at the beginning of the process. Then the frequency of new detected faults decreases, and after a certain time no new faults seem to be detectable by the sequence. So the length of the pseudo-random sequences was limited to the efficient beginning part. Next we used a sequential ATPG to generate a set of deterministic test sequences. Sequences detecting only faults already detected by the pseudo-random sequence were eliminated, and we ran our learning procedure on the remaining sequences. Learned HMMPs/IMPs were implemented as described in Section 5 and used to generate 10 sets of biased sequences. The total length of these sequences was chosen in the same way as for the pseudo-random sequences. Finally, among these biased sequence sets we retained that detecting the greatest number of faults non-detected by the previously selected pseudo-random sequence. Note that our method can be adapted for all the set(s) and/or reset(s) configurations of memory cells. In our experiments, we considered that a global reset is available on each CUT, so ATPG sequences all begin with such a reset.

The results are reported in Table 2. After the name of the CUT, its number of primary inputs (#PI), memory cells (#FF) and potential faults (#F), we provide the total length and fault coverage of the ATPG sequences. In the following columns, we indicate the length of the pseudo-random sequences and the fault coverage achieved by the best among ten. The next columns indicate the results obtained using HMMPs and IMPs. The Length column indicates the total length of the sequences generated by both HMMP and IMP. The %Cover columns indicate the best fault coverage achieved in the 10 simulations (*i.e.* the faults detected by the best HMMP/IMP sequence set plus those detected by the pseudo-random sequence). The #State column indicates the number of states for both HMMPs and IMPs, while the next column indicates the number of non-null transitions of the HMMPs. %GE columns indicate the ratio of area overhead used to implement the HMMP/IMP with respect to the size of the CUT. Both the area overhead and the size of the CUT were computed in number of Gate Equivalents (GE) as follows: $0.5n$ GE is counted for an n -input NAND or NOR gate, $0.5(n+1)$ GE for an n -input AND or OR gate, 0.5 GE for a NOT gate, 1.5 GE for a demultiplexer, and 4 GE for a flip-flop. Note that the area overhead includes all components in the Selecting Logic and ML boxes depicted in Figure 4 and corresponds to the implementation proposed

in Section 5 before logic optimization. Finally, the Time columns indicate the computing time of the learning procedure (in seconds) on a Pentium III 350 Mhz PC.

First, we see that using HMMP or IMP improves the fault coverage achieved by the pseudo-random sequence. Next, we observe that this fault coverage is often equal to (s344, s713), sometimes smaller (s820, s832) and sometimes larger (s526, s1423 and general means) than the fault coverage of the ATPG sequences. This result is surprising and clearly confirms the validity of our approach. This demonstrates that it is possible to infer very efficient construction rules from ATPG sequences. These rules do not guarantee exact generation of the original sequences, but they achieve high fault coverage when the biased sequence set is large enough. The good results of our method could also be explained by the fact that for some circuits ATPG fails to achieve good fault coverage within a reasonable CPU time (*e.g.* for s13207 the ATPG ran for several days), and on the other side by the easiness of achieving relatively high fault coverages for some circuits (see results obtained by the LFSR alone). Compared with HMMPs, it appears that for several circuits IMPs achieve almost as good fault coverage, while the area overhead is reduced to about half. Nevertheless, for some circuits having a high sequential depth (*e.g.* s820, s832) IMPs fail to achieve acceptable fault coverage and HMMPs have to be used. Moreover, neither IMP nor HMMP have been able to improve the fault coverage of the pseudo-random sequence in the case of the s991 circuit (data not shown). The reason is that all the faults not detected by the LFSR (but detected by the ATPG) are *combinational* faults detected by a great number of single patterns which cannot be easily merged. However, note that this circuit is the sole for which we encountered such difficulties.

Finally, we can observe that the %GE values decrease when the size of the CUTs increases for both HMMPs and IMPs, and become very low for the biggest circuits.

#States	HMMP		IMP	
	%Best	%GE	%Best	%GE
7	97.2%	8.2%	96.8%	4.8%
6	97.1%	7.4%	95.6%	4.0%
5	96.6%	6.6%	93.5%	3.6%
4	94.2%	5.0%	93.1%	2.6%

Table 3. Fault coverage related to the HMMP/IMP size.

Concerning the choice of the number of states, a good solution is to simulate HMMPs/IMPs with decreasing sizes (as obtained from the learning algorithm) and to select the best one according to the size and fault coverage constraints. For example, Table 3 shows the fault coverage and the area

overhead related to the number of states for the s1488 circuit. Strong size constraints will likely lead to choosing an IMP with 4 states, while the IMP with 7 states will be chosen when more attention is given to the fault coverage.

7. Conclusion

We presented a new BIST technique for the generation of test sequences in non-scan sequential circuits. Our method uses a learning algorithm to build a biased pseudo-random process capable of reproducing the essential features of deterministic sequences provided by an ATPG. We proposed two different models to capture the features of these sequences and we showed experimentally that reproducing these features is enough to achieve high fault coverage. We also proposed a low cost area overhead implementation for both models. Our method provides an easy trade off between hardware overhead and test efficiency. Another important point is that the results are closely associated with the efficiency of the ATPG, so our method should benefit from further development of incoming ATPGs.

Nevertheless, our technique could be improved in several ways. First, for some CUTs the fault coverage needs to be improved. A solution could be to weight ATPG sequences with the number of faults they detect and with the difficulty of these faults. Another improvement could be to improve the first pseudo-random stage, currently performed by a simple LFSR, by low cost techniques used for combinational circuits in order to first detect the combinational faults. For example, seeding technique [6] should be efficient for the s991 circuit. Next, we think that the area overhead may be significantly reduced by using logic optimization as well as techniques based on fault simulation.

References

- [1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. 1990.
- [2] L. E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in statistical analysis of probabilistic functions in Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.
- [3] L. Br  h  lin, O. Gascuel, and G. Caraux. Hidden Markov Models with Patterns to learn boolean vector sequences; application to the built-in self-test for integrated circuits. Technical Report 99006, University of Montpellier II, LIRMM, Jan. 1999.
- [4] S. Chiusano, F. Corno, P. Prinetto, and M. S. Reorda. Cellular automata for deterministic sequential test pattern generation. In *Proceedings of the ACM*, pages 60–65, Nov. 1997.
- [5] F. Corno, N. Gaudenzi, P. Prinetto, and M. Sonza-Reorda. On the identification of optimal cellular automata for built-in self-test of sequential circuits. In *Proceedings of the VTS*, pages 424–429, 1998.
- [6] C. Fagot, O. Gascuel, P. Girard, and C. Landrault. On calculating efficient LFSR seeds for built-in self test. In *IEEE ETW*, May 1999.
- [7] G. D. Forney. The Viterbi algorithm. *Proc. IEEE*, 61(3):268–278, Mar. 1973.
- [8] V. Iyengar, K. Chakrabarty, and B. Murray. Built-in self-testing of sequential circuits using precomputed test sets. In *Proceedings of the VTS*, pages 418–423, 1998.
- [9] I. Pomeranz and S. M. Reddy. Built-in test generation for synchronous sequential circuits. In *Proceedings of the ACM ICCAD*, pages 421–427, Nov. 1997.
- [10] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the ICGI*, volume 862 of *LNAI*, pages 106–118, Berlin, 1994.
- [11] N. Toubia and E. McCluskey. Altering a pseudo-random bit sequence for scan-based BIST. In *Proceedings of the ITC*, pages 167–175, Oct. 1996.
- [12] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick. An efficient BIST scheme based on reseeding of multiple polynomial linear feedback shift registers. In *Proceedings of the IEEE/ACM ICCAD*, pages 572–577, Nov. 1993.
- [13] H.-J. Wunderlich and G. Kiefer. Bit-flipping BIST. In *Proceedings of the IEEE/ACM ICCAD*, pages 337–345, Nov.10–14 1996.