Testability of Switching Lattices in the Stuck at Fault Model

Anna Bernasconi
Dipartimento di Informatica
Università di Pisa, Italy
anna.bernasconi@unipi.it

Valentina Ciriani Luca Frontini
Dipartimento di Informatica
Università degli Studi di Milano, Italy
{valentina.ciriani, luca.frontini}@unimi.it

***Abstract*—Switching lattices are two-dimensional arrays of four-terminal switches proposed in a seminal paper by Akers in 1972 to implement Boolean functions. Recently, with the advent of a variety of emerging nanoscale technologies based on regular arrays of switches, synthesis methods targeting lattices of multi-terminal switches have found a renewed interest. In this paper, the testability under the stuck-at-fault model (SAFM) of switching lattices is analyzed, and properties of fully testable lattices are identified and discussed. Experimental results are given to analyze the testability of lattices synthesized with different methods.**

***Index Terms*—Switching lattices; testability; logic synthesis.**

I. Introduction

A switching lattice is a two-dimensional lattice of four-terminal switches linked to the four neighbors of a lattice cell, so that these are either all connected, or disconnected. A Boolean function can be implemented by a lattice associating each four-terminal switch to a Boolean literal, so that if the literal takes the value 1 the corresponding switch is ON and connected to its four neighbors, otherwise it is not connected. The function evaluates to 1 if and only if there exists a connected path between two opposing edges of the lattice, e.g., the top and the bottom edges (see Figure 1 for an example). The synthesis problem on a lattice consists in finding an assignment of literals to switches in order to implement a given target function with a lattice of minimal size.

The idea of using regular two-dimensional arrays of switches to implement Boolean functions dates back to a seminal paper by Akers in 1972 [2], but has found a renewed interest recently, thanks to the development of a variety of nanoscale technologies. Synthesis algorithms targeting lattices of multi-terminal switches have been designed [3], [5], [13], [14], and methods based on function decomposition techniques have been exploited to mitigate the cost of implementing switching lattices [8], [9], [10]. Moreover, several studies on fault tolerance for nano-crossbar arrays have been published recently [4], [15], [16], [17].

Besides synthesis and fault tolerance, testability is a major aspect of the design process. While detailed studies on testability have been performed for standard two-level and three-level networks (see for instance [1], [6], [7], [11], [12], [18]), to the best of our knowledge, the testability of switching lattices has not been considered so far. Therefore, in this paper, we study redundancies of lattices under a static fault model: the stuck-at-fault model (SAFM). In particular, we prove that under the SAFM, switching lattices minimized with respect to the number of literals controlling the switches are free of redundancies by construction. Whereas, it can be shown by counter examples that lattices minimized with respect to the number of switches, i.e. minimized with respect to the size, are not in general fully testable. We also identify the properties that make a switching lattice fully testable in the SAFM, and show how these properties resemble the properties that guarantee the full testability of the SOP forms in the SAFM, i.e., the primality of the products and the irredundancy of the cover. Finally, we propose a method for identifying redundant cells in a lattice. We conclude the paper reporting experimental results regarding the testability of lattices synthesized with two different methods [5], [13].

II. Preliminaries

A. Fault Models (FMs)

The standard stuck-at faults model (shortly, SAFM) is well-known and used throughout the industry for many years [1], [11]. In SAFM it is assumed that a defect causes a basic cell input or output to be fixed to either 0 or 1, i.e., signal lines can assume constant values independent of the inputs.

Definition 1: A *stuck-at fault* with fault location v is a tuple $(v[i], \epsilon)$ or $([i]v, \epsilon)$. $v[i]$ ($[i]v$) denotes the i-th input (output) pin of v, $\epsilon \in \{0, 1\}$ is the fixed constant value.
In the following we simply speak of stuck-at-0 (SA0) and stuck-at-1 (SA1) faults. Now, let C be any combinational logic circuit over a fixed library.

Definition 2: An input t to C is a *test* for a fault f, iff the primary output values of C on applying t in presence of f are different from the output values of C in the fault free case.

A fault is *testable*, iff there exists a test for this fault. The goal of any test pattern generation process is a *complete* test set for the circuit under test, i.e., a test set that contains a test for each testable fault. The construction of complete test sets requires the determination of the faults which are not testable (= *redundant*), even though it is easy to see that in general the detection of redundancies is *coNP-complete*. Redundancies have further unpleasant properties: they may

978-1-5386-4756-1/18/$31.00

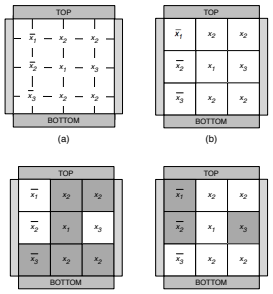

Fig. 1. A four terminal switching network implementing the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$ (a); its corresponding lattice form (b); the lattice evaluated on the assignments 1,1,0 (c) and 0, 0, 1 (d), with grey and white squares representing ON and OFF switches, respectively.

invalidate the completeness of the test set and often correspond to locations of the circuit where area is wasted [11]. For this, synthesis procedures which result in non-redundant circuits are desirable. A node v in C is called *fully testable*, if there does not exist a redundant fault with fault location v. If all nodes in C are fully testable, then C is called *fully testable*.

Finally, we recall that the investigations with respect to the SAFM are usually based on the single fault assumption, i.e., one assumes that there is at most one fault in the circuit.

B. Switching Lattices

A switching lattice is a two-dimensional array of four-terminal switches linked to the four neightbours of a lattice cell, so that these are either all connected (when the switch is ON), or disconnected (when the switch is OFF). A Boolean function can be implemented by a lattice in terms of connectivity across it:

- each four-terminal switch is controlled by a literal;
- each switch may be also labelled with the constant 0, or 1;
- if the literal takes the value 1, the corresponding switch is connected to its four neightbours, else it is not connected;
- the function evaluates to 1 if and only if there exists a connected path between two opposing edges of the lattice, e.g., the top and the bottom edges;
- input assignments that leave the edges unconnected correspond to output 0.

For instance, the 3×3 network of switches in Figure 1 (a) corresponds to the lattice form depicted in Figure 1 (b), which implements the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$. If we assign the values 1, 1, 0 to the variables x_1, x_2, x_3, respectively, we obtain paths of gray square connecting the top and the bottom edges of the lattices (Figure 1 (c)), indeed on this assignment f evaluates to 1. On the contrary, the assignment $x_1 = 0, x_2 = 0, x_3 = 1$, on which f evaluates to 0, does not produce any path from the top to the bottom edge (Figure 1 (d)).

The synthesis problem on a lattice consists in finding an assignment of literals to switches in order to implement a given target function with a lattice of minimal size. The size is measured in terms of the number of switches in the lattice.

A switching lattice can similarly be equipped with left edge to right edge connectivity, so that a single lattice can implement two different functions. This fact is exploited in [5] where the authors propose a synthesis method for switching lattices simultaneously implementing a function f according to the connectivity between the top and the bottom plates, and its dual function[1] f^D according to the connectivity between the left and the right plates. In [13], the authors have proposed a different approach to the synthesis of minimal-sized lattices, which is formulated as a satisfiability problem in quantified Boolean logic and solved by quantified Boolean formula solvers. This method uses the previous algorithm to find an upper bound on the dimensions of the lattice. It then searches for successively better implementations until either an optimal solution is found, or a preset time limit has been exhausted. Experimental results show how this alternative method can decrease lattice sizes considerably. In this approach the use of fixed inputs (i.e., constant values 0 and 1) is allowed.

III. Lattices: definitions and properties

In this section we introduce some definitions and present some properties of switching lattices that will be exploited in Section IV for the analysis of their testability.

Let the first row of a lattice be the *top row*, the last row be the *bottom row*, and any other row be an *internal row*. Two cells in a lattice are *adjacent* if they are in the same column and in two adjacent rows or in the same row and in two adjacent columns. Hereafter, in a lattice we denote *path* any list $l_1, l_2, \ldots, l_{m-1}, l_m$ of literals such that l_i and l_{i+1} (for all $1 \leq i < m$) are contained in adjacent cells and: 1) l_1 is contained in a cell in the top row, 2) l_m is contained in a cell in the bottom row, and 3) all the other literals (i.e., $l_2, \ldots, l_{m-1}$) are contained in cells of the internal rows. Note that paths in lattices may contain more occurrences of the same literal.

Definition 3: A path in a lattice is *unsatisfiable* (resp., *satisfiable*) if contains (resp., does not contain) both a variable x and its complement $\overline{x}$.

Definition 4: The *product associated to a satisfiable path* is the conjunction of all literals of the path, without repetitions. The *product associated to an unsatisfiable path* is 0.

For example, in the lattice in Figure 1 (b) the path x_2, x_1, x_2 is satisfiable and the path $\overline{x}_1, \overline{x}_2, x_1, x_2$ is unsatisfiable. The associated products are x_1x_2 and 0, respectively.

With a slight abuse of notation, we consider the products associated to all paths in a lattice L as implicants of the function f_L implemented by L. Indeed, f_L evaluates to 1 on the set of minterms covered by these products. In this framework, the set of minterms covered by an implicant can be empty: this happens whenever a path is unsatisfiable, as, in this case, the associated product evaluates to 0.

Definition 5: An *accepting path* for a minterm v in a lattice is a satisfiable path whose associated product covers v.

We now introduce the concept of *primality of a path* in a lattice which is strictly related to the concept of prime implicant in a SOP:

[1]The dual of a Boolean function f depending on n binary variables is the function f^D such that $f(x_1, x_2, \ldots, x_n) = \overline{f^D}(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$.

Definition 6: A path $l_1, \ldots, l_i, \ldots, l_m$ in a lattice L is *prime* w.r.t. a literal l_i ($1 \leq i \leq m$), if the product associated to the sequence of literals obtained removing l_i from the path is not an implicant of the function implemented by L.

The primality of a path with respect to a literal l implies that:

(1) the path cannot contain other occurrences of l, since the corresponding product would not change if we remove one occurrence of l from the path, leaving the others;
(2) the path cannot contain pairs $x, \overline{x}$, with $x \neq l$, since the removal of l would leave the associated product unchanged, and equal to 0;
(3) the path might contain cells associated to $\overline{l}$ (in this case the path is unsatisfiable, and becomes satisfiable after the removal of l).

For instance, in the lattice in Figure 1 (b):

- the satisfiable path x_2, x_3, x_2 is prime w.r.t. x_3, as the product x_2 obtained removing x_3 from the path is not an implicant of $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$;
- the satisfiable path x_2, x_3, x_2 is not prime w.r.t. x_2, as the removal of one occurrence of x_2 leaves the associated product x_2x_3 unchanged;
- the unsatisfiable path $x_2, x_1, \overline{x}_2, \overline{x}_3$ is prime w.r.t. x_2, as the product $x_1\overline{x}_2\overline{x}_3$ is not an implicant of f;
- the unsatisfiable path $\overline{x}_1, \overline{x}_2, x_1, x_2$ is never prime, as the removal of any of its literal leaves the associated product unchanged, and equal to 0.

We finally focus on the single cells in a lattice, and we introduce a property that can be associated to the irredundancy of a SOP. Let c be a cell in a switching lattice L that implements a function f_L.

Definition 7: The cell c is *essential* in L if there exists at least a minterm v in the on-set of f_L whose accepting paths always contain c.

For instance, in the lattice in Figure 1 (b) all cells on the leftmost column are essential, as they form the only accepting path for the on-set minterm 000; while the top-left cell c in the lattice in Figure 2 (b) is not essential, since for any on-set minterm of the function implemented by the lattice there exists an accepting path that does not include c.

Observe that setting to the constant value 0 a literal in a cell c is equivalent to removing all paths that include c from the lattice L: indeed, the 0 in c disconnects, i.e., makes unsatisfiable, all paths going through c. If, in addition, the cell c is essential for a minterm v, then all accepting paths for v are removed from L. Thus, the function implemented by the lattice changes, at least on v. In this sense, we can associate the notion of essential cell to that of irredundant product: if we remove an irredundant product from a SOP, the function represented by the expression changes, and if we remove from the lattice L all paths that include an essential cell, the function implemented by L changes. We conclude this section with the following two propositions that characterize how the function implemented by a lattice may change setting one cell to a constant value, i.e., forcing a stuck-at-fault in the cell. Let L be a switching lattice, and let f_L be the function implemented by L. Now, consider the lattice $L^{c\leftarrow 1}$ obtained replacing a literal in a cell c of L with the constant 1.

Proposition 1: The on-set of the function $f_{L^{c\leftarrow 1}}$ implemented by $L^{c\leftarrow 1}$ is a superset of the on-set of f_L, i.e., $f_L^{on} \subseteq f_{L^{c\leftarrow 1}}^{on}$.

Proof. We show that any satisfiable path in L remains satisfiable in $L^{c\leftarrow 1}$. Let $p = l_1, l_2, \ldots, l_{m-1}, l_m$ be a satisfiable path in L. If this path does not include the cell c, than the corresponding path in $L^{c\leftarrow 1}$ is composed by exactly the same literals of p, and it is satisfiable. If p includes the cell c, than the corresponding path in $L^{c\leftarrow 1}$ is obtained replacing one of the literals of p with the constant 1, and it is satisfiable. ■

Consider now the lattice $L^{c\leftarrow 0}$ obtained replacing a literal in a cell c of L with the constant 0. We have:

Proposition 2: The on-set of the function $f_{L^{c\leftarrow 0}}$ implemented by $L^{c\leftarrow 0}$ is a subset of the on-set of f_L, i.e., $f_{L^{c\leftarrow 0}}^{on} \subseteq f_L^{on}$.

Proof. We show that any satisfiable path in $L^{c\leftarrow 0}$ is satisfiable in L. Let $p = l_1, l_2, \ldots, l_{m-1}, l_m$ be a satisfiable path in $L^{c\leftarrow 0}$. The thesis immediately follows since the satisfiable path p cannot include the cell c (that has value 0), thus the corresponding path in L is composed by exactly the same literals of p, and is satisfiable. ■

IV. Testability in the SAFM

In this section we analyze the properties that make a switching lattice fully testable in the SAFM. As we will see, these properties resemble the two properties that guarantee the full testability of the SOP forms in the SAFM: the primality of the products, that ensures the testability of AND gates, and the irredundancy of the cover, that guarantees the testability of the OR gate. In our analysis we will consider stuck-at-faults at the literals that control the four-terminal switches in the lattice. We first introduce the notion of irredundant literal in a lattice.

Definition 8: A literal in a lattice's switch is *0-irredundant* (resp., *1-irredundant*) if it cannot be substituted by the constant 0 (resp., 1) without changing the function computed by the lattice.

For instance, the literal x_1 in the top-left cell of the lattice in Figure 2 (b) is not 0-irredundant, while the literal x_4 in the center-right cell of the lattice in Figure 2 (a) is not 1-irredundant. The notion of irredundancy can be extended to the whole lattice as follows:

Definition 9: A lattice is *0-irredundant* (resp., *1-irredundant*) if any literal contained in it is 0-irredundant (resp., 1-irredundant).

Definition 10: A lattice is *irredundant* if it is 0-irredundant and 1-irredundant.

Observe that 0-irredundant literals guarantee the testability of SA0 faults in the corresponding cell of the lattice, while 1-irredundant literals guarantee the testability of SA1 faults. Indeed, since the function implemented by the lattice changes if we set a literal in a cell c to the constant 1 or to the constant 0, the fault in c can be tested on the minterm on which the function changes. Thus, we have

Proposition 3: An irredundant lattice is fully testable with respect to the SAFM.

The two lattices in Figure 2 (c) and (d) are irredundant, and thus fully testable in the SAFM.

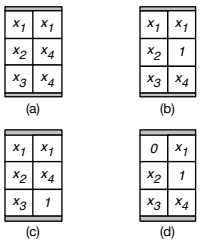

Fig. 2. Four different mininum size switching lattices implementing the function $f = x_1x_2x_3 + x_1x_4$. (a) A 0-irredundant, but not 1-irredundant, lattice; (a) a 1-irredundant, but not 0-irredundant, lattice; (c) a fully testable lattice; (d) a fully testable lattice with a minimum number of literals.

We now investigate the relations between minimality of a switching lattice and full testability in the SAFM. We first consider the minimality with respect to the number of switches, i.e., the area of the lattice. Under the SAFM, it can be shown by counter examples that a lattice of minimum size for a given target function is not in general fully testable. Consider for instance the lattice in Figure 2 (a) that implements the function $f = x_1x_2x_3 + x_1x_4$: this lattice is of minimum size, but it is not fully testable, because it is not 1-irredundant. On the contrary, lattices minimized with respect to the number of literals in the switches are free of redundancies.

Theorem 1: A switching lattice L with a minimum number of literals is fully testable in the SAFM.

Proof. Suppose that L is not fully testable. This means that there exists at least one redundant cell in the lattice whose literal can be replaced by a constant value, 0 or 1, without changing the implemented function. If replace the literal in this cell with a constant value, we get a new lattice for the same function, with a smaller number of literals, in contradiction with the minimality of the number of literals in L. ■

The lattice in Figure 2 (d) contains a minimum number of literals (4) and is therefore fully testable. Note that the minimality of the number of literals is not necessary for the full testability. For example, the lattice in Figure 2 (c) is fully testable, but not minimum with respect to the number of literals, as it contains 5 literals instead of 4.

Finally, we prove in the following theorems that the structural properties of switching lattices that guarantee their full testability in the SAFM are the primality of the paths (for the stuck-at-1 faults) and the essentiality of the cells (for the stuck-at-0 faults).

Theorem 2: A SA1 in a lattice cell c with literal l is testable if and only if there exists a path p that contains the cell c and is prime with respect to l.

Proof. Let L be a switching lattice, and let f_L be the function implemented by L.

(If part). Consider a path p in L that contains the cell c and is prime w.r.t. l. Recall that the primality of p w.r.t. l implies that p cannot contain other occurrences of l, p might contain cells associated to $\bar{l}$, and p cannot contain pairs $x, \overline{x}$, with $x \neq l$. We must show that the SA1 in c can be propagated to the output of the lattice L in order to be tested, i.e., we must prove that the function implemented by the faulty lattice differs from the original function f_L on at least one minterm. This immediately follows from the primality of the path p. Indeed, if we substitute with the constant 1 the unique occurrence of l in p, the resulting product, which cannot be empty because of the properties of prime paths (see Section III), is not an implicant of f_L. Therefore, there exists at least one minterm in the off-set of f_L covered by this new product. On this minterm the faulty lattice computes 1, instead of 0, since the path p, with l replaced by 1, becomes an accepting path.

(Only-if part). Now suppose that the SA1 in the cell c, with literal l, is testable. As we are injecting a constant value 1 into the lattice, we know, by Proposition 1, that the faulty lattice is always correct on the on-set of f_L. Thus, the testability of the SA1 implies the existence of an off-set minterm v of f_L on which the faulty lattice computes 1, while the original lattice computes 0. Since whenever the literal l gets the value 1, correct and faulty lattices are identical and have the same behaviour, l must be 0 on v. Let us now denote with p an accepting path for v in the faulty lattice. p is satisfiable and contains the cell c with the constant 1, while it cannot contain cells with literal l since l is 0 on v. Consider the corresponding path q in the original lattice L. p and q are identical in all cells but c, which is labelled by 1 in p and by l in q. The product of all literals in q is an implicant of f_L, possibly empty if it contains $\bar{l}$. If we remove l, the resulting product is not an implicant since it covers the off-set minterm v. Therefore path q is prime w.r.t. l. ■

Theorem 3: A SA0 in a lattice cell c is testable if and only if the cell c is essential.

Proof. Let L be a switching lattice, and let f_L be the function implemented by L.

(If part). We show that the function implemented by the lattice with a SA0 in the essential cell c differs from f_L on at least one minterm. As we are injecting the value 0 into one cell of L, Proposition 2 implies that the faulty lattice is always correct on the off-set of f_L. Thus, to prove the testability of the SA0, we must show that there is an on-set minterm of f_L on which the faulty lattice computes 0, while the original lattice computes 1. Let l be the literal associated to cell c. Since c is essential, there exists an on-set minterm v for which all accepting paths include c. On all these paths, l must be 1. Now, if we substitute with the constant 0 the literal l in c, we disconnect all paths going through c, and in particular all accepting paths for v. This the faulty lattice computes 0 instead of 1 on v, and the fault can be tested.

(Only-if part). We prove by contraposition that if c is not essential, then a SA0 in c cannot be tested. So, suppose that cell c, with literal l is not essential. By Proposition 2, we know that the faulty lattice, with a SA0 in c, is always correct on the off-set of f_L. Thus, it is enough to show that the faulty lattice is correct also on the on-set of f_L. Let v be a minterm of f_L. Since c is not essential, there exists an accepting path p for v not including c. Thus, the SA0 in c has no effect on p, and the faulty lattice correctly computes 1 on v. ■

V. Algorithms for irredundancy test

In this section we describe a strategy for identifying the *redundant*, i.e., non irredundant, cells in a lattice. Recall that an irredundant lattice is fully testable under the SAFM. Moreover, recall that in our analysis we only consider stuck-at-faults in non-constant cells, i.e., only stuck-at-faults at the literals that control the switches. We first describe a methodology to test whether a given cell in a lattice is 0-irredundant (resp., 1 irredundant). Algorithm 1 shows the strategy based on Theorem 3: a cell c, in the lattice L, is 0-irredundant if and only if c is essential, i.e., there exists at least one miniterm v in the on-set of f_L whose accepting paths always contains c.

Algorithm 1: Algorithm for the testing of the 0-irredundancy of a cell c.

0-irredundant (cell c)
INPUT: A cell c (containing the literal l) in a lattice L
OUTPUT: **true** if c is 0-irredundant in L, **false** otherwise

forall sub-path p_T from a top cell of L to c ($c \notin p_T$)
  if (p_T contains $\overline{l}$) discard p_T;
  if (p_T contains x and $\overline{x}$) discard p_T;
  else
    forall sub-path p_B from c to a bottom cell of L ($c \notin p_B$)
      if (p_B contains $\overline{l}$) discard p_B;
      if ($p_T p_B$ contains x and $\overline{x}$) discard p_B;
      else forall minterm m of the product associated to p_T, l, p_B
        if m is not in the on-set of $L^{c\leftarrow 0}$ **return true** ;
return false;

The algorithm starts from the given cell c (containing the literal l) and considers any sub-path p_T in the lattice from one top cell to c (where c is non included in p_T). If p_T contains $\overline{l}$ the path is discarded since the SA0 in c will not change the output of any minterm computed by L through p_T. Moreover, if p_T contains a variable x and its complement $\overline{x}$ any path containing as a prefix p_T outputs 0. Thus, the output is not affected by the SA0 in c and p_T can be discarded. Any other sub-path p_T is consider in combination with a sub-path p_B from c to a bottom cell of L. Again, we can discard some of the sub-paths p_B in a similar way. In order to check if c is essential, we are left to consider any minterm m of the product p associated to p_T, l, p_B and test if m is in the on-set of $L^{c\leftarrow 0}$. We can omit to test the minterms in the product associated to $p_T, \overline{l}, p_B$ since we have $c = 0$ in $L^{c\leftarrow 0}$. If we find a minterm of p that outputs 0 in $L^{c\leftarrow 0}$, the fault is testable.

Algorithm 2 shows the strategy based on Theorem 2: a cell c (containing the literal l), in the lattice L, is 1-irredundant if and only if there exists a path p that contains the cell c and is prime with respect to l, i.e., the product associated to the sequence of literals obtained removing l from the path is not an implicant of the function implemented by L.

Algorithm 2: Algorithm for the testing of the 1-irredundancy of a cell c.

1-irredundant (cell c)
INPUT: A cell c (containing the literal l) in a lattice L
OUTPUT: **true** if c is 1-irredundant in L, **false** otherwise

forall sub-path p_T from a top cell of L to c ($c \notin p_T$)
  if (p_T contains l) discard p_T;
  if (p_T contains x and $\overline{x}$) discard p_T;
  else
    forall sub-path p_B from c to a bottom cell of L ($c \notin p_B$)
      if (p_B contains l) discard p_B;
      if ($p_T p_B$ contains x and $\overline{x}$) discard p_B;
      else forall minterm m of the product associated to $p_T, \overline{l}, p_B$
        if m is not in the on-set of L **return true** ;
return false;

The first part of the algorithm is similar to Algorithm 1. In order to check if there exists a path p that contains the cell c and is prime with respect to l, we are left to consider any minterm m of the product p associated to $p_T, \overline{l}, p_B$ and test if m is in the on-set of L. It is easy to see that a lattice L can be modeled with an undirected graph $G_L = (V, E)$ where each vertex v in V corresponds to a cell c in L and it is labeled with the literal in c. The edge (v, w) is in E iff the vertices v and w correspond to adjacent cells in L. A vertex corresponding to a cell in the top (resp., bottom) row is a *top* (resp., *bottom*) *vertex* . Both the algorithms consider sub-paths from top cells to c and from c to bottom cells. They can be easily implemented by DFS visits in the graph G_L from c to top (resp., bottom) vertices. The final test in both algorithms can be implemented by OBDD based methods, as classically performed in SOP forms for primality and irredundancy test. More precisely, we need to build the OBDD representing the target function f computed by L, and the OBDD containing all products associated to a satisfiable path through c. The two algorithms have then time complexity polynomial in the size of the OBDDs and the graph G_L. An alternative strategy is a simulation in the given lattice L and in the transformed ones $L^{c\leftarrow 0}$ and $L^{c\leftarrow 1}$. The irredundacy of the overall lattice is simply tested by applying the 0-irredundant and 1-irredundant tests on all the non-constant cells of the lattice.

VI. Experimental results

In this section we discuss the experiments aimed at evaluating the testability of switching lattices synthesized with the recent methods presented in [5], [13]. These experiments are based on the fault injection in lattices by substituting a literal controlling a single cell with a SA1 or a SA0. The fault injection procedure is repeated for each cell of the lattice.

The defect simulations have been run on a machine with two AMD Opteron 4274HE for a total of 16 CPUs at 2.5 GHz and 128 GByte of main memory, running Linux CentOS 7. The benchmarks functions are expressed in PLA form and are taken from a subset of LGSynth93 [19]. A total of about 580 functions were considered, and for each function each output is implemented as a separate Boolean function. The software used for simulations is written in C++. We used ESPRESSO to implement the method described in [5], and a collection of Python scripts for computing minimum-area lattices by transformation to a series of SAT problems, to simulate the results reported in [13].

Table I reports a sample of benchmark functions, referring to lattices synthesized as described in [5] and [13]. The benchmarks synthesized with [13] method were stopped after ten minutes of each SAT execution. The first column in the table reports the name and the number of the considered output of each function. The following columns report, for each synthesis method, the dimension ($r \times s$, and area) of the lattice, and the percentages of 0-redundant and 1-redundant

TABLE I
A SAMPLE OF BENCHMARK FUNCTIONS SYNTHESIZED WITH [5] AND [13] APPROACHES AND THEIR PERCENTAGES OF 0-REDUNDANT AND 1-REDUNDANT CELLS

	[5]				[13]			
name	col × row	area	(R_0 / area)%	(R_1 / area)%	col × row	area	(R_0 / area)%	(R_1 / area)%
addm4 (6)	10×11	110	49%	79%	6×4	24	0%	0%
b11 (3)	3×6	18	22%	56%	3×4	12	8%	8%
b7 (27)	2×5	10	0%	30%	3×3	9	22%	0%
bench (3)	4×6	24	8%	58%	4×3	12	8%	0%
dc2 (1)	7×12	84	40%	62%	6×4	24	4%	13%
ex5 (34)	10×4	40	8%	53%	6×4	24	0%	8%
exps (32)	2×7	14	43%	29%	2×5	10	10%	0%
m3 (3)	5×4	20	10%	55%	5×3	15	7%	7%
m3 (4)	8×6	48	27%	42%	7×3	21	0%	0%
max128 (23)	11×12	132	33%	82%	–	–	–	–
newtag (0)	8×4	32	13%	69%	6×3	18	0%	0%
newxcpla1 (18)	10×7	70	44%	71%	3×7	9	0%	0%
p3 (10)	6×10	60	10%	67%	4×5	20	0%	15%
p82 (13)	5×7	35	29%	34%	3×5	15	0%	0%
rd53 (1)	10×10	100	18%	80%	–	–	–	–
risc (21)	2×5	10	20%	20%	2×4	8	13%	0%
root (1)	8×8	64	36%	73%	6×4	24	8%	8%
sex (4)	3×5	15	40%	27%	3×4	12	17%	17%
tms (0)	4×11	44	32%	41%	3×6	18	0%	0%

TABLE II
OVERALL RESULTS OF THE SIMULATIONS

Synthesis Method	Average area	(R_0/area)%	(R_1/area)%
[5]	30	20%	29%
[13]	15	4.5%	4.5%

cells. Table II describes the overall results for the benchmarks we considered, and it shows the average values for lattice area and percentages of 0-redundant and 1-redundant cells.

We can note that the percentage of cells that are redundant is higher in the [5] synthesis method. This is due to the more constrained structure of the lattices. Indeed, the method proposed in [5] computes a lattice that implements both the target function and its dual, and is in general less compact than the corresponding lattice given by [13].

VII. CONCLUSION

In this paper we have analyzed the testability of switching lattices under the SAFM, and characterized the properties of fully testable lattices. We have also proposed an algorithm to detect redundancies. Future work includes the design of a method to transform non testable lattices into testable ones, by replacing some literals with a constant value, without changing the implemented function.

VIII. ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 691178.

REFERENCES

[1] M. Abramovici and M. Breuer, *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, 1994.

[2] S. B. Akers, "A Rectangular Logic Array," *IEEE Trans. Comput.*, vol. 21, no. 8, pp. 848–857, Aug. 1972.

[3] D. Alexandrescu, M. Altun, L. Anghel, A. Bernasconi, V. Ciriani, L. Frontini, and M. B. Tahoori, "Logic synthesis and testing techniques for switching nano-crossbar arrays," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 54, pp. 14–25, 2017.

[4] M. Altun, V. Ciriani, and M. B. Tahoori, "Computing with nano-crossbar arrays: Logic synthesis and fault tolerance," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, 2017, pp. 278–281.

[5] M. Altun and M. D. Riedel, "Logic Synthesis for Switching Lattices," *IEEE Trans. Computers*, vol. 61, no. 11, pp. 1588–1600, 2012.

[6] A. Bernasconi, V. Ciriani, R. Drechsler, and T. Villa, "Logic Minimization and Testability of 2-SPP Networks," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1190–1202, 2008.

[7] A. Bernasconi, V. Ciriani, G. Trucco, and T. Villa, "Logic Minimization and Testability of 2SPP-P-Circuits," in *Euromicro Conference on Digital Systems Design: Architectures, Methods and Tools (DSD)*, 2009.

[8] A. Bernasconi, V. Ciriani, L. Frontini, V. Liberali, G. Trucco, and T. Villa, "Enhancing logic synthesis of switching lattices by generalized shannon decomposition methods," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 56, pp. 193–203, 2018.

[9] A. Bernasconi, V. Ciriani, L. Frontini, and G. Trucco, "Synthesis on switching lattices of dimension-reducible boolean functions," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2016.

[10] ——, "Composition of Switching Lattices and Autosymmetric Boolean Function Synthesis," in *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, 2017, pp. 137–144.

[11] M. Breuer and A. Friedman, *Diagnosis & Reliable Design of Digital Systems*. Computer Science Press, 1976.

[12] V. Ciriani, A. Bernasconi, and R. Drechsler, "Testability of SPP Three-Level Logic Networks in Static Fault Models," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 12 241–2248, 2006.

[13] G. Gange, H. Søndergaard, and P. J. Stuckey, "Synthesizing Optimal Switching Lattices," *ACM Trans. Design Autom. Electr. Syst.*, vol. 20, no. 1, pp. 6:1–6:14, 2014.

[14] M. C. Morgul and M. Altun, "Synthesis and optimization of switching nanoarrays," in *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 161–164.

[15] O. Tunali and M. Altun, "Permanent and transient fault tolerance for reconfigurable nano-crossbar arrays," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 5, pp. 747–760, 2017.

[16] ——, "A survey of fault-tolerance algorithms for reconfigurable nano-crossbar arrays," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 79:1–79:35, 2017.

[17] ——, "Logic synthesis and defect tolerance for memristive crossbar arrays," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 425–430.

[18] T. Williams and K. Parker, "Design for Testability - A Survey," *IEEE Transactions on Computers*, vol. 31, no. 1, pp. 2–15, 1982.

[19] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," Microelectronic Center, User Guide, 1991.