Testability of Switching Lattices in the Cellular Fault Model

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

Valentina Ciriani
Dipartimento di Informatica
Università degli Studi di Milano, Italy
valentina.ciriani@unimi.it

Luca Frontini
I.N.F.N.
Sezione di Milano, Italy
luca.frontini@mi.infn.it

***Abstract*—A switching lattice is a two-dimensional array of four-terminal switches implemented in its cells. Each switch is linked to the four neighbors and is connected with them when the switch is ON, or is disconnected when the switch is OFF. Recently, with the advent of a variety of emerging nanoscale technologies based on regular arrays of switches, lattices of multi-terminal switches, originally introduced by Akers in 1972, have found a renewed interest. In this paper, the testability under the Cellular Fault Model (CFM) of switching lattices is defined and analyzed. Moreover, some techniques for improving the testability of lattices are discussed and experimentally evaluated.**

***Index Terms*—Switching lattices; testability; cellular fault model; logic synthesis.**

I. Introduction

A switching lattice is a two-dimensional array of four-terminal switches linked to the four neighbors of a lattice cell, so that these are either all connected (when the switch is ON), or disconnected (when the switch is OFF). The first description of lattices for implementing Boolean functions is due to a seminal paper by Akers in 1972 [1]. 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 [2], [3], [4], [5], [7], [8], [10], [11], [12].

Beside synthesis, testability is a major aspect of the design process. For this reason, the testability should be considered from the very beginning. The classical Stuck-at Fault Model (SAFM) is well-known and used throughout the industry for many years for CMOS technology. In the SAFM it is assumed that a defect causes a basic cell input or output to be fixed to either 0 or 1. The testability of lattices in the SAFM has been studied in [6].

The strongest cell-based fault model that controls the correct static behavior of a combinational circuit is the *Cellular Fault Model* (CFM), which tries to completely verify the function computed by each basic cell in the circuit [9]. The investigations with respect to CFM and SAFM are usually based on the single fault assumption, i.e., one assumes that there is at most one fault (according to the considered fault model) in the circuit.

In lattice model we do not deal with gates, but with literals that control switches. Thus, in this paper we define and study a *cellular fault* (CF) in a switching lattice as the event that replaces a controlling literal with a different (faulty) one.

First, we prove that the testability of a general cellular fault is related to the testability of the fault where the correct controlling literal behaves as if it had been inverted. We then discuss how to derive a test set for a general CF starting from the test set for the inverted literal fault. This result allows to simplify the testability analysis.

Finally, out of all possible CFs, we consider the cellular faults that immediately derive from the physical layout of a lattice. To this aim, we denote *adjacent cellular fault* a cellular fault where the faulty literal is the controlling literal of an adjacent switch. For this special case of CFs, we propose some techniques for improving the testability of a lattice without increasing its dimension.

In the experimental results we compare the CF testability of lattices obtained with the two main synthesis methods and evaluate the effect of the proposed lattice restructuring techniques on lattices' testability.

The paper is organized as follows. Preliminaries on switching lattices and on the cellular fault model are reviewed in Section II. The analysis of CF testability is presented in Section III. Section IV describes two methods for improving the testability of adjacent cellular faults in a lattice. Finally, Section V provides the experimental results and Section VI concludes the paper.

II. Preliminaries

A. Switching Lattices

A Boolean function can be implemented by a lattice in the following way:

- each four-terminal switch (or cell of the lattice) is controlled by a literal or a constant value (0 or 1);
- if the literal takes the value 1 (or the cell is labeled by the constant 1), the corresponding switch is connected to its four neighbors; 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 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

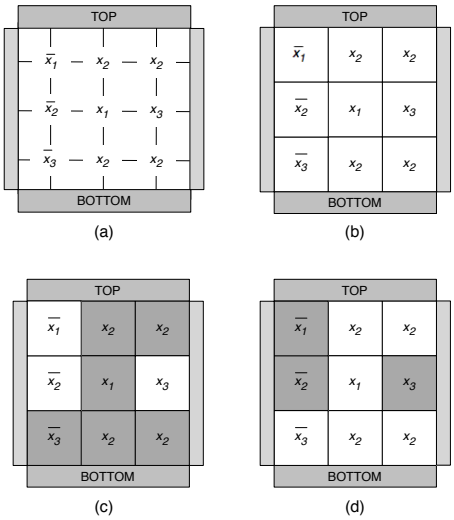

Fig. 1. A four terminal switching network implementing the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$ (a); its corresponding lattice form (b); the lattice evaluated on the assignments 1,1,0 (c) and 0, 0, 1 (d), with grey and white cells representing ON and OFF switches,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

obtain paths of gray (i.e., connected) cells 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 connected 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 [4] 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 f^D according to the connectivity between the left and the right plates. Recall that the dual of a Boolean function f depending on n binary variables is the function f^D such that $f(x_1, x_2, \ldots, x_n) = \overline{f^D}(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$. This method consists of the following steps [4]:

Step 1: find an irredundant, or a minimal, SOP representation for f and f^D: $SOP(f) = p_1 + p_2 + \ldots p_s$ and $SOP(f^D) = q_1 + q_2 + \ldots q_r$;

Step 2: form a $r \times s$ switching lattice and randomly assign each product p_j $(1 \leq j \leq s)$ of $SOP(f)$ to a column and each product q_i $(1 \leq i \leq r)$ of $SOP(f^D)$ to a row;

Step 3: for all $1 \leq i \leq r$ and all $1 \leq j \leq s$, randomly assign to the lattice cell $c_{i,j}$ one literal that is shared by q_i and p_j (the fact that f and f^D are duals guarantees that such a shared literal exists for all i and j).

Taking Step 3 into consideration, let $S_{i,j}$ be the, non-empty, set of literals that are shared by q_i and p_j. We observe that $|S_{i,j}| \geq 1$ and that the algorithm, proposed in [4], randomly assigns a literal in $S_{i,j}$ as the controlling literal of the cell $c_{i,j}$. In Section IV we discuss how to choose the controlling literal in $S_{i,j}$ in order to possibly improve the lattice's testability.

We can observe that the synthesis algorithm, proposed in [4], produces a lattice for f whose size depends on the number of products in the irredundant SOP representations of f and f^D. For instance, the lattice depicted in Figure 1 has been built according to this algorithm, and it implements both the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$ and its dual $f^D = x_1\overline{x}_2x_3 + \overline{x}_1x_2 + x_2\overline{x}_3$.

The time complexity of the algorithm is polynomial in the number of products. However, the method does not always build lattices of minimal size for every target function, since it ties the dimensions of the lattices to the number of products in the SOP forms. In [10], 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. Experimental results show how this alternative method can decrease lattice sizes considerably, at the expense of computational time. In fact, bigger benchmarks cannot be handled by this method. While in the method proposed in [4] the constant labels are not exploited, in this approach the use of fixed inputs (i.e., constant values 0 and 1) is allowed.

We now review some definitions and present some properties of switching lattices that will be exploited for the analysis of their testability [6].

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. A path in a lattice is *unsatisfiable* (resp., *satisfiable*) if contains (resp., does not contain) both a variable x and its complement $\overline{x}$. 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. An *accepting path* for a minterm v in a lattice is a satisfiable path whose associated product covers v. 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. Let c be a cell in a switching lattice L that implements a function f_L. 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 in the lattice.

B. Cellular Fault Model in a Lattice

The cellular fault model CFM [9] for CMOS circuits considers faults that modify the behavior of exactly one gate in a given Boolean circuit. In our model we do not deal with gates, but with controlling literals in a lattice. Thus, we can define a *cellular fault* CF in a switching lattice L as a tuple (c, l_c, l_f), where c is the cell of the lattice L (i.e., the fault location), l_c is the correct controlling literal in c, and $l_f (\neq l_c)$ is the faulty controlling literal.

Out of all the possible faulty literals in a cell c, we can describe a specific cellular fault model that derives from the physical implementation of a lattice. We denote *adjacent cellular fault* a cellular fault where the faulty literal l_f is in an adjacent cell. More precisely:

Definition 1: Let $l_{i,j}$ be the literal in the cell $c_{i,j}$ of a lattice L. We have that:

1) A *Left Adjacent Cellular Fault (L-ACF)* is the cellular fault $(c_{i,j}, l_{i,j}, l_{i,j-1})$,
2) A *Right Adjacent Cellular Fault (R-ACF)* is the cellular fault $(c_{i,j}, l_{i,j}, l_{i,j+1})$,
3) A *Bottom Adjacent Cellular Fault (B-ACF)* is the cellular fault $(c_{i,j}, l_{i,j}, l_{i+1,j})$,
4) A *Top Adjacent Cellular Fault (T-ACF)* is the cellular fault $(c_{i,j}, l_{i,j}, l_{i-1,j})$.

Finally, we denote T^L (resp., T^R, T^B, and T^T), with $1 \leq i \leq r$, $1 \leq j \leq s$ the number of testable cells with a L-ACF (resp., R-ACF, B-ACF, and T-ACF) in the given lattice.

III. Testability in the CFM

In this section we analyze the testability of general cellular faults in a lattice. The special case of adjacent cellular faults will be further discussed in Section IV, together with some techniques for improving their testability.

Consider a CF (c, l_c, l_f) in a cell c with controlling literal l_c and faulty literal l_f. We denote *test set* of CF the set $T_{(l_c \leftarrow l_f)}$ of all input vectors that give an uncorrected output on the faulty lattice. The vectors in $T_{(l_c \leftarrow l_f)}$ are called *test vectors*. A fault is testable if and only if its test set is not empty.

First of all, we observe that the testability of a general cellular fault CF (c, l_c, l_f) is related to the testability of the CF $(c, l_c, \bar{l}_c)$, i.e., the fault where the correct controlling literal l_c behaves as if it had been inverted. Indeed, we prove that if the CF (c, l_c, l_f) is testable, then the fault $(c, l_c, \bar{l}_c)$ is testable as well. On the other hand, if the fault $(c, l_c, \bar{l}_c)$ is testable on a test vector v on which the literals l_c and l_f assume different values, then v can be used as a test for the fault (c, l_c, l_f):

Proposition 1: A CF (c, l_c, l_f) in a lattice cell c with literal l_c is testable if and only if the CF $(c, l_c, \overline{l_c})$ is testable and the test set $T_{(l_c \leftarrow \bar{l}_c)}$ contains at least one input vector where l_f and l_c assume different values.

Proof. *Only-if part.* Let us suppose that the CF (c, l_c, l_f) is testable, and let $v \in T_{(l_c \leftarrow l_f)}$. This means that the function $f_{L'}$ implemented by the faulty lattice L' differs from the function f_L computed by the original lattice L on v. Now, observe that since L and L' have the same behaviour if the literals l_c and l_f assume the same value, the value of l_f on all test vectors in $T_{(l_c \leftarrow l_f)}$ must be different from the value of l_c, and therefore equal to the value of $\bar{l}_c$. This implies that the minterm v can also be used to test the fault $(c, l_c, \bar{l}_c)$, as on this vector the correct and the faulty lattice compute different values. Thus $T_{(l_c \leftarrow \bar{l}_c)}$ is not empty, and the fault $(c, l_c, \bar{l}_c)$ is testable.

If part. Now, suppose that the CF $(c, l_c, \bar{l}_c)$ is testable and that $T_{(l_c \leftarrow \bar{l}_c)}$ contains a vector v where the two literals l_f and l_c assume different values. Thus, since on v the correct and the faulty lattice behave differently, and l_c is different from l_f (i.e., l_f and $\bar{l}_c$ get the same value), we can use v to test the fault where the controlling literal l_c in cell c is replaced by l_f. Thus, $v \in T_{(l_c \leftarrow l_f)}$, and the fault (c, l_c, l_f) is testable. ■

Observe that this proposition implies that any vector in the test set for the fault (c, l_c, l_f) can be used to test the fault $(c, l_c, \bar{l}_c)$, while a test vector v for the CF $(c, l_c, \bar{l}_c)$ is a test vector for the fault (c, l_c, l_f) if and only if l_c and l_f assume different values on v, as stated below.

Remark 1: Let $B_{l_c \neq l_f}$ denote the subset of the space $\{0,1\}^n$ containing all minterms on which the two literals l_c and l_f assume different values. Then $T_{(l_c \leftarrow l_f)} = T_{(l_c \leftarrow \bar{l}_c)} \cap B_{l_c \neq l_f}$.

Due to the relationship between these two kinds of cellular faults, we start the testability analysis with the fault $(c, l_c, \bar{l}_c)$, and then we will discuss how to derive a test set for a general fault CF (c, l_c, l_f) starting from the test set for CF $(c, l_c, \bar{l}_c)$. As we will discuss in Section III-B, this approach allows to simplify the testability analysis.

A. Testability of the CF $(c, l_c, \bar{l}_c)$

Consider a lattice L that implements a function f_L depending on n binary variables. Let L' be the lattice L with the fault $(c, l_c, \bar{l}_c)$ in the cell c, and let p be a path that contains c. We denote with p' the subpath composed by all cells but c (with an abuse of notation, we use the path terminology for p').

First of all, we observe that the fault can only be tested on paths p such that p' is satisfiable:

Proposition 2: A CF $(c, l_c, \bar{l}_c)$ cannot be tested if for each path p through c, the subpath $p' = p \setminus \{c\}$ is unsatisfiable.

Proof. If p' is unsatisfiable, then also the two paths $p = p' \cup \{l_c\}$ and $p_f = p' \cup \{\bar{l}_c\}$ are unsatisfiable. This means that if we change the literal l_c in c with its complement, the path remains unsatisfiable and the behavior of the lattice does not change on that path: the products associated to p and p_f are both 0. If this happens for any path p through c, then the fault cannot be tested. ■

Therefore, we restrict the analysis to paths p such that p' is a satisfiable path, i.e., to paths that without the literal in the faulty cell c are satisfiable. We consider three different cases:

1) p' is satisfiable and contains an occurrence of $\bar{l}_c$;
2) p' is satisfiable and contains an occurrence of l_c;
3) p' is satisfiable and does not contain l_c or $\bar{l}_c$.

In the first case, the fault changes the overall path p from $p = p' \cup \{l_c\}$ to $p_f = p' \cup \{\bar{l}_c\}$. Note that the original path p is unsatisfiable since p' contains $\bar{l}_c$. Thus, the effect of the fault is to change p from an unsatisfiable path to a satisfiable one, as the product associated to the new path p_f coincides with the product associated to p', and p' is satisfiable. In order to test the fault, we should then look for an off-set minterm v of f_L that is covered by the product associated to p', so that on v the faulty lattice L' computes 1 instead of 0. This implies that, with respect to the product p, the fault in c becomes equivalent to a *stuck-at-1* fault (SA1), i.e., a fault where the controlling literal l_c is fixed to the constant value 1. The testability of lattices in the *stuck-at-fault* model (SAFM) has been studied in [6], where it has been proved that a SA1 in a lattice cell c can be tested if and only if there exists a path containing the cell c that is prime with respect to the controlling literal l_c of c. The reason is that if we substitute l_c with 1 in a prime path p, we get an accepting path in the faulty lattice L' for an off-set minterm. Thus, we can apply this result to the CF $(c, l_c, \bar{l}_c)$, and state the following proposition.

Proposition 3: The CF $(c, l_c, \bar{l}_c)$ can be tested on a path $p = p' \cup \{l_c\}$, where p' is satisfiable and contains an occurrence of $\bar{l}_c$ if and only if p is prime with respect to l_c.

Proof. Follows from the above considerations and from Theorem 2 in [6]. ■

Let us now consider the second case, where p' is satisfiable and contains an occurrence of l_c. In this case, the original path $p = p' \cup \{l_c\}$ is satisfiable since the product associated to p is equivalent to the product associated to the satisfiable path p'. Thus, the effect of the fault is to change p from a satisfiable path to an unsatisfiable one, as $p_f = p' \cup \{\bar{l}_c\}$ contains both l_c and $\bar{l}_c$. To test the fault, we should look for an on-set minterm v of f_L such that *(i)* p is an accepting path for v, and *(ii)* the faulty lattice L' computes 0 on v. Observe that in this case, the fact that p becomes unsatisfiable because of the fault is not enough to conclude that the fault is testable. Indeed, each on-set minterm covered by the product associated to p, could have at least another accepting path that does not include the faulty cell c. Again, we have a connection with a stuck-at fault, in particular with the *stuck-at-0* fault (SA0), i.e., a fault where the controlling literal l_c is fixed to the constant value 0. The testability of SA0 faults has been studied in [6], where it has been proved that a SA0 in a cell c can be tested if and only if c is essential. Indeed, the essentiality of c implies that there is at least an on-set minterm whose accepting paths always include c, so that the fault $(c, l_c, \bar{l}_c)$ disconnect all of them.

Applying this result to the CF $(c, l_c, \bar{l}_c)$, we have

Proposition 4: The CF $(c, l_c, \bar{l}_c)$ can be tested on a path $p = p' \cup \{l_c\}$, where p' is satisfiable and contains an occurrence of l_c if and only if c is essential.

Proof. Follows from the above considerations and from Theorem 2 in [6]. ■

In the third and final case, we assume that p' is satisfiable and contains no occurrences of l_c and $\bar{l}_c$. In this case, the testability can be reduced to the testability under both SA0 and SA1 faults. Indeed, the original path $p = p' \cup \{l_c\}$ is now changed to $p_f = p' \cup \{\bar{l}_c\}$, and both paths are satisfiable, but under different input assignments. In particular, the effect of the fault is the following.

- All on-set minterms covered by the product associated to p are such that l_c is equal to 1, and therefore the new path p_f, that contains $\bar{l}_c$, is not an accepting path for them (as in a SA0). In order to test the fault, we need the essentiality of the cell c, as in this case there exists at least a minterm v in the on-set of f_L whose accepting paths always contain c, and on v the faulty lattice L' computes 0 instead of 1.
- The new path p_f becomes an accepting path for all off-set minterms, if any, covered by the product associated to p' and such that l_c is equal to 0 (as in a SA1). In order to test the fault, we now need the primality of p with respect to l_c, that implies that there exists an off-set minterm v on which the faulty lattice L' computes 1 instead of 0.

We can therefore state the following proposition.

Proposition 5: The CF $(c, l_c, \bar{l}_c)$ can be tested on a path $p = p' \cup \{l_c\}$, where p' is satisfiable and does not contain l_c or $\bar{l}_c$ if and only if p is prime with respect to l_c or the cell c is essential.

Proof. Follows from the above considerations and from Theorems 2 and 3 in [6]. ■

B. Testability of the CF (c, l_c, l_f)

We now discuss the testability of the general fault CF (c, l_c, l_f). Observe that, for any lattice cell c, there are $2n-1$ possible cellular faults, where n is the number of input variables. In fact, the faulty literal l_f can be any literal x, $\overline{x}$, but l_c. Fortunately, all these different faults can be seen as particular instances of the fault CF $(c, l_c, \bar{l}_c)$, and this fact allows us to simplify the testability analysis, as proved in the following theorem, summarizing all previous results.

Theorem 1: For any literal l_f different form l_c, the CF (c, l_c, l_f) in a lattice cell c with controlling literal l_c is testable if and only if $T_{(l_c \leftarrow \bar{l}_c)} \cap B_{l_c \neq l_f} \neq \emptyset$.

Proof. Follows from Proposition 1 and Remark 1. ■

Therefore, once the test set for the fault $(c, l_c, \bar{l}_c)$ has been computed, we can use it to derive, by a simple set intersection, the test sets of all the other $2n-2$ cellular faults (c, l_c, l_f), where l_f can be any literal but l_c and $\bar{l}_c$.

IV. Improving the Testability in the ACFM

In Section III, we discuss the testability properties of lattices under the CF model. In this section, we concentrate on improving the testability of Adjacent Cellular Faults for the lattices synthesized by the Altun-Riedel method [4] described in Section II-A. First of all, we note that the considered minimization algorithm defines many equivalent lattices for the given function f, all of dimension $r \times s$. These lattices may exhibit a different ACF sensitivity for a single fault. In particular, the controlling literal in the lattice cell $c_{i,j}$ is selected choosing arbitrarily in the corresponding set $S_{i,j}$ (see Step 3 of the algorithm). Consider, for example, the lattice for $f = x_1x_2 + x_1x_3 + x_2x_3$ depicted in Figure 2, where

in each cell $c_{i,j}$ it is represented the corresponding set $S_{i,j}$. Starting from this lattice we can build 8 different equivalent lattices by choosing the controlling literal in the diagonal cells. Instead of selecting arbitrarily the controlling literal, we will exploit the degrees of freedom offered by the Altun-Riedel method to detect the most testable one in the ACF model (see Definition 1).

We first discuss a lower bound on the number of cells that are not testable in a lattice. Consider the cell $c_{i,j}$ and its controlling literal $l_{i,j} \in S_{i,j}$. Suppose that $l'_{i,j} \in S_{i,j}$, $l'_{i,j} \neq l_{i,j}$, is another possible controlling literal for $c_{i,j}$. In presence of the CF $(c_{i,j}, l_{i,j}, l'_{i,j})$ we cannot test the fault since the two lattice are equivalent. Therefore, in a lattice the number of non testable faults is lower-bounded by the total number of these faults. In particular, consider the ACF model. Let $c_{i,j}$ be a cell of the given lattice, and $l_{i,j} \in S_{i,j}$ be the controlling literal. For example, if the controlling literal $l_{i,j-1}$ of the left adjacent cell $c_{i,j-1}$ is contained in $S_{i,j}$ then we cannot test the L-ACF. More precisely, the number of non-testable ACFs for the cell $c_{i,j}$ ($1 \leq i \leq r$ and $1 \leq j \leq s$) is lower-bounded by $N_{i,j} = n_{i-1,j} + n_{i+1,j} + n_{i,j-1} + n_{i,j+1}$, where $n_{h,k} = 1$ iff $l_{h,k} \in S_{i,j}$ (we consider $n_{h,k} = 0$ if $h < 1$ or $k < 1$ or $h > r$ or $k > s$). Considering this observation we can give a heuristic algorithm for choosing the controlling literal in $S_{i,j}$ for the cell $c_{i,j}$. This heuristic tries to avoid to choice of controlling literals occurring in the sets of the adjacent cells. Thus, Step 3 of the Altun-Riedel algorithm can be replaced by the following strategy.

Algorithm 1: Heuristic for choosing the controlling literal in each cell of a lattice L .

ControllingLiterals (lattice L)
INPUT: a lattice L ($r \times s$) and, for each cell $c_{i,j}$, the set $S_{i,j}$ of its possible controlling literals
OUTPUT: a lattice L' where each cell $c'_{i,j}$ contains exactly one controlling literal $l'_{i,j}$

for $i = 1$ **to** $r - 1$
  for $j = 1$ **to** $s - 1$
    $S = S_{i,j} \setminus (S_{i+1,j} \cup S_{i,j+1})$
    if ($S \neq \emptyset$) choose randomly $l'_{i,j} \in S$;
    else
      $S = S_{i,j} \setminus S_{i+1,j}$
      if ($S \neq \emptyset$) choose randomly $l'_{i,j} \in S$;
      else
        $S = S_{i,j} \setminus S_{i,j+1}$
        if ($S \neq \emptyset$) choose randomly $l'_{i,j} \in S$;
        else choose randomly $l'_{i,j} \in S_{i,j}$;
for $i = 1$ **to** $r - 1$ // last column
  $S = S_{i,s} \setminus S_{i+1,s}$
  if ($S \neq \emptyset$) choose randomly $l'_{i,s} \in S$;
  else choose randomly $l'_{i,s} \in S_{i,s}$;
for $j = 1$ **to** $s - 1$ // last row
  $S = S_{r,j} \setminus S_{r,j+1}$
  if ($S \neq \emptyset$) choose randomly $l'_{r,j} \in S$;
  else choose randomly $l'_{r,j} \in S_{r,j}$;
choose randomly $l'_{r,s} \in S_{r,s}$;

$\{x_1, x_2\}$	$\{x_1\}$	$\{x_2\}$
$\{x_1\}$	$\{x_1, x_3\}$	$\{x_3\}$
$\{x_2\}$	$\{x_3\}$	$\{x_2, x_3\}$

Fig. 2. A lattice for the function $f = x_1x_2 + x_1x_3 + x_2x_3$, where each cell $c_{i,j}$ contains the corresponding set $S_{i,j}$.

A second observation on Altun-Riedel algorithm is that in the Step 2 of the procedure (see Section II-A) each product of the SOP for f is assigned to a column, and each product of the SOP for the dual f^D is assigned to a row, without any specific rule for these assignments. As a consequence, any permutation of the products in $SOP(f)$ and in $SOP(f^D)$ gives rise to a correct, and possibly different, lattice for f. We are then allowed to permute columns and rows in order to minimize the number of adjacent cells containing the same literal. In fact, we can easily observe that if two adjacent cells contain exactly the same literal, the corresponding ACF cannot be tested.

In summary, we propose a new version of Altun-Riedel algorithm in order to avoid some possible non-testable ACFs. The overall strategy, which we implement and show in Section V, exploits Algorithm 1 for Step 3 of the method [4], and inserts a new last Step 4 that performs column and row permutations in order to decrease the number of adjacent cells containing the same literal. Thus, the new synthesis algorithm that improves ACF testability is:

- **Step 1:** find an irredundant, or a minimal, SOP representation for f and f^D: $SOP(f) = p_1 + p_2 + \ldots p_s$ and $SOP(f^D) = q_1 + q_2 + \ldots q_r$;
- **Step 2:** form a $r \times s$ switching lattice and assign each product p_j ($1 \leq j \leq s$) of $SOP(f)$ to a column and each product q_i ($1 \leq i \leq r$) of $SOP(f^D)$ to a row;
- **Step 3:** for all $1 \leq i \leq r$ and all $1 \leq j \leq s$, assign to the switch on the lattice site (i, j) one literal that is shared by q_i and p_j following the strategy described in Algorithm 1;
- **Step 4:** permute rows and columns in order to minimize the number of adjacent cells containing the same literal.

Note that the proposed approach possibly improves the testability without increasing the lattice's area.

V. Experimental results

The aim of this section is twofold. In the first part we compare the testability of ACFs for lattices obtained with [4] and [10] synthesis methods. In the second part we evaluate the effect of the lattice restructuring methods proposed in Section IV on the testability of lattices obtained with [4].

The experiments 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 benchmark functions (expressed in PLA form) are taken from

TABLE I
A SAMPLE OF BENCHMARKS SYNTHESIZED WITH THE METHODS [4] AND [10], AND THEIR PERCENTAGES OF TESTABLE ACFS

		[4]							[10]						
name	n	r	s	area	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$	r	s	area	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$
add6(1)	4	6	6	36	53%	42%	67%	42%	5	3	15	**87%**	**93%**	**100%**	**100%**
addm4(6)	5	10	11	110	55%	47%	24%	24%	6	4	24	**100%**	**100%**	**96%**	**100%**
adr4(3)	4	6	6	36	69%	72%	59%	67%	5	3	15	**93%**	**100%**	**100%**	**100%**
al2(33)	6	2	5	10	60%	60%	**100%**	**100%**	2	4	8	**100%**	**100%**	**100%**	**100%**
al2(36)	6	2	5	10	60%	60%	**100%**	**100%**	2	4	8	**100%**	**100%**	**100%**	**100%**
amd(7)	6	6	7	42	57%	50%	48%	48%	5	4	20	**100%**	**100%**	**100%**	**95%**
b11(8)	6	2	5	10	70%	70%	90%	80%	2	4	8	**87%**	**87%**	**100%**	**100%**
b7(4)	6	3	5	15	60%	66%	**100%**	**100%**	3	4	12	**100%**	**100%**	**100%**	**100%**
bench(7)	6	4	6	24	**100%**	**100%**	**100%**	**100%**	3	5	15	**100%**	**100%**	**100%**	**100%**
ex5(35)	6	7	3	21	**90%**	86%	57%	57%	6	3	18	89%	**89%**	**72%**	**67%**
exp(13)	6	2	5	10	90%	**100%**	**100%**	**100%**	2	4	8	**100%**	**100%**	**100%**	**100%**
fout(1)	6	9	10	4	**100%**	**100%**	**100%**	**100%**	6	4	24	87%	92%	**100%**	**100%**
fout(2)	6	7	9	63	**100%**	**100%**	**100%**	**100%**	6	3	18	**100%**	94%	**100%**	**100%**
fout(4)	6	9	8	72	**100%**	**100%**	**100%**	**100%**	6	4	24	**100%**	92%	96%	96%
fout(5)	6	7	6	42	**100%**	**100%**	**100%**	**100%**	5	3	5	80%	80%	**100%**	**100%**
fout(6)	6	8	9	72	**100%**	**100%**	**100%**	**100%**	5	4	20	95%	95%	95%	95%
fout(7)	6	8	10	80	64%	64%	27%	40%	6	4	24	**92%**	**92%**	**96%**	**100%**
fout(8)	6	9	10	90	61%	56%	27%	33%	6	4	24	**96%**	**92%**	**87%**	**96%**
inc(0)	6	6	7	42	55%	52%	45%	45%	4	4	16	**100%**	**100%**	**94%**	**100%**
lin.rom(21)	6	5	5	25	64%	63%	72%	72%	4	4	16	**81%**	**81%**	**100%**	**100%**
luc(16)	6	6	8	48	50%	52%	52%	46%	4	5	20	**100%**	**100%**	**90%**	**90%**
luc(6)	6	4	7	28	75%	63%	64%	50%	3	4	12	**100%**	**100%**	**100%**	**100%**
mish(1)	6	5	6	30	60%	67%	67%	63%	5	3	15	**100%**	**100%**	**100%**	**100%**
p82(6)	5	6	5	30	63%	57%	80%	63%	3	5	15	**100%**	**100%**	**87%**	**87%**
pope.rom(0)	6	7	7	49	**94%**	**94%**	33%	23%	6	3	18	**94%**	**94%**	**83%**	**72%**
risc(21)	5	2	5	10	80%	80%	90%	80%	2	4	8	**100%**	**100%**	**100%**	**100%**
sqr6(3)	6	9	9	81	51%	68%	42%	43%	6	3	18	**94%**	**94%**	**100%**	**100%**
t2(10)	5	4	5	20	90%	**85%**	**100%**	**100%**	3	4	12	**92%**	83%	92%	75%
Z5xp1(5)	5	10	10	100	30%	29%	23%	28%	4	5	20	**100%**	**100%**	**100%**	**95%**

TABLE II
OVERALL RESULTS OF THE COMPARISON BETWEEN [4] AND [10]

Synthesis Method	Average area	(T^R/area)%	(T^L/area)%	(T^T/area)%	(T^B/area)%
[10]	12	95.6%	95.7%	95.8%	95.4%
[4]	27	69.1%	67.9%	68.1%	69.2%

LGSynth93 [13]. Each output of a benchmark is implemented as a separate Boolean function for a total of 520 functions.

Since the simulation time depends on the number of variables, we consider lattices with a number of variables lower than 6. Notice that this limitation is due to the onerous procedure for the fault simulation, and it is not due to our proposed algorithms for improving testability.

The software used for simulations is written in C++. We used ESPRESSO to implement the method described in [4], 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 [10].

A. Testability of lattices synthesized with methods [4] and [10]

In Table I we compare the number of testable cells for each ACF between lattices synthesized as described in [4] and [10]. The first column in the table reports the name and the number of the considered output of each function; the second column reports the number of input variables. The following columns report, for each synthesis method, the dimensions (r, s and area) and the percentage of testable cells (T^R, T^L, T^T, T^B) w.r.t. the lattice area. From our experiments, it turns out that for more than 70% of the benchmarks, the lattice synthesized with [10] contains a higher percentage of testable cells than the one obtained with [4]. Table II shows the average values of lattice areas and percentages of testable cells for any ACF, for the two synthesis methods. This analysis is done only for lattices where the algorithm proposed in [10] produces a lattice different from [4]; for this reason the percentages of testable ACF w.r.t. the lattice area are different from the percentages of Table V.

B. Improving the testability of lattices synthesized with the method [4]

In a second set of experiments, we are interested in evaluating the efficacy of the two methods proposed in Section IV to improve the ACF testability.

First, we consider the method concerning the choice of the controlling literals described in Algorithm 1. Thus, in Table III, we compare lattices synthesized using [4] with different literal choices: arbitrary and using the proposed algorithm. In particular the first column contains the name and the output number of each function, the second column reports the number of input variables. The columns from three to five contain the dimensions and the area of the lattice; the columns from six to nine report the percentage of testable ACFs w.r.t. the lattice area of a lattice where the literals are chosen arbitrary; columns from ten to thirteen report the

TABLE III
A SAMPLE OF BENCHMARKS SYNTHESIZED WITH THE METHOD [4] AND THEIR PERCENTAGES OF TESTABLE ACFS WITH AN ARBITRARY LITERAL CHOICE AND WITH THE CHOICE PROPOSED BY ALGORITHM 1

					Arbitrary				Proposed Algorithm			
name	n	col	row	area	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$
add6(2)	6	16	16	256	**20%**	20%	20%	21%	**20%**	**22%**	**24%**	**23%**
addm4(6)	5	10	11	110	55%	47%	24%	24%	**62%**	**50%**	**25%**	**25%**
al2(20)	4	1	4	4	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
amd(7)	6	6	7	42	**57%**	50%	48%	48%	55%	**52%**	**55%**	**55%**
b11(26)	5	2	4	8	**62%**	**62%**	**100%**	**100%**	**62%**	**62%**	**100%**	**100%**
b12(0)	6	4	6	24	50%	37%	**58%**	**75%**	**62%**	**54%**	46%	68%
exp(13)	6	2	5	10	**90%**	**100%**	**100%**	**100%**	**90%**	**100%**	**100%**	**100%**
f51m(2)	6	14	14	196	24%	24%	**24%**	**27%**	**31%**	**30%**	**24%**	**27%**
fout(0)	6	9	12	108	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
fout(3)	6	10	12	120	39%	35%	**34%**	**42%**	**52%**	**47%**	30%	30%
fout(8)	6	9	10	90	**61%**	56%	37%	33%	**61%**	**58%**	**39%**	**38%**
in4(14)	6	3	4	12	**50%**	**50%**	**100%**	**100%**	**50%**	**50%**	**100%**	**100%**
in6(3)	5	2	4	8	**100%**	**100%**	**62%**	**62%**	**100%**	**100%**	**62%**	**62%**
inc(0)	6	6	7	42	55%	52%	45%	**45%**	**60%**	**64%**	**48%**	43%
jbp(32)	5	2	4	8	**100%**	**100%**	**62%**	**62%**	**100%**	**100%**	**62%**	**62%**
log8mod(1)	6	6	9	54	50%	70%	54%	46%	**54%**	**72%**	**57%**	**50%**
luc(17)	6	9	10	90	38%	43%	31%	38%	**48%**	**51%**	**33%**	**40%**
m1(3)	6	3	4	12	**83%**	**83%**	**67%**	**50%**	**83%**	**83%**	**67%**	**50%**
m4(8)	6	1	6	6	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
m181(0)	6	4	6	24	50%	37%	**58%**	**75%**	**62%**	**54%**	46%	58%
mish(1)	6	5	6	30	60%	67%	67%	63%	**63%**	**70%**	**63%**	**70%**
mish(22)	6	2	5	10	**100%**	**100%**	**60%**	**60%**	**100%**	**100%**	**60%**	**60%**
misj(3)	4	3	2	6	**83%**	**83%**	**83%**	**83%**	**83%**	**83%**	**83%**	**83%**
mlp4(5)	6	9	10	90	41%	42%	17%	19%	**44%**	**47%**	**22%**	**28%**
newxcpla1(15)	6	6	1	6	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
p1(9)	6	7	8	56	59%	59%	**36%**	**41%**	**61%**	**61%**	30%	39%
p82(8)	5	5	7	35	60%	60%	49%	49%	**66%**	**69%**	**51%**	**51%**
p82(10)	5	2	5	10	**70%**	**70%**	**100%**	**100%**	**70%**	**70%**	**100%**	**100%**
pope.rom(8)	6	13	10	130	44%	41%	**32%**	**35%**	**52%**	**53%**	27%	26%
pope.rom(14)	6	10	7	70	**67%**	**56%**	41%	39%	**67%**	**56%**	**47%**	**44%**
pope.rom(16)	6	1	6	6	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
radd(2)	6	16	16	256	20%	**20%**	20%	21%	20%	**22%**	**24%**	**23%**
rd53(1)	5	10	10	100	70%	66%	**26%**	**36%**	**73%**	**69%**	16%	16%
shift(1)	5	4	4	16	62%	**44%**	**44%**	**62%**	**62%**	**44%**	**44%**	**62%**

percentage of testable ACFs w.r.t. the lattice area of a lattice where the literals are chosen using Algorithm 1.

The second method, based on row and column permutations, is evaluated in Table IV. Recall that the goal of the permutations is to minimize the number of adjacent cells that contain the same literal. To compute the best permutation of rows and columns we use the linear optimizer GLPK (GNU Linear Programming Kit)

Finally, Table V summarizes the experimental results showing the percentages of testable ACFs where the literal choice and the column and row permutation are arbitrary, with the percentages obtained with two methods proposed in Section IV. The first column shows the used approach, the following columns show the percentage of testable ACFs w.r.t. the lattice area, the percentage of lattices that have a higher testability w.r.t. the arbitrary approach, and the percentage of cells whose testability is improved for each type of ACF.

The method based on permutations is clearly more effective, guaranteeing a good improvement of ACF testability. This is probably due to the fact that only a small number of cells have a literal set with cardinality strictly greater than 1, so that the effect of the first method is limited.

VI. CONCLUSION

In this paper we have extended the notion of cellular faults to switching lattices, and we have proved that the testability of a general cellular fault is related to the testability of the inverted literal fault. We have exploited this result for simplifying the testability analysis of general CFs. We have then considered the adjacent cellular faults, for which we have proposed some techniques for improving the testability of a lattice without increasing its dimension. The experimental results validated the proposed approach.

Future work includes the study of different fault models for lattices. Moreover it would be interesting to design techniques to improve testability of lattices synthesized with the method proposed in [10].

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

TABLE IV

A SAMPLE OF BENCHMARKS SYNTHESIZED WITH THE METHOD [4] AND THE PERCENTAGES OF TESTABLE ACFS BEFORE AND AFTER THE COLUMN AND ROW PERMUTATIONS

					ordered				randomly chosen			
name	Col	Row	Area	n	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$	$\%T^R$	$\%T^L$	$\%T^T$	$\%T^B$
add6(1)	6	6	36	4	**69%**	**72%**	**42%**	**47%**	53%	42%	33%	42%
al2(9)	2	5	10	6	**60%**	**60%**	**100%**	**100%**	**60%**	**60%**	**100%**	**100%**
alcom(2)	2	4	8	5	**62%**	**62%**	**100%**	**100%**	**62%**	**62%**	**100%**	**100%**
alcom(3)	4	4	16	6	81%	**87%**	**62%**	**87%**	**87%**	81%	37%	62%
alcom(5)	4	3	12	5	**75%**	**75%**	**100%**	**100%**	**75%**	**75%**	83%	67%
alcom(34)	1	4	4	4	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
alu2(0)	5	4	20	4	**90%**	**85%**	**50%**	**55%**	70%	65%	**50%**	40%
alu2(1)	8	7	56	6	**95%**	**87%**	**32%**	**32%**	59%	59%	30%	18%
b12(0)	4	6	24	6	**68%**	**62%**	**58%**	**75%**	50%	37%	**58%**	**75%**
bench(2)	6	7	42	6	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
bench(3)	4	6	24	6	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
bench(6)	4	8	32	5	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
dc1(2)	4	4	16	4	**75%**	**75%**	**94%**	69%	62%	56%	56%	**81%**
dc1(4)	4	5	20	4	70%	**75%**	**60%**	**70%**	**75%**	65%	**60%**	45%
dc1(5)	4	4	16	4	**75%**	**81%**	**87%**	**62%**	62%	56%	44%	56%
dekoder(3)	5	3	15	4	**93%**	**87%**	**60%**	67%	87%	80%	**60%**	**73%**
dekoder(5)	4	3	12	4	**92%**	**92%**	**50%**	**58%**	83%	**92%**	42%	**58%**
exp(0)	3	5	15	6	**60%**	**60%**	**100%**	**100%**	**60%**	**60%**	**100%**	**100%**
inc(1)	6	7	42	6	**79%**	**81%**	**74%**	**74%**	40%	50%	48%	48%
inc(4)	3	3	9	4	**78%**	**78%**	**100%**	**100%**	**78%**	**78%**	**100%**	**100%**
inc(8)	2	3	6	4	**67%**	**67%**	**100%**	**100%**	**67%**	**67%**	**100%**	**100%**
luc(1)	1	5	5	5	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
luc(6)	4	7	28	6	71%	**86%**	**64%**	**64%**	**75%**	64%	**64%**	50%
luc(17)	9	10	90	6	**84%**	**84%**	**46%**	**43%**	38%	43%	31%	38%
luc(18)	8	8	64	6	**72%**	**73%**	**55%**	**64%**	47%	39%	53%	58%
luc(22)	5	8	40	6	**77%**	**70%**	47%	**55%**	52%	62%	**50%**	45%
m2(1)	2	2	4	3	**75%**	**75%**	**100%**	**100%**	**75%**	**75%**	**100%**	**100%**
m4(2)	6	2	12	6	94%	**100%**	**100%**	**100%**	**100%**	92%	**100%**	**100%**
misg(2)	3	2	6	4	**100%**	**100%**	**67%**	**67%**	**100%**	**100%**	**67%**	**67%**
mish(1)	5	6	30	6	**77%**	**80%**	**70%**	**67%**	60%	67%	67%	63%
newcwp(0)	5	4	20	4	**75%**	**80%**	**80%**	**80%**	60%	65%	60%	65%
newcwp(1)	4	4	16	3	**81%**	**81%**	**81%**	**81%**	75%	75%	69%	69%
p82(4)	3	6	18	5	**83%**	78%	**67%**	**56%**	78%	**83%**	56%	**56%**
radd(1)	6	6	36	4	**69%**	**72%**	**42%**	**47%**	53%	42%	33%	42%

TABLE V

SUMMARY OF THE TESTABILITY RESULTS FOR LATTICES SYNTHESIZED WITH [4] AND LATTICES OBTAINED WITH THE TWO OPTIMIZATION METHODS

	R-ACF			L-ACF			T-ACF			B-ACF		
	(T^R/area)%	% of improved lattices	% of increase of T^R	(T^L/area)%	% of improved lattices	% of increase of T^L	(T^T/area)%	%of improved lattices	% of increase of T^T	(T^B/area)%	% of improved lattices	% of increase of T^B
[4]	83.9%	–	–	83.5%	–	–	85.5%	–	–	85.5%	–	–
[4] with Algorithm 1	84.6%	12%	16%	84.2%	12%	15%	85.8%	9%	6%	85.9%	8%	3%
[4] with permutations	88%	22%	52%	88%	23%	54%	89%	18%	40%	90%	21%	40%

[2] 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.

[3] 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.

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

[5] L. Anghel, A. Bernasconi, V. Ciriani, L. Frontini, G. Trucco, and I. I. Vatajelu, "Fault mitigation of switching lattices under the stuck-at-fault model," in *IEEE Latin American Test Symposium, LATS 2019, Santiago, Chile, March 11-13, 2019*, 2019, pp. 1–6.

[6] A. Bernasconi, V. Ciriani, and L. Frontini, "Testability of switching lattices in the stuck at fault model," in *IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018*, 2018, pp. 213–218.

[7] 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.

[8] A. Bernasconi, V. Ciriani, L. Frontini, and G. Trucco, "Composition of switching lattices for regular and for decomposed functions," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 60, pp. 207–218, 2018.

[9] A. Friedman, "Easily testable iterative systems," *IEEE Transactions on Computers*, vol. C-22, pp. 1061–1064, 1973.

[10] 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.

[11] M. C. Morgül and M. Altun, "Optimal and heuristic algorithms to synthesize lattices of four-terminal switches," *Integration*, vol. 64, pp. 60–70, 2019.

[12] F. Peker and M. Altun, "A fast hill climbing algorithm for defect and variation tolerant logic mapping of nano-crossbar arrays," *IEEE Trans. Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 522–532, 2018.

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