

March Test Generation Revealed

*Original*

March Test Generation Revealed / Benso, Alfredo; Bosio, Alberto; DI CARLO, Stefano; DI NATALE, Giorgio; Prinetto, Paolo Ernesto. - In: IEEE TRANSACTIONS ON COMPUTERS. - ISSN 0018-9340. - STAMPA. - 57:12(2008), pp. 1704-1713. [10.1109/TC.2008.105]

*Availability:*

This version is available at: 11583/1845176 since:

*Publisher:*

IEEE Computer Society

*Published*

DOI:10.1109/TC.2008.105

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

(Article begins on next page)

# March Test Generation Revealed

Alfredo Benso, *Senior Member, IEEE*, Alberto Bosio, *Member, IEEE*, Stefano Di Carlo, *Member, IEEE*, Giorgio Di Natale, *Member, IEEE*, and Paolo Prinetto, *Member, IEEE*

**Abstract**—Memory testing commonly faces two issues: the characterization of detailed and realistic fault models and the definition of time-efficient test algorithms. Among the different types of algorithms proposed for testing Static Random Access Memories, march tests have proven to be faster, simpler, and regularly structured. The majority of the published march tests have been manually generated. Unfortunately, the continuous evolution of the memory technology introduces new classes of faults such as dynamic and linked faults and makes the task of handwriting test algorithms harder and not always leading to optimal results. Although some researchers published handmade march tests able to deal with new fault models, the problem of a comprehensive methodology to automatically generate march tests addressing both classic and new fault models is still an open issue. This paper proposes a new polynomial algorithm to automatically generate march tests. The formal model adopted to represent memory faults allows the definition of a general methodology to deal with static, dynamic, and linked faults. Experimental results show that the new automatically generated march tests reduce the test complexity and, therefore, the test time, compared to the well-known state of the art in memory testing.

**Index Terms**—Automatic test generation, memory test, memory fault modeling, march tests.

## 1 INTRODUCTION

MEMORIES are the predominant majority in semiconductor device production, with also the fastest growing technology [1]. The complex nature of their internal behavior and the very high density of their cell arrays make memories extremely vulnerable to physical defects. The challenge of memory testing stems from the difficulty of defining realistic fault models and designing time-efficient test algorithms [2].

In the last years, the so-called static faults (i.e., faults sensitized by the execution of just a single memory operation) [3] have been the predominant class of fault models addressed by researchers and the industry. Unfortunately, the latest technologies show new faulty behaviors like dynamic [4] and linked faults [5] that need to be considered as well.

March tests [2] are an efficient class of memory tests with low time complexity and high fault coverage. Several hand-designed and automatically generated march tests have been proposed in the literature.

One of the first march test generation algorithms is presented in [6]. It is based on the notion of a transition tree where each path from the root to a leaf corresponds to a certain march test able to cover a target set of faults. The main drawback of this approach is that the transition tree is

unbounded and the search process is of exponential complexity in general. Several improvements to this technique have been proposed. Zarrineh et al. [7] restricts the search process to the parts of the tree where a solution exists using the notion of primitive march tests, whereas in [8], a branch-and-bound approach and a fault-collapsing procedure are used to limit the search space. Al-Harbi and Gupta [9] applies the methodology presented in [8] to generate march tests detecting linked faults. The generation process is not detailed, and only one march test is generated.

All these solutions consider a limited set of static fault models, and the extension to new faults is complex. Niggemeyer and Rudnick [10] presents a generation algorithm for the test and diagnosis of memory faults based on a fault description able to model the complete set of single-cell and two-cell static faults. It suffers from the same problems as [6] since it is still based on transition trees, but it theoretically allows covering all possible static faults, even if only stuck-at faults (SAFs), transition faults (TFs), and idempotent and inversion coupling faults (CFid and CFinv) are considered in the paper.

Wu et al. [11] presents a completely different approach, named Test Algorithm Generation by Simulation (TAGS). Several known march tests of different lengths are generated, and their fault coverage is evaluated using the RAMSES fault simulator [12]. The approach allows high flexibility in terms of fault models, but its complexity is still exponential since it requires an exhaustive search. The authors also propose heuristics to overcome the complexity issue, but in this case, they cannot guarantee the optimality of the results.

In [13], we presented a generation algorithm based on a Test Pattern Graph (TPG) to model static memory faults. The generation problem is thus a search problem on the graph. The main contribution of [13] is the extended set of addressed functional faults. Its main drawback is the computational complexity that reduces the number of total faults that can be included in the target fault list.

- A. Benso, S. Di Carlo, and P. Prinetto are with the Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy.  
E-mail: {alfredo.benso, stefano.dicarlo, paolo.prinetto}@polito.it.
- A. Bosio and G. Di Natale are with the Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier II/CNRS, 161, rue Ada, 34392 Montpellier Cedex 5, France.  
E-mail: {alberto.bosio, giorgio.dinatale}@lirmm.fr.

Manuscript received 20 Feb. 2007; revised 11 Dec. 2007; accepted 24 Apr. 2008; published online 10 July 2008.

Recommended for acceptance by C. Metra.

For information on obtaining reprints of this article, please send e-mail to: [tc@computer.org](mailto:tc@computer.org), and reference IEEECS Log Number TC-0064-0207.  
Digital Object Identifier no. 10.1109/TC.2008.105.

In [14], we introduced a generation algorithm able to manage both static and dynamic unlinked faults. It is based on a graph representation where the set of edges represents the fault models. March tests are generated by traversing each edge of the graph. Despite its effectiveness, the proposed approach lacks of a rigorous formalization, and in the worst case, it has a nonpolynomial complexity.

In this paper, we propose a new approach to automatically generate march tests targeting static, dynamic, and linked faults. The main contributions of this work are, from one side, a formal fault model representation that extends the fault primitive notation proposed in [3] and a completely new march test generation algorithm with a polynomial complexity that strongly reduces the generation time.

The paper is organized as follows: Section 2 introduces the memory model and the fault model used to automatically generate the march test, whereas Section 3 details the steps of the automatic generation process. Section 4 presents the results obtained by using the proposed algorithm. Section 5 proposes some optimizations, and finally, Section 6 summarizes the main contributions of the paper.

## 2 FORMAL MODELS

The generation algorithm proposed in this paper relies on the use of formal models to describe both the good and the faulty memory behaviors.

### 2.1 Memory Model

This section introduces the formal model adopted to represent the memory under test. In this paper, we focus on bit-oriented memories only. The extension of the obtained march tests to word-oriented memories can be easily done according to the algorithm proposed in [15].

**Definition 1.** An  $N$ -cell 1-bit memory can be formally defined as a 4-upla:

$$\langle A, N, M, X_0 \rangle, \quad (1)$$

where

- $A = \{0, 1\}$  is the set of possible states of a memory cell,
- $N$  is the number of cells,
- $M = (c_0, c_1, c_2, \dots, c_{N-1}) | c_i \in A, 0 \leq i \leq N-1$  is the array of  $N$  cells, and
- $X_0 = \{r^i, w_d^i | 0 \leq i \leq N-1, d \in A\}$  is the set of possible memory operations, where  $r^i$  means a read operation on the cell  $i$ , whereas  $w_d^i$  means a write operation of the value  $d \in A$  on the  $i$ th cell.

The behavior of an  $N$ -cell 1-bit memory (Definition 1) can be formally represented by an Edge-Labeled Directed Graph (ELDG)  $G$  defined as

$$G = (V, E, L_e), \quad (2)$$

where

- $V$  is the set of  $2^N$  vertices representing the possible states of the memory,
- $E = \{(u, v) | u, v \in V\}$  is the set of edges, each one representing one of the possible memory operations that cause the transition from a vertex  $u$  to a vertex  $v$ , and

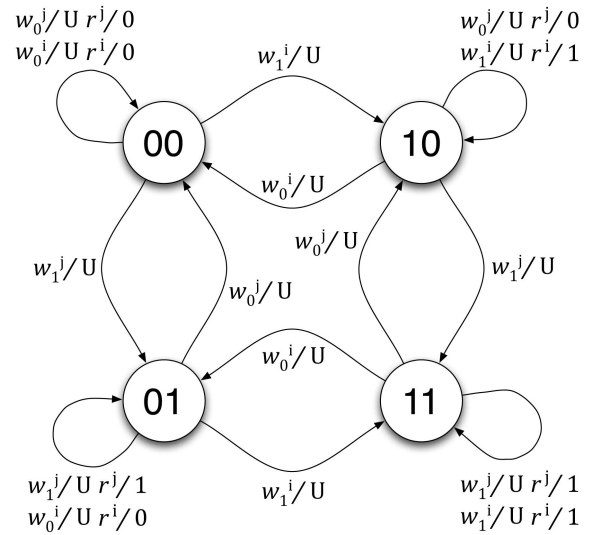


Fig. 1. Two-cell fault-free memory model  $G_0$ .

- $L_e : E \rightarrow \{\text{labels}\}$  is a label function that maps edges to labels, where given the edge  $(u, v)$ , the corresponding label is defined as follows:

$$\text{label} = x/k, \quad (3)$$

where

- $x \in X_0$  (Definition 1) is the memory operation able to fire the transition from  $u$  to  $v$ , and
- $k \in \{0, 1, U\}$  is the corresponding memory output, where the symbol  $U$  denotes the unknown value at the memory data output signals when a write operation is performed.

The proposed model is a modification of the mealy automata model proposed in [16]. The use of the ELDG allows modeling faulty situations where the result of a read operation is incorrect even if the content of the memory is correct. These situations were not representable using the model presented in [16].

As an example, Fig. 1 shows the model of a two-cell memory where the letters  $i$  and  $j$  are used to identify the first and the second cell, respectively. The ELDG has four states corresponding to the four possible combinations of the values stored into the cells. For the sake of readability, edges having the same initial and final state have been represented as a single edge with multiple labels.

### 2.2 Fault Model

This section introduces the formalism used to represent the target memory functional faults. Faults are modeled starting from *Faulty Behaviors* (FBs), i.e., deviations of the memory behavior from the expected one. An FB is expressed using the following notation:

- $D^i$  represents a faulty value  $D \in A$  (Definition 1) stored in the cell  $i$ .
- $R_D^i$  represents an erroneous output  $D \in A$  (Definition 1) obtained while reading the content of the cell  $i$ . This formalism is needed to represent classes of faults where a read operation returns an erroneous value, while the content of the memory cell is correct.

For example,  $0^i$  means that the  $i$ th cell assumes an erroneous value 0, while  $R_1^i$  means that a read operation on cell  $i$  returns the value 1 even if the content of the cell is 0.

The set of faulty memory cells involved in the faulty behavior is called  $f$ -cells. Based on the  $f$ -cells cardinality ( $|f\text{-cells}|$ ), faults can be clustered into the following classes:

- *Single-cell faults* ( $|f\text{-cells}| = 1$ ).
- *Two-cell faults* ( $|f\text{-cells}| = 2$ ). In this case, we can distinguish between an *aggressor cell* ( $a$ -cell) and a *victim cell* ( $v$ -cell). The former is the cell that sensitizes the FB, and the latter is the cell that shows the effect of the FB;
- *Multiple-cell faults* ( $|f\text{-cells}| \geq 2$ ). Not all multiple-cell faults are detectable using march tests. Our model allows describing multiple-cell faults, but the proposed generation algorithm will be able to produce results only for those faults detectable using march tests, i.e., faults described by two-cell FBs sharing the same aggressor cell, for example, three-cell linked faults [5].

An FB is sensitized by the application of a sequence of *stimuli* on the  $f$ -cells. A *stimulus*  $S$  is composed of an *initial condition*  $C$  of the  $f$ -cells, followed by an optional sequence of *memory operations*  $op_1, op_2, \dots, op_m$ :

$$S = (C, op_1, op_2, \dots, op_m) | op_i \in X, m \geq 0, \quad (4)$$

where

- $C = (s^1, s^2, \dots, s^k) | s^i \in A \cup \{-\}, 1 \leq k \leq |f\text{-cells}|$ , where  $s^i$  identifies one of the  $f$ -cells, and “-” denotes a don’t care condition, i.e., the initial value of that cell does not influence the faulty behavior,
- $op_i \in X = X_0 \cup \{r_d^i | 0 \leq i \leq N-1, d \in A\}$ , where  $X_0$  is defined in Definition 1, and  $r_d^i$  is a *read-and-verify* operation performed on the  $i$ th cell. The value  $d$  means “read the content of the cell  $i$  and verify that its value is equal to  $d$ .” The sequence of memory operations can be omitted when an FB is sensitized just by the  $f$ -cells being in a certain condition (e.g., State Coupling fault [17]).

According to the number of operations in the sequence ( $m$ ), the fault is classified as *Static* ( $m \leq 1$ ) or *Dynamic* ( $m > 1$ ).

Examples of possible stimuli are the following:

- $S = 0^i$  corresponds to an FB sensitized by the state of the faulty cell  $i$  equal to 0.
- $S = -, w_1^i$  corresponds to an FB sensitized by writing 1 into the faulty cell  $i$ , regardless of the current state of the cell.
- $S = 1^i, w_1^i r^i$  corresponds to an FB sensitized by a write operation of the value 1 on the faulty cell  $i$ , immediately followed by a read operation on the same cell. In this case, the FB is sensitized only if the two operations are applied starting with the cell  $i$  containing the value 1.

**Definition 2.** A Functional Fault Primitive (FFP) represents the difference between an expected (fault-free) and the observed

(faulty) memory behavior under a set of performed operations, denoted by

$$FFP = \langle S/FB \rangle, \quad (5)$$

where  $S$  (4) and  $FB$  represent a stimulus and a faulty behavior, respectively.

A functional memory fault model is a nonempty set of FFPs. For example, the *Inversion Coupling Fault* (a transition performed on an aggressor cell  $a$  causes the inversion of the logic value stored in a victim cell  $v$  [2]) can be described by the following two FFPs:

$$C_{inv} = \{FFP_1 = \langle \underbrace{0^a 0^v}_S, \underbrace{w_1^a / 1^v}_{FB} \rangle, FFP_2 = \langle \underbrace{0^a 1^v}_S, \underbrace{w_1^a / 0^v}_{FB} \rangle\}. \quad (6)$$

### 3 AUTOMATIC TEST GENERATION METHODOLOGY

The proposed march test generation methodology is based on the functional memory model introduced in Section 2.1 and on the definition of functional faults in terms of FFPs (Section 2.2). The main steps of the generation process are summarized as follows:

1. *Fault list representation.* Translate each FFP in the fault list into an “operational” representation of the faulty behavior, referred to as *Addressed FFP*, or *AFFP* (Section 3.1).
2. *Test pattern (TP) generation.* Generate the set of TPs able to cover each AFFP. Each TP is represented by an additional edge on the ELDG modeling the memory (Section 3.2).
3. *March test generation.* Traverse the ELDG to generate the march test.

#### 3.1 Fault List Representation

The FFP formalism describes the conditions to sensitize and detect FBs by considering the  $f$ -cells only. It does not consider the actual position of these cells in a generic  $N$ -cell memory. To map an FFP into a generic  $N$ -cell memory model, we therefore introduce the concept of AFFP. An *AFFP* is an instantiation of an FFP with an explicit indication of the addresses of the involved cells and both the faulty and the fault-free final memory state, after applying the stimulus  $S$  defined in the FFP (see Definition 2). The AFFP formalism strictly depends on both the number of memory cells involved in the fault ( $|f\text{-cells}|$ ) and on the size  $N$  of the target memory. It can be formalized as

$$AFFP = \langle I, E_s, F_v, G_v \rangle, \quad (7)$$

where

- $I = (i_0, i_1, i_2, \dots, i_{N-1}) | i_k \in A \cup \{-\}, 0 \leq k \leq N-1$  is the initial state of the memory, i.e., the values stored in the  $N$  cells of the target memory as defined by the initial conditions of the FFP. The first value corresponds to the less significant bit (i.e., the memory cell with the lowest address).

- $E_s = (op_1, op_2, \dots, op_m) | op_i \in X, 1 \leq i \leq m$  is the sequence of  $m$  operations, performed on the aggressor cell, needed to sensitize the fault, according to the stimulus defined in the FFP. Each operation belongs to the alphabet  $X$  defined in (4).
- $F_v = (f_0, f_1, \dots, f_{N-1}) | f_k \in A \cup \{U\}, 0 \leq k \leq N-1$  are the logical values (faulty state) stored in the memory cells after applying  $E_s$  in case of a faulty memory.
- $G_v = (g_0, g_1, \dots, g_{N-1}) | g_k \in A \cup \{U\}, 0 \leq k \leq N-1$  are the logical values (expected state) stored in the memory cells after applying  $E_s$  on a fault-free memory.

Since  $N$  does not necessarily corresponds to the number of involved faulty memory cells ( $|f\text{-cells}|$ ), each FFP can generate several AFFPs. For example, considering the Inversion Coupling fault FFPs defined in (6) applied to the two-cell memory represented in Fig. 1 ( $N = 2, i = \text{address of the first cell}, j = \text{address of the second cell}$ ), we obtain the following AFFPs:

$$FFP_1 = \langle 0^a 0^v, w_1^a / 1^v \rangle \begin{cases} AFFP_1 = \langle 00, w_1^i, 11, 10 \rangle, \\ AFFP_2 = \langle 00, w_1^j, 11, 01 \rangle, \end{cases} \quad (8)$$

$$FFP_2 = \langle 0^a 1^v, w_1^a / 0^v \rangle \begin{cases} AFFP_3 = \langle 01, w_1^i, 10, 11 \rangle, \\ AFFP_4 = \langle 10, w_1^j, 01, 11 \rangle. \end{cases}$$

### 3.2 Test Pattern Generation

From an AFFP, it is easy to define the sequence of memory operations, TP, used to detect the corresponding faulty behavior as

$$TP = \langle AFFP, O_s \rangle, \quad (9)$$

where AFFP is the target AFFP, and  $O_s = \{r_d^i\}$  is the read-and-verify operation (4) performed on the victim cell, needed to observe the fault effect.

For example, the four AFFPs defined in (8) are covered by the following TPs in a two-cell memory:

$$AFFP_1 = \langle 00, w_1^i, 11, 10 \rangle \rightarrow TP_1 = \langle \langle 00, w_1^i, 11, 10 \rangle, r_0^j \rangle,$$

$$AFFP_2 = \langle 00, w_1^j, 11, 01 \rangle \rightarrow TP_2 = \langle \langle 00, w_1^j, 11, 01 \rangle, r_0^i \rangle,$$

$$AFFP_3 = \langle 01, w_1^i, 10, 11 \rangle \rightarrow TP_3 = \langle \langle 01, w_1^i, 10, 11 \rangle, r_1^j \rangle,$$

$$AFFP_4 = \langle 10, w_1^j, 01, 11 \rangle \rightarrow TP_4 = \langle \langle 10, w_1^j, 01, 11 \rangle, r_1^i \rangle. \quad (10)$$

Each TP can be represented by an additional directed edge (faulty edge) from the state  $I$  to the state  $G_v$  defined in (7) on the memory model introduced in Section 2.1. The faulty edge label is defined as  $E_s, O_s$ , where  $E_s$  is the sequence of sensitizing operations, and  $O_s$  is the read-and-verify operation required to detect the fault, as defined in (7) and (9), respectively. The ELDG including the faulty edges is named *Pattern Graph* (PG) and is defined as

$$PG = (V, E \cup F, L_e \cup L_f), \quad (11)$$

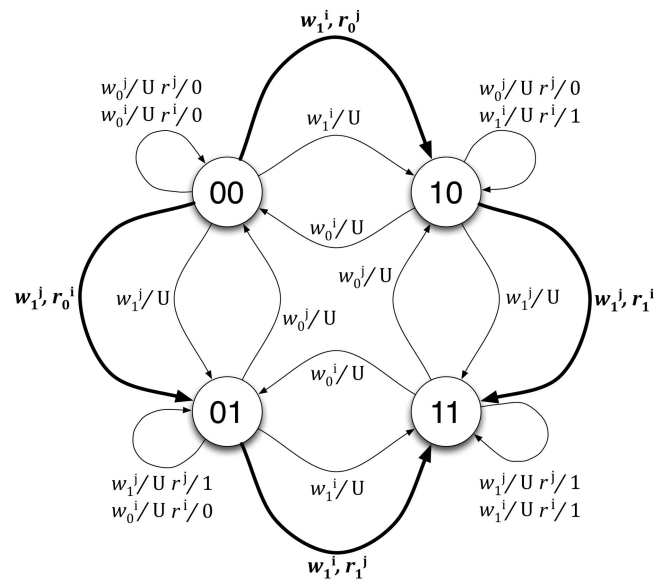


Fig. 2.  $CF_{inv}$  PG ( $PG_{CF}$ ).

where each vertex  $v \in V$  is associated to one of the  $2^N$  memory states (1),  $E$  is the set of edges modeling the fault-free memory (1),  $F$  is the set of faulty edges,  $L_e$  is the label function for the fault-free edges (2), and  $L_f$  is the label function for the faulty edges.

It is clear that considering a real  $N$ -cell memory, the complexity of the model explodes due to the number of nodes of the PG. Anyway, a march test covering an  $n$ -cell fault can detect the same fault on any memory of size  $N \geq n$  [2]. In general, the minimum number of required states of the PG can be defined as  $2^{MaxF}$ , where  $MaxF$  is the maximum number of faulty cells involved in the FFPs defined in the fault list. As stated in Section 2.2, the proposed algorithm is designed to work with faults described by two-cell FBs only, thus requiring a four-state PG.

Fig. 2 shows the PG (named  $PG_{CF}$  in the sequel) modeling the four TPs defined in (10). Bold edges represent the faulty edges.

### 3.3 March Test Generation Algorithm

In this section, the PG introduced in Section 3.2 is used to generate a march test detecting the set of faults described in the graph.

**Definition 3.** A march test consists of a sequence of march elements (MEs). An ME consists of a sequence of operations applied to each cell in the memory based on a given addressing order (AO) ( $\uparrow$  for the up AO,  $\downarrow$  for the down AO, and  $\updownarrow$  for any AO) [2].

The generation problem consists of finding a sequence of TPs (i.e., memory operations) able to detect the target set of memory faults while respecting the definition of march test (see Definition 3).

**Definition 4.** Given an ELDG  $G = (V, E, L_e)$ , a directed path  $(v_0, v_i)$  with  $v_0, v_i \in V$  is an ordered sequence of vertices and edges  $(v_0, e_0, v_1, e_1, \dots, v_{i-1}, e_{i-1}, v_i)$ , where each edge  $e_j \in E$



```

1:  $V \leftarrow$  select  $v$  from possible initial states with
    $\max(\#(FE_v))$ .
2:  $ME \leftarrow \emptyset, AO \leftarrow NULL$ 
3: repeat
4:    $fe \leftarrow get\_fe(FE_V)$  it chooses one  $fe$  incident from
     the current state  $V$ 
5:   if ( $fe = \emptyset$ ) then
6:      $close\_me(ME)$ 
7:      $print(ME)$ 
8:      $ME \leftarrow \emptyset, AO \leftarrow NULL$ 
9:      $V \leftarrow get\_new\_state()$ 
10:  else
11:     $put\_fe\_in\_me(fe, ME)$ 
12:    Update  $V$  with  $fe$  {it determines the new state
      by moving on the good machine}
13:    delete  $fe$ 
14:  end if
15: until ( $FE_v = \emptyset; \forall v$ )

```

Fig. 4. March test generation algorithm.

It represents the set of TPs with the same aggressor cell and for which the application of the sensitizing operations changes the memory state from  $v_1$  to  $v_2$ .

We call transition a TP that belongs to  $FE_{v_1 \rightarrow v_2}^i$  with  $v_1 \neq v_2$ , while we call loop a TP that belongs to  $FE_{v_1 \rightarrow v_2}^i$  with  $v_1 = v_2$ .

As an example, let us consider the two TPs,  $TP_1 = \langle \langle 00, w_1^i, 11, 10 \rangle, r_0^j \rangle$  and  $TP_2 = \langle \langle 00, r^i, 01, 00 \rangle, r_0^j \rangle$ . Both  $TP_1$  and  $TP_2$  have the same aggressor cell ( $i$ ). After applying  $TP_1$ , the memory state changes from 00 to 01. According to Definition 7,  $TP_1$  is a transition, while  $TP_2$  is a loop.

We can define  $FE_v^i$  in terms of  $FE_{v_1 \rightarrow v_2}^i$  as

$$FE_v^i = \overbrace{FE_{v \rightarrow v}^i}^{\text{Loops}} \cup \left( \bigcup_{\forall v_j \neq v} \overbrace{FE_{v \rightarrow v_j}^i}^{\text{Transitions}} \right). \quad (16)$$

We actually split the set of faulty edges into two sets: the former representing the set of faulty edges incident from  $v$  and incident to  $v$  (the loops set) and the latter being the union of the sets of faulty edges incident from  $v$  and incident to a node  $v_j$  different from  $v$  (the transition set).

**Definition 8.** We define the cardinality of a set  $FE_v$  (written as  $\#(FE_v)$ ) as the number of elements in the set and the cost (\$) of a set  $FE_v$  as

$$\$(FE_v) = \sum_{i=0}^{N-1} \$(FE_v^i). \quad (17)$$

The cost of the  $i$ th component  $FE_v^i$  is defined as the cost of the loop set ( $FE_{v \rightarrow v}^i$ ) plus the cost of the transition set:

$$\$(FE_v^i) = \$(FE_{v \rightarrow v}^i) + \sum_{\forall w \in H_1} \$(FE_{v \rightarrow w}^i | w \neq v), \quad (18)$$

where  $H_1$  is the set of nodes reachable from  $v$  by traversing a single faulty edge.

The cost of the loop set corresponds to its cardinality:  $\$(FE_{v \rightarrow v}^i) = \#(FE_{v \rightarrow v}^i)$ . The cost of the transition set is

```

1: if ( $AO$  not defined) then
2:   select  $i$  where  $\$(FE_v^i) = \max(\$(FE_v^i))$  with  $i =$ 
    $0 \vee i = N - 1$ .
3:   if ( $i=0$ ) then
4:      $AO \leftarrow '\uparrow'$ 
5:   else
6:      $AO \leftarrow '\downarrow'$ 
7:   end if
8: else
9:   if ( $AO=\uparrow$ ) then
10:     $i \leftarrow 0$ 
11:  else
12:     $i \leftarrow N - 1$ 
13:  end if
14: end if
15: if ( $\#(FE_{v \rightarrow v}^i) > 0$ ) then
16:   random select  $fe \in FE_{v \rightarrow v}^i$ 
17:   return( $fe$ )
18: else
19:   select  $fe \in FE_{v \rightarrow w}^i$  where  $O_s(fe)$  belongs to
     current ME, if no  $fe$  satisfy the constraints random
     select  $fe$ .
20:   return( $fe$ )
21: end if

```

Fig. 5. Function  $get\_fe(FE_v)$ : returns the selected  $fe$ .

defined as 0 if the set is empty and as 1 plus the cost of the  $i$ th component of the destination state if the set is not empty. The value 1 represents the cost of the operation needed to move from state  $v$  to state  $w$ :

$$\$(FE_{v \rightarrow w}^i) = \begin{cases} 1 + \$(FE_w^i) & \text{if } \#(FE_{v \rightarrow w}^i) > 0, \\ 0 & \text{if } \#(FE_{v \rightarrow w}^i) = 0. \end{cases} \quad (19)$$

We now have all the elements and the definitions required to present the generation algorithm, whose main steps are summarized in Fig. 4 and described in the following.

The first operation required by the algorithm (step 1 in Fig. 4) is to select the *initial state*. Due to the march test characteristics (see Definition 3), the only valid initial states are the ones with all memory cells initialized with the same value (i.e., all “0” or all “1”). Among them, the algorithm chooses the state  $V$  with the maximum number of faulty edges incident from it, i.e.,  $\max(\#(FE_v)) \forall v$ . An empty ME is initialized (step 2 in Fig. 4). At this point, no AO is specified for the current ME. The algorithm has to select one faulty edge ( $fe$ ) incident from the current state  $V$  (step 4 in Fig. 4).

To select the new  $fe$  the  $get\_fe$  function is invoked (Fig. 5). This function identifies the  $FE_v^i$  (see Definition 6) with the maximum cost (step 2 in Fig. 5). Since no AO is still selected and since we consider classic up and down AOs only, the only choice the algorithm has to generate a marchable sequence of operations (see Lemma 1) is to select an  $fe$  with an  $a$ -cell equal to the first or to the last cell of the given memory model.

First, the algorithm tries to select a loop that still needs to be traversed (if more than one loop exists, then the choice is random) (steps 16 and 17 in Fig. 5); otherwise, one of the

```

1: if ( $E_s(fe) \notin ME$ ) then
2:   add  $E_s(fe)$  to ME {add the sensitizing operation(s)}
3: end if
4: if ( $O_s(fe) \notin ME$ ) then
5:   if ( $O_s(fe)$  is on the same cell of  $E_s(fe)$ ) then
6:     add  $O_s(fe)$  in ME or, when ME is closed place the read operation at the beginning of the next ME {single-cell}
7:   else if ( $O_s(fe) < E_s(fe)$  and AO is down) or ( $O_s(fe) > E_s(fe)$  and AO is up) then
8:     insert  $O_s(fe)$  at the beginning of the current ME (if not exists before) {two or more cells}
9:   else
10:    insert  $O_s(fe)$  at the beginning of the next ME and close the ME {two or more cells}
11:   end if
12: end if

```

Fig. 6. Function *put\_fe\_in\_me* (*fe*, *ME*).

transitions is selected (again, the choice among the transitions is random) (steps 19 and 20 in Fig. 5). The selected faulty edge automatically determines the AO (steps 3-7 in Fig. 5). From this point to the end of the ME generation, only operations performed on the first cell of the identified addressing sequence (the first cell in case of  $\uparrow$  and the last cell in case of  $\downarrow$ ) can be added (steps 9-13 in Fig. 5).

This constraint guarantees that the resulting ME is a marchable SMT0 (see Lemma 1). Every time a faulty edge is traversed, the operations in its label are added to the ME (step 11 in Fig. 4). If not already present in the ME (due to previous operations), also the read-and-verify operation needed to observe the fault is added. At this point, the new memory state is calculated according to the fault-free memory model (step 12 in Fig. 4), and the faulty edge is deleted from the graph (the fault is detected) (step 13 in Fig. 4).

The ME generation ends if no faulty edges can be selected from the current state  $V$  (step 5 in Fig. 4). Before completing the generation of the ME (step 7 in Fig. 4), the *close\_me* function (Fig. 7) is invoked (step 6 in Fig. 4). It simulates the application of the operations in the ME on each memory cell of the target memory model. Starting from  $V$ , the initial state of the current ME, each operation is applied, and the current state is updated according to the memory model of the fault-free memory (step 2 in Fig. 7). If during this operation additional faulty edges are traversed, they are marked as detected and removed from the graph (step 3 in Fig. 7).

At this point, we have reached the final state of the ME. The ME is completely defined, and if there are still faulty edges to traverse (step 15—Fig. 4), the next ME is initialized, and the process is repeated.

Each time a faulty edge is selected the *put\_fe\_in\_me* (Fig. 6) function is invoked to append the faulty edge label to the ME. Before adding the operations needed to sensitize and observe the target fault (step 2 in Fig. 6), the function checks whether the ME already contains the required

```

1: for all memory cells do
2:   apply  $ME$ 
3:   delete traversed faulty edges ( $fe$ )
4: end for

```

Fig. 7. Function *close\_me* (*ME*).

operations (step 1 in Fig. 6). If it does, then no operations are added.

After the sensitizing operations, the read and verify must be inserted. Two cases may occur:

- The  $a$ -cell is equal to the  $v$ -cell (i.e., a single-cell fault) (step 5 in Fig. 6). The read-and-verify operation is placed after the sensitizing operations (if it does not already exists) (step 6 in Fig. 6).
- The  $a$ -cell differs from the  $v$ -cells (i.e., an  $n$ -cell fault with a single aggressor cell and a set of victim cells):
  - If  $a$ -cell  $>$   $v$ -cells and the AO is  $\uparrow$  (or  $a$ -cell  $<$   $v$ -cells and AO is  $\downarrow$ ) (step 7 in Fig. 6), the read operation is inserted at the beginning of the ME (step 8 in Fig. 6).
  - Otherwise, the ME is closed, and the read operation is added as the first operation of the next ME according to [6] (step 10 in Fig. 6).

The behavior of the algorithm is greedy since it tries to insert in each ME the highest possible number of faulty edges. A key point is the faulty edge selection (step 4 in Fig. 4). The basic idea is to choose the faulty edges from the  $FE_v^i$  having the highest cost (step 2 in Fig. 5), where, in fact, the cost corresponds to the number of loops and transitions. This actually means selecting the  $FE_v^i$  allowing the highest number of movements (transitions) on the PG.

When the current memory state has an empty  $FE_V$ , the algorithm needs to move to a new memory state  $v_1$  having the highest  $\$(FEv_1)$  (step 9 in Fig. 4). The function *get\_new\_state* (Fig. 8) performs this operation by traversing the fault-free edges of the graph and by appending the corresponding operations (labels) to the ME. In other words, the algorithm builds an initialization ME to reach the target state  $v_1$ .

To better understand the generation algorithm, we will show its application on the PG in Fig. 3. The initial selected state is  $V = 00$ , since  $\$(FE_{00}) = 3$  (00 is the state with the highest number of faulty edges incident from it). We have to choose one faulty edge, so we calculate the cost of the different sets of faulty edges:  $FE_{00}^i = \{“r^i, r_0^j”, “w_1^i, r_0^j”\}$ ,  $\$(FE_{00}^i) = 2$ ,  $FE_{00}^j = \{“w_1^j r^j, r_0^i”\}$ , and  $\$(FE_{00}^j) = 1$ . The algorithm chooses  $FE_{00}^i$  (up AO as defined in steps 3-7 in Fig. 5), and  $“r^i, r_0^j”$  (loop set) is selected and added to the ME, which becomes  $(r_0^j)$  with the current memory state still equal to “00.” At this point, the algorithm chooses the only

```

1: select  $v$  such as  $FE_v = \max(\$(FE_v))$ 
2: copy the operation required to reach  $v$  into ME
3: return( $v$ )

```

Fig. 8. Function *get\_new\_state*(): returns the new memory state.



TABLE 1  
Generated March Tests for Unlinked Static Faults

Name	Algorithm	O(n)	FaultList
MATS [2]	$\{\uparrow(w_1) \downarrow(r_1 w_0) \downarrow(r_0)\}$	4n	SAF
MATS+ [2]	$\{\uparrow(w_1) \uparrow(r_1 w_0) \downarrow(r_0 w_1)\}$	5n	SAF,ADF
X [2]	$\{\uparrow(w_0) \uparrow(r_0 w_1) \downarrow(r_1 w_0) \uparrow(r_0)\}$	6n	SAF,TF ADF
C- [2]	$\{\uparrow(w_1) \uparrow(r_1 w_0) \uparrow(r_0 w_1) \downarrow(r_1 w_0) \downarrow(r_0 w_1) \downarrow(r_1)\}$	10n	SAF, CFid ADF, TF, CFinv
CLI [13]	$\{\uparrow(w_1) \uparrow(r_1 w_0 w_1) \downarrow(r_1)\}$	5n	CFinv
SS [19]	$\{\uparrow(w_0) \uparrow(r_0 r_0 w_0 r_0 w_1) \uparrow(r_1 r_1 w_1 r_1 w_0) \downarrow(r_0 r_0 w_0 r_0 w_1) \downarrow(r_1 r_1 w_1 r_1 w_0) \downarrow(r_0)\}$	22n	All static

remaining choice in  $FE_{00}^i$  represented by “ $w_1^i, r_0^j$ ”.  $w_1^i$  (sensitizing sequence) is added to the ME, which becomes  $(r_0^i w_1^i)$ , while the observation is not required since it is already present.

The new state is now  $v = 10$ .  $FE_{10}^i = \{\emptyset\}$ , so the algorithm closes the ME and generates the corresponding ME:  $\uparrow(r_0 w_1)$ . It then simulates the operations of the ME on cell  $j$ . The simulation shows that also “ $w_1^j, r_1^i$ ” (from state “10”) is traversed. The read-and-verify operation performed on cell  $i$  is added to the next ME, which becomes  $(r_1^i)$  again with up AO (step 10 in Fig. 6).

The current state is now  $v = 11$ . Since  $FE_{11}$  is empty, the ME is closed, and the corresponding ME is generated:  $\uparrow(r_1)$ . We now must change state to “00,” which is the only state still having faulty edges. The operation to move to “00” is  $w_0^i$ , it is added to the new ME, which becomes  $(w_0^i)$  with up AO. The ME is closed, and the corresponding generated ME is  $\uparrow(w_0)$ . Now, the algorithm traverses the last faulty edge, and it adds the operations on its label into a new ME, obtaining an ME equal to  $(w_1^j r_1^j)$  with down AO. It also inserts the read-and-verify operation, obtaining an ME equal to  $(r_0^j w_1^j r_1^j)$ . At this point, all the faulty edges are traversed, the ME is closed, and the generated ME is  $\downarrow(r_0 w_1 r_1)$ . The algorithm ends, and the final generated march test is  $\uparrow(w_0) \uparrow(r_0 w_1) \uparrow(r_1) \uparrow(w_0) \downarrow(r_0 w_1 r_1)$ , where the first ME allows initializing the memory in the selected initial state.

## 4 EXPERIMENTAL RESULTS

This section reports some experimental results obtained by applying the proposed generation algorithm to different fault lists. Tables 1 and 2 report the generated march tests for different sets of target unlinked static and dynamic faults. For each march test, we report the name (column 1), the algorithm (column 2), the complexity in terms of the number of operations (column 3), and the target fault list (column 4).

Table 1 reports march tests targeting different sets of static unlinked faults. Static faults have been deeply studied in the last years, and all the algorithms generated by the tool were already published in previous works.

When we move to the more complex dynamic unlinked faults, the proposed algorithm demonstrates its real value. In this case, we have been able to generate tests shorter than

TABLE 2  
Generated March Tests for Unlinked Dynamic Faults

Name	Algorithm	O(n)	FaultList
ABdrf	$\{\uparrow(w_1) \downarrow(w_0 r_0 r_0 w_1 r_1 r_1)\}$	7n	Dynamic read faults
AB1 [21]	$\{\uparrow(w_0) \downarrow(w_1 r_1 w_1 r_1 r_1) \downarrow(w_0 r_0 w_0 r_0 r_0)\}$	11n	Single-Cell dynamic Faults
AB [21]	$\{\uparrow(w_1) \downarrow(r_1 w_0 r_0 w_0 r_0) \downarrow(r_0 w_1 r_1 w_1 r_1) \uparrow(r_1 w_0 r_0 w_0 r_0) \uparrow(r_0 w_1 r_1 w_1 r_1) \downarrow(r_1)\}$	22n	Two-Cells Dynamic faults
AB2	$\{\uparrow(w_1) \downarrow(w_1 r_1 w_0 r_0 w_0 r_0 w_1 r_1)\}$	9n	Dynamic Read Destructive Fault (dRDF)
AB3	$\{\uparrow(w_0) \downarrow(w_0 r_0 r_0 w_1 r_1 r_1 w_1 r_1 r_1 w_0 r_0 r_0)\}$	13n	Dynamic Deceptive Read Destructive Fault (dDRDF)
AB4	$\{\uparrow(w_0) \uparrow(r_0 w_0 r_0 w_1 r_1) \uparrow(r_1 w_1 r_1 w_0 r_0) \downarrow(r_0 w_0 r_0 w_1 r_1) \downarrow(r_1 w_1 r_1 w_0 r_0) \downarrow(r_0)\}$	22n	Dynamic Read Destructive Coupling Fault (dCFrd)

the ones already published. We started from the set of dynamic faults described in [20]. These faults are considered to be some of the most realistic ones for current memory technologies.

March AB1 (Table 2), with a complexity of 11n, is able to detect the entire set of single-cell two-operation dynamic faults [20]; compared with March RAW1 (13n), which was manually designed, it guarantees the same fault coverage but reduces the test length by two operations or 18.18 percent. March AB (Table 2), with a complexity of 22n, is able to detect the entire set of realistic two-cell two-operation dynamic faults [20]; compared with March RAW (26n), again manually designed, it provides the same fault coverage but reduces the test length by four operations or 15.38 percent. Finally, we propose three new march tests, March AB2, March AB3, and March AB4, generated for particular subsets of dynamic faults in order to demonstrate the freedom in choosing the target fault list.

We also applied the proposed algorithm, after applying the modification to the PG proposed in [22] to model linked faults, to the set of realistic linked faults presented in [5]. In Table 3, Fault List #1 includes single-, two- and three-cell linked faults proposed in [5], whereas Fault List #2 includes all single-cell linked faults proposed in [5].

TABLE 3  
Generated March Tests for Linked Faults

Name	Algorithm	O(n)	FaultList
AB	$\{\uparrow(w_1) \downarrow(r_1 w_0 r_0 w_0 r_0) \downarrow(r_0 w_1 r_1 w_1 r_1) \uparrow(r_1 w_0 r_0 w_0 r_0) \uparrow(r_0 w_1 r_1 w_1 r_1) \downarrow(r_1)\}$	22n	#1
ABL1	$\{\uparrow(w_0) \downarrow(w_0 r_0 r_0 w_1) \downarrow(w_1 r_1 r_1 w_0)\}$	9n	#2

TABLE 4  
BIST Optimized March Test

Algorithm	O(n)	FaultList
$\{\uparrow(w_0) \uparrow(r_0 w_1 w_0) \uparrow(r_0 w_1) \uparrow(r_1 w_0 w_1) \uparrow(r_1 w_0) \uparrow(r_0)\}$	12n	SAF, TF RDF

We compared our new generated march tests with already published algorithms targeting the same fault list. In particular, we considered the following:

- *43n March test* [9]. It is automatically generated and deals only with a subset of faults defined in Fault List #1.
- *41n March SL* [5]. It is the state of the art in the class of hand-generated march tests. It covers Fault List #1.
- *23n March MSL* [23]. It is automatically generated, and it reduces the complexity of March SL. It covers Fault List #1.
- *11n March LF1* [5]. It is one of the most used march tests, and it is able to cover Fault List #2.

March AB of complexity 22n, targeting Fault List #1, reduces the test time by 48.8 percent with respect to the 43n march test, 46.3 percent with respect to March SL, and 4.35 percent with respect to March MSL. Similarly, March ABL1, which targets Fault List #2, reduces the test length by 18.1 percent with respect to March LF1 (the state of the art for the same list of faults).

It is relevant to note that using our algorithm, we have been able to generate a new test, March AB, able to detect both dynamic and linked faults. Moreover, the same algorithm is also able to detect the full list of static unlinked faults detected by the 22n March SS. In all the experiments, the generation process was shorter than 1 second of CPU time. All generated march tests have been verified by fault simulation using the memory fault simulator published in [24] and [25] to validate the correctness of the test with respect to the target fault list.

## 5 OPTIMIZATIONS

March tests are critical components in any ATE-based or BIST Memory test architecture. In the latter case, it has been shown that the BIST hardware overhead can be reduced if the march test shows some particular characteristics such as uniformity [8] (a constant number of operations in each ME), symmetry [2] (particularly important on transparent march tests), or single AO. The proposed march test generation algorithm already produces, if possible, symmetric march tests (see Table 3). Additional constraints can be very easily added in the generation phase performed by traversing the ELDG. We successfully implemented the possibility of generating single-AO march tests, by adding this additional constraint in the *get\_fe* function (see Fig. 5), which is the function defining the AO. Using this optimization, we have been able to generate the single-AO march test proposed in Table 4. Other constraints can be easily implemented; the only drawback is that they can lead to situations where no solutions can be generated.

## 6 CONCLUSIONS

This paper addresses two very important issues usually faced by researchers and test engineers in the field of memory testing. It provides a clear and flexible formalism to model memories and faulty behaviors, and it proposes an efficient algorithm to automatically generate march tests. The flexibility of the fault model formalism allows describing not only traditional static and dynamic faults but also linked and user-defined faults. This feature makes the proposed research very appealing for both memory manufacturers and users. With respect to previously presented approaches, our methodology allows generating shorter march tests in a very low computation time, without exhaustive searches. The paper presents march tests for the complete set of static faults and for most of the known dynamic faults, obtaining both already published and new test algorithms. What emerges from the experimental results is the efficiency of the algorithm, which is able to significantly reduce the march test length and, therefore, the test time for many significant fault lists. Ongoing research activities are focusing on the extension of the model to multiport memory faults and on the possibility of introducing additional constraints on the generated march tests.

## REFERENCES

- [1] International Technology Roadmap for Semiconductors, <http://www.itrs.net/>, 2008.
- [2] A.J. van de Goor, "Using March Tests to Test SRAMs," *IEEE Design and Test of Computers*, vol. 10, no. 1, pp. 8-14, Jan.-Mar. 2004.
- [3] A.J. van de Goor and Z. Al-Ars, "Functional Memory Faults: A Formal Notation and a Taxonomy," *Proc. 18th IEEE VLSI Test Symp. (VTS '00)*, pp. 281-289, Apr./May 2000.
- [4] S. Hamdioui, R. Wadsworth, J. Reyes, and A. van de Goor, "Importance of Dynamic Faults for New SRAM Technologies," *Proc. Eighth IEEE European Test Workshop (ETW '03)*, pp. 29-34, May 2003.
- [5] S. Hamdioui, Z. Al-Ars, A.J. van de Goor, and M. Rodgers, "Linked Faults in Random Access Memories Concept Fault Models Test Algorithms and Industrial Results," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 5, pp. 737-757, May 2004.
- [6] B. Smit and A.J. van de Goor, "The Automatic Generation of March Tests," *Proc. IEEE Int'l Workshop Memory Technology, Design and Testing (MTDT '94)*, pp. 86-91, Aug. 1994.
- [7] K. Zarrineh, S. Upadhyaya, and S. Chakravarty, "Automatic Generation and Compaction of March Tests for Memory Arrays," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 845-857, Dec. 2001.
- [8] S. Al-Harbi and S. Gupta, "An Efficient Methodology for Generating Optimal and Uniform March Tests," *Proc. 19th IEEE VLSI Test Symp. (VTS '01)*, pp. 231-237, Apr.-May 2001.
- [9] S.M. Al-Harbi and S. Gupta, "Generating Complete and Optimal March Tests for Linked Faults in Memories," *Proc. 21st IEEE VLSI Test Symp. (VTS '03)*, pp. 254-261, Apr./May 2003.
- [10] D. Niggemeyer and E.M. Rudnick, "Automatic Generation of Diagnostic Memory Tests Based on Fault Decomposition and Output Tracing," *IEEE Trans. Computers*, vol. 53, no. 9, pp. 1134-1146, Sept. 2004.
- [11] C.-F. Wu, C.-T. Huang, K.-L. Cheng, and C.-W. Wu, "Simulation-Based Test Algorithm Generation for Random Access Memories," *Proc. 18th IEEE VLSI Test Symp. (VTS '00)*, pp. 291-296, Apr./May 2000.
- [12] C.-F. Wu, C.-T. Huang, and C.-W. Wu, "Ramses: A Fast Memory Fault Simulator," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems (DFT '99)*, pp. 165-296, Nov. 1999.
- [13] S. Di Carlo, G. Di Natale, A. Benso, and P. Prinetto, "An Optimal Algorithm for the Automatic Generation of March Tests," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE '02)*, pp. 938-939, Mar. 2002.

- [14] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, and P. Prinetto, "Automatic March Tests Generation for Static and Dynamic Faults in SRAMs," *Proc. 10th IEEE European Test Symp. (ETS '05)*, pp. 122-127, May 2005.
- [15] A. van de Goor and I.B.S. Tlili, "A Systematic Method for Modifying March Tests for Bit-Oriented Memories into Tests for Word-Oriented Memories," *IEEE Trans. Computers*, vol. 52, no. 10, pp. 1320-1331, Oct. 2003.
- [16] J. Brzozowski and H. Jurgensen, "A Model for Sequential Machine Testing and Diagnosis," *J. Electronic Testing Theory and Application*, vol. 3, no. 3, pp. 219-234, Aug. 1992.
- [17] R. Dekker, F. Beenker, and L. Thijssen, "A Realistic Fault Model and Test Algorithms for Static Random Access Memory," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 567-572, June 1990.
- [18] A. Ney, P. Girard, C. Landrault, S. Pravossoudovitch, A. Virazel, and M. Bastian, "Slow Write Driver Faults in 65 nm SRAM Technology: Analysis and March Test Solution," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE '07)*, pp. 1-6, Apr. 2007.
- [19] S. Hamdioui, A.J. van de Goor, and M. Rodgers, "March SS: A Test for All Static Simple RAM Faults," *Proc. IEEE Int'l Workshop Memory Technology, Design and Testing (MTDT '02)*, pp. 95-100, July 2002.
- [20] S. Hamdioui, Z. Al-Ars, and A.J. van de Goor, "Testing Static and Dynamic Faults in Random Access Memories," *Proc. 20th IEEE VLSI Test Symp. (VTS '02)*, pp. 395-400, Apr./May 2002.
- [21] A. Benso, A. Bosio, S.D. Carlo, G.D. Natale, and P. Prinetto, "March AB, March AB1: New March Tests for Unlinked Dynamic Memory Faults," *Proc. IEEE Int'l Test Conf. (ITC '05)*, pp. 834-841, Nov. 2005.
- [22] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, and P. Prinetto, "Automatic March Tests Generations for Static Linked Faults in SRAMs," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE '06)*, pp. 1-6, Mar. 2006.
- [23] G. Harutanyan, V. Vardanian, and Y. Zorian, "Minimal March Test Algorithm for Detection of Linked Static Faults in Random Access Memories," *Proc. 24th IEEE VLSI Test Symp. (VTS '06)*, pp. 120-125, Apr./May 2006.
- [24] A. Benso, S.D. Carlo, G.D. Natale, and P. Prinetto, "Specification and Design of a New Memory Fault Simulator," *Proc. 11th IEEE Asian Test Symp. (ATS '02)*, pp. 92-97, Nov. 2002.
- [25] A. Benso, A. Bosio, S.D. Carlo, G.D. Natale, and P. Prinetto, "Memory Fault Simulator for Static-Linked Faults," *Proc. 15th IEEE Asian Test Symp. (ATS '06)*, pp. 31-36, Nov. 2006.



**Alfredo Benso** currently holds a tenured associate professor position in computer engineering in the Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy, where he teaches microprocessor systems and advanced programming techniques. In his scientific career, mainly focused on hardware testing and dependability, he coauthored more than 60 publications in books, journals, and conference proceedings. He is also actively involved in the IEEE Computer Society, where he has been the leading volunteer for several projects such as the TEchnical Committees Archives (TECA) database and the Conference Information Management Application (CIMA). He is an IEEE Computer Society Golden Core member and a senior member of the IEEE.



**Alberto Bosio** received the PhD degree in computer engineering from the Politecnico di Torino, Italy, in 2006. Since 2002, he worked in the area of digital systems dependability for safety-critical applications at the Politecnico di Torino. He is currently an associate professor in the Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier II/CNRS, Montpellier, France. His research activity mainly focused on the definition of new methodologies and the implementation of tools able to improve the development of highly dependable systems, at different levels: for basic digital components, for systems on chip, up to microprocessor-based systems. He is a member of the IEEE.



in journals and conference proceedings. He is a Golden Core member of the IEEE Computer Society and a member of the IEEE.

**Stefano Di Carlo** received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Torino, Italy, in 1999 and 2003, respectively. Since 2008, he has been an assistant professor in the Department of Control and Computer Engineering, Politecnico di Torino. In his scientific career, mainly focused on DFT techniques, SoC testing, BIST, and memory testing, he coauthored more than 40 publications



Technology Technical Council (TTTC) of the IEEE Computer Society as the webmaster. He is a member of the IEEE.

**Giorgio Di Natale** received the PhD degree in computer engineering from the Politecnico di Torino, Torino, Italy, in 2003. Currently, he is a senior researcher in the Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier II/CNRS, Montpellier, France. He has published articles in publications spanning diverse disciplines, including memory testing, fault tolerance, and secure chip design and test. He also serves the Test



IEEE Computer Society Test Technology Technical Council (TTTC) as the elected chair. He is a member of the IEEE and a member of the IEEE Computer Society.

**Paolo Prinetto** received the MS degree in electronic engineering from the Politecnico di Torino, Torino, Italy. He is a full professor of computer engineering in the Department of Control and Computer Engineering, Politecnico di Torino, and a joint professor at the University of Illinois, Chicago. His research interests include testing, test generation, BIST, and dependability. He is a Golden Core member of the IEEE Computer Society, and he served the

► **For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/publications/dlib](http://www.computer.org/publications/dlib).**