Automatic March tests generation for static and dynamic faults in SRAMs

Automatic March Tests Generation for Static and Dynamic Faults in SRAMs

A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto
Dipartimento di Automatica e Informatica
Politecnico di Torino - Turin, Italy
E-mail:{alfredo.benso, alberto.bosio, stefano.dicarlo, giorgio.dinatale, paolo.prinetto}@polito.it

Abstract

New memory production modern technologies introduce new classes of faults usually referred to as Dynamic Memory Faults. Although some hand-made March Tests to deal with these new faults have been published, the problem of automatically generate March Tests for Dynamic Faults has still to be addressed. In this paper we propose a new approach to automatically generate March Tests with minimal length for both Static and Dynamic Faults. The proposed approach resorts to a formal model to represent faulty behaviors in a memory and to simplify the generation of the corresponding tests.

1. Introduction

Silicon area is now so cheap and integration technologies so advanced that one can embed in a System-On-a-Chip (SOC) all the components and functions that historically were placed on a hardware board. Within SOCs, embedded memories are the densest components, accounting for up to 90% of chips area [1]. It is thus common finding, on a single chip, tens of memories of different types, sizes, access protocols and timing. Moreover they can recursively be embedded in embedded cores. The challenge of testing memories stems from the definition of realistic fault models and the design of test algorithms with minimal test application time [2].

Memory defects strongly depend on the target technology. Every time a new technology is introduced new defects appear and new fault models must be defined. In the last years, the so called Static Faults (SFs) (e.g., stuck-at faults, coupling faults …) [2] have been the predominant fault type. They are characterized by being sensitized by the execution of just a single memory operation. New faulty behaviors occur in latest technologies [3]. As an example, a write operation on a memory cell, immediately followed by a read operation, may cause the cell to flip. These behaviors cannot be modeled as Static Faults, since they require more than one operation to be sensitized. They are usually referred to as Dynamic Faults (DFs). The set of possible DFs is theoretically unlimited and wherever a new fault is observed a new custom test algorithm has to be generated.

Although the peculiar set of faults that can affects SRAMs require ad-hoc testing strategies, their regular structure allows adopting particularly simple algorithms, the most popular one being March Tests [4]. Several March Tests targeting different set of memory faults have been proposed [2]. Most of them have been generated by hand but, with the occurrence of new and more complex DF models, the task of hand writing test algorithms is becoming harder and it may lead to non optimal results. To overcome this problem in the last years several methodologies have been developed to automatically generate March Tests. In this paper we propose a new approach to automatically generate March Tests targeting both static and dynamic memory faults.

We successfully applied the proposed algorithm to an extensive set of static and realistic dynamic faults, obtaining both known and new March Tests, with a computation time in order of few seconds. To prove their correctness, all generated tests have been fault simulated using an in-house developed memory fault simulator [5]. Despite primarily targeting March Tests, our generation process can deal with faults requiring more complex test algorithms, as well.

The paper is structured as follows: Section 2 presents a survey of previous works in the field of automatic March Test generation and Section 3 introduces the memory and the fault model. Section 4 details the steps of the automatic March Test generation process, whereas Section 5 presents experimental results. Section 6 finally summarizes the main contributions and outlines future research activities.

2. State of the Art

Several authors faced the problem of the automatic generation of March Tests. [6] [7] present an algorithm for March Test generation exploiting a transition tree. The transition tree is generated in such a way that each path from the root node to a leaf represents a March Test. The March Test able to address the selected fault list is searched into the tree. The main problem of this

Proceedings of the European Test Symposium (ETS'05)
1530-1877/05 $20.00 © 2005 **IEEE**

approach is that the transition tree is unbounded. In order to limit the size of the tree, an upper bound on the number of nodes in a path is used. This can cause a high number of reiterations to find a solution making the algorithm inefficient and time consuming. Furthermore, this method performs an exhaustive search to find the shortest path on the transition tree. As the size of the transition tree increases, the algorithm becomes more and more inefficient.

In [8] [9] the authors present a branch and bound method that limits the search process to the parts of the tree where a solution exists and therefore a solution is found faster and more efficiently.

The approach presented in [10] generates a *Primitive March Test* for each fault. Then, different PMTs are combined to generate all possible March Tests by using a set of combination rules. The main drawback is that the rules used to compose the March Tests depend on the fault models.

[11] presents a completely different approach named Test Algorithm Generation by Simulation (TAGS). The proposed algorithm starts from an empty March Test and adds new elements, such as March Elements or Memory Operations. If the added elements do not improve the coverage then they are dropped. The process is repeated until the test reaches the fault coverage required or a test length limit. No information on test generation time is given.

[12] proposes a generation process able to deal with read disturbs faults and destructive read faults [13]. This approach is affected by the same problems of [6] [7].

Our previous works [14][15] present a generation algorithms able to deal with both classic static faults and new user defined static faults. The memory is modelled by using a graph called Test Pattern Graph (TPG). From the TPG a sequence of operations with minimum length is extracted by solving an instance of the Asymmetric Travelling Salesman Problem [16]. The sequence of operations is then translated into a March Test by applying a set of rewrite rules. The main drawback is the ATSP complexity (NP) that reduces the number of total faults that can be included in the fault list. To our best knowledge no Automatic Test Generations for Dynamic Faults has so far been proposed.

3. Test Generation Methodology

Our test generation methodology relies on a formal model representing the memory behaviour based on directed graphs (see Section 3.1), and on the definition of Functional Faults in terms of Fault Primitives (FPs) (see Section3.2). FPs are translated to an "*operational*" representation of the faulty behaviours, referred to as *Addressed FPs*, or AFPs (see Section 3.3). AFPs are represented on the graph modelling the memory as additional arcs and the resulting graph is traversed to generate the test (see Section 4). Although extracting the final solution is an NP complete problem, an efficient implementation has been found, profitably exploiting pruning conditions imposed by the goal of primarily generating March Tests. The overall generation methodology is summarized in Figure 1.

Test Generation {
1. To define the Functional Faults in terms of FP's
2. To build the digraph modelling the memory (number of nodes is equal to $2^{\#\text{ of involved cells}}$)
3. To express the FP's in terms of AFP's
4. To expand the digraph according to AFP's
5. To traverse the digraph to find the test

}

Figure 1. Automatic Test Generation Methodology

3.1. Memory Model

The March Test generation process starts from a formal definition of the memory model. The problem of modeling memory behaviors has already been faced in [14] and [15] adopting a behavioral model based on Finite State Machines (FSM). An n one-bit cells memory is represented by a deterministic Mealy Automata formally defined as follow:

$$M = (Q, X, Y, \delta, \lambda) \quad (1)$$

where:

- $Q = \{(0,1,-)^n\}$ is the set of possible memory states;
- $X = \{r^i, w^i_d \mid 0 \le i \le n\text{-}1, d \in (0,1)\} \cup \{T\}$ is the *input alphabet*, composed by all the possible memory operations:
  - r^i: a read operation performed on the cell i ;
  - w^i_d: a write operation of the value d with $d \in (0,1)$ performed on the cell i;
  - T : a wait operation for a defined period of time. This additional element is needed to deal with *Data Retention Faults* [7]
- $Y \in \{0,1,-\}$ is the *output alphabet*, composed of the value '0' or '1' obtained by performing a read operation. '-' denotes the value obtained when a write operation is performed;
- $\delta = Q \times X \rightarrow Q$ is the *state transition function;*
- $\lambda = Q \times X \rightarrow Y$ is the *output function.*

We can represent (1) as a labeled directed graph:

$$G = \{V, E\} \quad (2)$$

where:

- V is the set of vertices: $|V| = 2^n$ and each vertex represents one of the possible states of the memory
- E is the set of edges: each edge represents one of the possible memory operations that cause the transition form a vertex u to a vertex v, labeled with input and output alphabets.

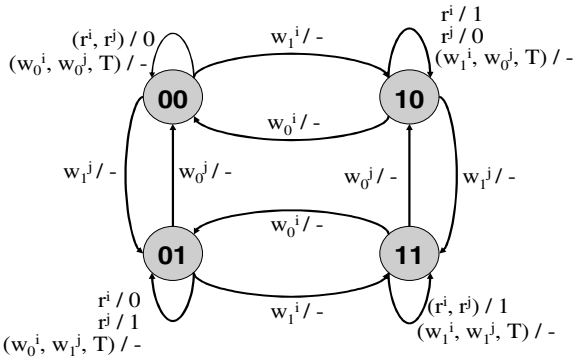

Figure 2: Fault Free Memory G0

As an example, Figure 2 conventionally named G_0. In G_0 the letters i and j are used to identify the first and the second cell respectively. Hereinafter, we shall assume $i < j$.

3.2. Fault Model

To represent memory faults we use the Fault Primitive formalism introduced in [17] and here summarized for sake of completeness. A *Fault Primitive* FP is a 4-triplet representing the difference between an expected (good) and the observed (faulty) memory behavior [17]:

$$< S\ ;\ I\ /\ F\ /\ R > \qquad (3)$$

where:

- S describes the *Sensitizing Operations Sequence (SOS)*, i.e., the shortest sequence of operations, performed on the aggressor cells, needed to sensitize the fault. $S \in \{\text{op}\}^m$, m being the minimum number of operations required to sensitize the fault[1]. Each *op*, in turn, can be represented as:
  $$\text{c}(i\text{O}d) \qquad (4)$$
  where:
  - c is the address of the memory cell. If omitted means any cell of the memory;
  - i is the initial value stored in the cell c, $i \in \{0,1\}$;
  - O is the performed operation on c, $O \in \{w,r\}$;
  - d is the value written in c in case of write operation, $d \in \{0,1\}$.
- I is the value (state) stored in the victim cells before applying S. When k victim cells are involved, $I \in \{0,1\}^k$;
- F is the value (state) stored in the victim cells after applying S. When k victim cells are involved, $F \in \{0,1\}^k$;
- R is the sequence of values read on the aggressor cell when applying S. $R \in \{0,1,-\}$, $R =$ '-' is used when a write operation sensitizes the fault.

[1] Functional Faults are usually classified as *Static* when $m \leq 1$ as *Dynamic* elsewhere (see Section 3.3)

As an example, an Inversion Coupling Fault (CF_{in}) involving two memory cells in such a way that a transition performed on the aggressor cell causes the inversion of the value stored in the victim cell can be described by the following two FPs:

$$FP1 = <0\text{w}1;0/1/->\ \text{and}\ FP2 = <0\text{w}1;1/0/-> \qquad (5)$$

Definition 1: A *Functional Fault model* is a non-empty set of fault primitives.

As pointed out in [17] and [3], the Static Fault Set, has cardinality 48. On the other hand, the cardinality of the Dynamic Fault Set (DFS) is infinite, the number of SOSs being not upper limited. The DFS is usually split in subsets, each including the DFs requiring the same number of operations to be sensitized. As an example, the 2-operations DFS includes 126 FPs. In the sequel of the paper, for sake of readability, we shall deal with 2-operations DFS only, even if the proposed approach can deal with a generic p-operations DFS, as well.

3.3. Faulty Memory Model

The behavior of a faulty memory can be modeled by extending the model defined in (2), adding a set of additional edges, derived from the FPs definition.

In the FP notation, each FP_i describes a faulty behavior involving k_i memory cells with $k_i = a_i + v_i$ where a_i is the number of aggressor cells and v_i is the number of victim cells. In the final model the number of states Q, and thus the number of nodes V, will be $2^{\max(k_i)}$ with $0 < i \leq \#FP$.

As an example, the FPs defined in (5) involve two cells, thus they require a 4 states memory model.

Since the FP notation not necessarily include the address of both aggressor and victim memory cells, we extend it by introducing the *Addressed Fault Primitive* concept used to add the additional arcs on the memory model.

Definition 2 : An *Addressed Fault Primitive (AFP)* is a representation of a FP as a sequence of memory operations specified with their target memory cells (0,1, …,n-l), formalized as:

$$AFP = (I,E,O) \qquad (6)$$

where:

- $I = \{(0,1)^l \mid 0 \leq l \leq k\text{-}1\}$ is the initialization state;
- $E = \{(e)^* \mid e \in X^*\}$ is the list of operations needed to excite the faulty behavior; E correspond to the SOS;
- $X^* = X \cup \{r^l_d\}$, where r^l_d is the operation needed to observe the fault effect. The notation r^l_d means "*read the content of the cell l and verify that its value is equal to d*".

The AFP formalism strictly depends on both the number k of memory cells involved in the fault and the number n of memory cells used by the memory model G. Since n not necessary corresponds to the number of

memory cells involved in each FP, it may happen that two or more AFPs have to be derived from a single FP.

As an example, let's consider the Transition Faults [18] that involves one cell that fails to undergo a transition (from 0 to 1). The FP modeling this fault is <0w1/0/->. Suppose to have a fault free memory model G_0 with 2 cells (i.e., 4 states), the resulting AFP's will be: $AFP_1 = (00, w^i_1, r^i_1)$, $AFP_2 = (00, w^j_1, r^j_1)$, $AFP_3 = (01, w^i_1, r^i_1)$, $AFP_4 = (10, w^j_1, r^j_1)$.

Moreover, when converting a FP modeling a two-cell fault[2] we have to consider both the case when the address of the aggressor cell is grater than the address of the victim cell and viceversa. For example the FP <0w1,0/1/->, that involves two cells, is represented by two AFP's: $AFP_1 = (00, w^i_1, r^j_0)$ and $AFP_2 = (00, w^j_1, r^i_0)$. In AFP1 the address of the aggressor cell is *i* (on the memory model G_0) and the address of the victim cell is *j* ($i < j$), and viceversa in AFP2.

Each AFP is represented on the memory model as an additional arc labelled with two elements:

- The operation that sensitize the fault that can be a single read, a write operation (static fault) or a list of two or more operations (dynamic fault);
- The operation used to observe the fault effect, that can only be a read operation.

The resulting memory model is still represented as a *directed graph G* (digraph) containing two categories of arcs:

- *Normal Arcs* (NA), representing the behavior of the fault free memory;
- *Faulty Arcs* (FA), representing the AFPs.

Let's consider as an example the Inversion CF. It is described by the following FPs: FP_1 = <0w1 ,0/1/-> and FP_2 = <0w1, 1/0/->. Resorting to the AFPs notation FP1 and FP2 are rewritten as $AFP_1=(00,w^i_1,r^j_0)$, $AFP_2=(00, w^j_1,r^i_0)$, $AFP_3=(01,w^i_1,r^j_1)$, $AFP_4=(10,w^j_1,r^i_1)$.

The resulting digraph is in Figure 3, where the four bolded arcs represent the AFPs.

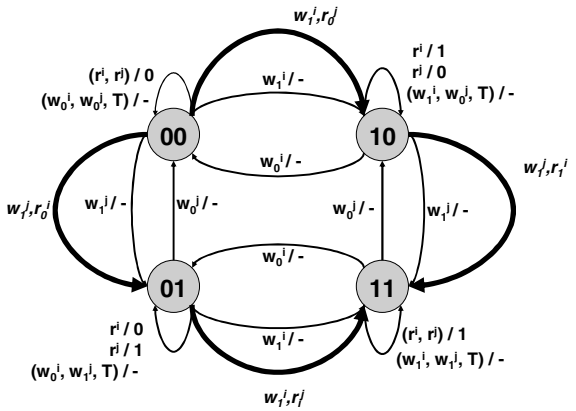

Figure 3. CFin memory model G

[2] A Fault that involves two memory cells

4. March Test Generation Algorithm

The March Test generation process starts from the model introduced in Section 3.3 representing the memory and all the possible FPs. The generation of a test algorithm consists in finding a sequence of operations able to sensitize and observe each FP in the fault list, performing the set of possible memory operations defined by the memory model.

Looking at the memory model of Section 3.3 this means we have to traverse each FA in order to find a sequence of operations able to cover our fault list. In addition, to obtain tests with minimum length the sequence must be as shortest as possible; thus we have to find constraints to reduce the length of our sequence of operations. A possible constraints is to traverse each FA exactly once.

Our problem is an instance of the *Rural Postman Problem* [19] [21]. The RPP is the practical extension of the well-known *Chinese Postman Problem* [16], members of the wider field of the *combinatorial optimization (CO) problems.* In a general instance of the CPP, we have to find a minimum length closed *walk* that traverses each edge of the graph at least once; whereas in the rural postman problem only a subset of the edges are required to be traversed at minimal cost. Finding an optimal solution in a graph with both *undirected* and *directed* edges is *NP-Complete* [19].

We introduced in Section 3.3 the directed graph $G = \{V, E\}$, representing the memory model, where E can be rewritten as: $E = FA \cup NA$.

Our problem can be formalized as follows: given a directed graph $G = \{V, E, FA\}$ with V representing the set of nodes, E representing the set of edges (memory operations), and FA ($\subseteq$ E) representing the set of edges to be traversed we have to find a walk, starting from a given node (memory state), traversing each required edge (FA) only once, and ending in a given node (open path).

Note that the solution admits that the NAs can be traversed more than once or never traversed. The optimal solution should also minimize the number of time that a generic NA is traversed.

4.1. Pruning conditions

Without introducing additional constraints, this problem is not manageable in case of complex fault lists. Nevertheless, the complexity can be significantly reduced if we consider that the final test algorithm we want to generate is a March Test. March algorithms apply the operations in a specific way.

Definition 3 : A *March Test* is a sequence of March Elements; each March Element is a sequence of memory operations applied on every cell in a specific address order (increasing, decreasing, random).

Definition 4 : The *Initial (Final) State* is the logical values stored in the memory cells before (after) the March Test application**.**

These definitions allow us to introduce a set of constraints in our problem that significantly reduce the space of the possible solutions. The problem is now to find a sequence of operations able to cover all the faults in the fault list while respecting the March Test constrains. Basically the March Test definition introduces the following two constraints:

1. **Final Memory State**: since from the definition of March Test, each sequence of operations must be applied on all the cells of the memory, the only possible final state for the memory is the one with all the cells initialized to '0' or all the cells initialized to '1'.
2. **March Element**: a *Sequence of Operations* SO in the final algorithm is *valid* if and only if all the operations are performed on the same memory cell *i*; otherwise the sequence causes a constraint violation.

In some cases it is possible to obtain a set of redundant AFPs modeling a single FP. A typical case is when the memory model involves a number of cells greater than the memory cells involved in the FP. As an example if we consider the FP representing the Transition Fault introduced in Section 3.3 we obtain four AFPs. Since march element operations are applied on each memory cell it is enough to consider just one of the redundant AFPs. The model can be extended by grouping AFPs into classes of equivalence.

Definition 6 :Two or more AFPs are *equivalent* if only one of them is necessary to cover a given fault.

Definition 7 : The *Address Specification* of a valid SO is the memory address on which the operations composing the sequence are performed.

A valid SO can be directly translated into a March Element by specifying its address order and by removing the address specification. Considering the G_0 memory model of Section 3.1 ($i < j$), the march element address order is defined as follow:

- If the address specification of a valid SO is equal to *i* then the address order will be '⇑'
- If the address specification of a valid SO is equal to *j* then the address order will be '⇓'

An AFP is compatible with an exisiting valid *SO* if its address specification is equal to the address of the sequence.

If the address of an AFP is not unique, that AFP cannot be covered by a March Test because the sequence of sensitizing operations of the AFP cannot be part of the same March Element.

The algorithm works on the graph representing the memory model. It attempts to generate a set of March Elements by building *valid* sequences of operations. It mainly consists in finding a path on the graph able to touch each faulty arc exactly once while respecting the March Test constraints. The main steps of the algorithm are summarized in Fig. 4.

1. Report (and remove) all the AFP which address is not unique
2. **Repeat**
   a. Initialize the Sequence of Operations (SO=∅)
   b. **While** (the next AFP is compatible with SO)
      i. Put the AFP into the Sequence of Operations SO
      ii. Delete the AFP and each AFP belonging to the same equivalence class
   c. **If** (The Sequence of Operations contains at least one AFP) **then**
      i. Apply the Sequence of Operations to each memory cell[3]
      ii. **If** (new AFPs are covered) **then** delete the covered AFPs and each AFP belonging to the same equivalence class
      iii. Translate the Sequence of Operations into a March Element by setting its address order
      iv. Print the March Element
   d. **Else**
      i.Report that the AFP cannot be cover by the March Test
3. **Until** (AFP list is empty)

Figure 4: March Test Generation Algorithm

5. Experimental results

This section reports some experimental results obtained by applying the proposed generation algorithm to different fault lists. The algorithm has been implemented in about 900 lines of C++ code, compiled with *gcc* compiler. All the experiments are performed on an *ASUS, AMD 1500Mhz* based Laptop with 512 MB of RAM. Table 1 reports the March Tests generated for different sets of target faults. We have been able to generate most of the already published March Tests, which have already been proved to be the best ones. In addition we have been able to generate a new March Test never published before. Table 2 reports the complexity of the algorithms and the CPU time, expressed in seconds, needed to generate the March Tests. Finally, we applied our algorithm to the complete set of two operations dynamic faults published in [17], obtaining the 100*n* March Test of Figure 5. All generated March Tests have been verified using a memory fault simulator able to validate their correctness w.r.t. the target fault list. The fault simulator has been also used to check the non-redundancy of each generated March Test [5].

{
1. M1:⇔(*w*1)
2. M2:⇓(*r*1*r*1*r*1*w*1*r*1*w*1*r*1*r*1*w*1*w*1*r*1*w*0*r*0*r*0*r*0*w*0*r*0*w*0*r*0*r*0*w*0*w*0*r*0*w*1)
3. M3:⇓(*r*1*w*0*r*0*r*0*w*1*r*1*r*1*w*1*w*0*r*0*w*0*w*1*r*1*w*0*w*1*r*1*w*0*w*0*r*0*w*1*w*0*r*0*w*1*w*1)
4. M4:⇑ (*r*1*w*0)
5. M5:⇓ (*r*0*r*0*r*0*w*0*r*0*w*0*r*0*r*0*w*0*w*0*r*0*w*1*r*1*r*1*r*1*w*1*r*1*w*1*r*1*r*1*w*1*w*1*r*1*w*0)
6. M6:⇓(*r*0*w*0*w*1*r*1*w*1*w*0*r*0*w*1*w*0*r*0*w*1*w*1*r*1*w*0*w*1*r*1*w*0*w*0*r*0*w*1*r*1*r*1*w*0*r*0)
7. M7:⇔(*r*0)

}

Figure 5. 2-Operations DF March Test, CPU time 0.943s

[3] E.g., if the address of the Sequence of Operations is *i*, now we try to apply the same sequence to the cell *j* and so on …

Type	March Test	Algorithm	Fault List	Ref.
Static	MATS	{⇑ w1 ⇓ r1 w0 ⇓ r0}	SAF	[4]
Static	MATS+	{⇑ w1 ⇑ r1 w0 ⇓ r0 w1}	SAF,ADF	[2]
Static	MATS++	{⇑ w_0 ⇑ r_0w_1 ⇓ r_1w_0 ⇑ r_0}	SAF,TF,ADF	[2]
Static	March C-	{⇑ w1 ⇑ r1 w0 ⇑ r0 w1 ⇓ r1 w0 ⇓ r0 w1 ⇓ r1}	SAF,TF,ADF,CFid,CFinv	[4]
Static	CLI	{⇑ w_1 ⇑ r_1 w_0 w_1 ⇓ r_1 }	CFinversion	[14]
Static	March SS	{⇑w0 ⇑r0r0w0r0w1 ⇑r1r1w1r1w0 ⇓r0r0w0r0w1⇓r1r1w1r1w0 ⇑r0}	All static fault	[20]
Dynamic	RAW1	{⇑w1 ⇓w1r1 ⇓r1w0r0 ⇓r0w0r0 ⇓r0w1r1 ⇑ r1}	Single Dynamic Faults	[20]
Dynamic	RAW	{⇑w1⇓r1w1r1r1w0r0⇓r0w0r0r0w1r1⇑r1w1r1r1w0r0⇑r0w0r0r0w1r1⇑r1}	dCFds,dCFdrd	[20]
Dynamic	unknown	{⇑w1 ⇓w0r0w1r1r1}	Dynamic read fault	

Table 1. Generated March Tests

March Test	O(n)	CPU Time (s)
MATS	4*n*	0.030
MATS+	5*n*	0.028
MATS++	6*n*	0.210
March C-	10*n*	0.204
CLI	5*n*	0.093
March SS	22*n*	0,201
RAW1	*13n*	0.212
RAW	26*n*	0.302
Unknown	*6n*	0.097

Table 2. March Tests Complexity

6. Conclusions

This paper presented a methodology to automatically generate March Tests. A formal model has been used to represent both known memory faults, and to possibly add new user-defined faults. The methodology is able to deal with both classic static faults and new complex dynamic faults. With respect to previously presented approaches our methodology allows generating non-redundant March Tests in a very low computation time, and without exhaustive searches. We have been able to generate March Tests for the complete set of known Static Faults and for most of the known dynamic faults obtaining both already published and new test algorithms. On going activities are focused on the extension of the model to multi-port memory faults and to linked fault models.

7. References

[1] International Technology Roadmap for Semiconductors, "*International technology roadmap for semiconductors 2004 Update*", http://public.itrs.net/Home.htm, 2004

[2] A. J. van de Goor, "Testing Semiconductor Memories: theory and practice", *Wiley, Chichester (UK)*, 1991

[3] S. Hamdioui, R. Wadsworth, J.D. Reyes, A.J. van de Goor, "Importance of Dynamic Faults for new SRAM Technologies", *ETW 2003, 8th IEEE European Test Workshop*, 2003, pp.29-34

[4] A. J. van de Goor, "Using March Tests to Test SRAMs", *IEEE Design & Test of Computers*, 1993 pp: 8–14.

[5] A. Benso,S. Di Carlo,G. Di Natale,P. Prinetto "Specification and design of a new memory fault simulator", *ATS 2002, 11th IEEE Asian Test Symposium*,2002.pp.92–97.

[6] A.J. van de Goor, B. Smit, "Automatic verification of March Tests", *MTDT 1993, IEEE International Workshop on Memory Technology, Design and Testing*, 1993.pp.131–136.

[7] A.J. van de Goor, B. Smit, "The automatic generation of March Tests", *MTDT 1994, IEEE International Workshop on Memory Technology Design and Testing*,1994, pp.86 -91.

[8] S.M. Al-Harbi, S.K. Gupta, "An efficient methodology for generating optimal and uniform March Tests", *VTS 2001, 19th IEEE VLSI Test Symposium*, 2001, pp. 231 -237

[9] S.M. Al-Harbi, S.K. Gupta, "Generating complete and optimal March Tests for linked faults in memories", *VTS 2003, 21st IEEE VLSI Test Symposium*, 2003, pp. 254 -261.

[10] K. Zarrineh, S.J. Upadhyaya, S. Chakravarty, "A new framework for generating optimal March Tests for memory arrays", *ITC 1998, IEEE International Test Conference*, 1998, pp.73 – 82.

[11] K-L. Cheng, C-W. Wang, J-N. Lee, Y-F. Chou, C-T. Huang; C-W. Wu, "Fault simulation and test algorithm generation for random access memories", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2002 pp. 480 – 490.

[12] D. Niggemeyer, E.M. Rudnick, "Automatic Generation of Diagnostic Memory Tests Based on Fault Decomposition and Output Tracing", *IEEE Transactions on Computers*, . 2004, pp. 1134–1146.

[13] R.D. Adams and E.S. Cooley, "Analysis of a Deceptive Destructive Read Memory fault Model and Recommended Testing", *NATW 1996. 5th IEEE North Atlantic Test Workshop*, 1996

[14] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "Memory read faults: taxonomy and automatic test generation", *ATS 2001, 10th IEEE Asian Test Symposium,* 2001. pp. 157 – 163.

[15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto "An optimal algorithm for the automatic generation of March Tests", *DATE 2002, IEEE Design, Automation and Test in Europe Conference and Exhibition*, 2002. pp. 938 – 939

[16] A. Gibbons, "Algorithmic Graph Theory", *Cambridge University Press* 1985.

[17] A. J. van de Goor, Z. Al-Ars, "Functional Memory Faults: A Formal Notation and a Taxonomy", *VTS 2000, 18th IEEE VLSI Test Symposium*, 2000, pp. 281-289.

[18] R. Dekker, F. Beenker, L. Thijssen, "A Realistic Fault Model and Test Algorithms for Satic Random Acces Memory",*IEEE Transaction on Computer-Aided Design*, 1990

[19] N.Christofides, V.Campos, A.Corberán, E.Mota "An Algorithm for the Rural Postman Problem on a directed graph", *Mathematical Programming Study*,1986, pp.155-166

[20] S. Hamdioui, Z. Al-Ars, A J. van de Goor, "Testing Static and Dynamic Faults in Random Access Memories", *VTS 02, 20th IEEE VLSI Test Symposium*, 2002. pp.395-400.

[21] W. L. Pearn, C. M. Liu,"Algorithms for the Chinese Postman Problem on Mixed Networks", *Computers & Operations Research*, Volume: 22, 1995, pp. 479-489.