An Optimal Algorithm for the Automatic Generation of March Tests

A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto
Politecnico di Torino
Dipartimento di Automatica e Informatica Torino, Italy
E-mail: {benso,dicarlo,dinatale,prinetto}@polito.it
Web: www.testgroup.polito.it

Abstract

This paper presents an innovative algorithm for the automatic generation of March Tests. The proposed approach is able to generate an optimal March Test for an unconstrained set of memory faults in very low computation time.

1. Introduction

Among the different types of algorithms proposed to test random access memories (RAM), March Tests have proven to be faster, simpler, regularly structured and linear in complexity. A March Test consists of a sequence of *March Elements*, each composed by a sequence of basic read/write operations to be performed on each cell of the memory, in either ascending or descending order, before proceeding to the next memory cell. The complexity of a March Test is given by the number of memory operations in all March Elements performed on each memory cell [1].

March Tests are able to cover a wide range of memory faults such as Stuck-at-Faults, Transition Faults, Stuck-Open Faults, Coupling Faults, Address Fault and Data Retention Faults. Different March Tests of variable complexity have been proposed in literature, each optimally covering a different set of memory faults. All of them have been manually generated, a task that always requires a lot of time, expertise, that do not always allows to obtain an optimal solution, and that sometimes do not succeeds in covering particularly complex memory faults.

This paper presents a methodology to automatically generate March Tests. A general representation is used to model known memory faults, and to possibly add new user-defined faults.

With respect to previously proposed approaches, which exhaustively generate all the possible March Tests and then select the optimal one, our approach allows generating the optimal March Tests in a very low computation time without exhaustive searches. In particular, the automatic March Test generation process is performed in the following steps: (i) the target memory fault list is modeled into a set of FSMs representing the faulty memory behaviors; (ii) a weighted graph is generated, which represent all the possible test patterns able to cover each target fault model; (iii) an optimal test sequence is generated finding an optimal path connecting all the nodes of the graph; (iv) from the so defined optimal sequence, a minimal March Test is derived applying a set of linear complexity transformations.

The paper is structured as follows: Section 2 presents the model used to represent the good and fault memory behavior. Section 3 summarizes the state of the art, whereas Section 4 details all the steps of the automatic March Test generation process. Section 5 analyzes a possible optimization of the algorithm, and Section 6 presents experimental results that proof the efficiency of our approach. Section 7 summarizes the main contributions and future developments of this research.

2. State of the Art

The problem of the automatic generation of March Tests has been already faced and several publications can be found in literature. [2] [3] [4] 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 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, when dealing with undetectable faults, the computation time becomes infinite. In addition, 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 [5] 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 will be found much faster and more efficiently.

An approach able to cover additional faults is presented in [6]. With respect to the previously mentioned works, it is able to deal with read disturb faults and destructive read faults. It mainly targets the diagnosis of memory faults and utilizes a fault description that allows modeling all possible single cell and two cells faults that occurs in memory arrays. This approach still uses exhaustive search and is affected by the same problems of [2] [3] and [4].

3. Memory Model

The problem of the automatic generation of March Tests requires, first of all, the definition of a formal model able to represent the behavior of both the good and the faulty memory. In [7] and [8] the problem has been solved proposing a memory behavioral model based on Finite State Machines (FSM). An *n* one-bit cells memory can be represented using a deterministic Mealy Automata:

$$M = (Q, X, Y, d, l) \quad \text{(f.2.1)}$$

where:

- $Q = \{(0,1,-)^n\}$ is the set of the possible *memory states* where the symbol $(-)$ represents the value of a non initialized memory cell;
- $X = \{r^i, w_0^j, w_1^j \mid 0 \le i \le n-1\} \cup \{T\}$ is the *input alphabet*. This alphabet is composed by all the possible memory operations. In particular:
  - r^i corresponds to a read operation performed on the cell *i*;
  - w_d^i corresponds to a write operation of the value $d \in \{0,1\}$ performed on the cell *i*;
  - *T* corresponds to a wait operation for a defined period of time. This additional element is needed to deal with *Data Retention Faults* [2].
- $Y = \{0,1,-\}$ is the *output alphabet*;
- $d : Q \times X \mapsto Q$ is the *state transition function*;
- $l : Q \times X \mapsto Y$ is the o*utput function*.

Using the proposed model, a fault free two cells RAM can be represented by the FSM shown in Figure 1, conventionally named M_0 in the reminder of this paper. In M_0, the letters *i* and *j* are used to identify the first and the second cell, respectively.

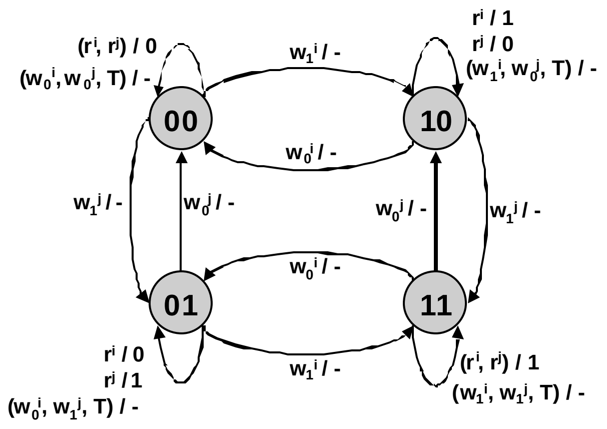

Figure 1: *M0 FSM representing a fault free RAM*

The proposed model is not manageable when used to represent large memories; nevertheless, the model of a two-cell memory is general enough to model memory faults. Therefore, the behavior of a faulty memory can be modeled using a deterministic Mealy Automata:

$$M_i = (Q_i, X, Y_i, d_i, l_i) \quad \text{(f.2.2)}$$

where:

- $Q_i \subseteq Q$ is the set of states;
- $Y_i \subseteq Y$ is the output alphabet;
- $d_i : Q_i \times X \mapsto Q_i$ is the state transition function
- $l_i : Q_i \times X \mapsto Y_i$ is the output function

The set of states used to represent a faulty memory is a subset of the whole set Q (see (f.2.1)) since only the cells involved in the fault should be represented. This consideration makes possible the use of the proposed model for very large memories as well. Moreover, the given representation for faulty memories is general enough to be used to model most of the known faults.

Considering as an example the Idempotent Coupling Fault $\langle \uparrow, 0 \rangle$ [9] (where the notation $\langle S, F \rangle$ denotes a fault involving two cells; S describes the condition of the first cell to sensitize the fault in the second cell denoted by F), we obtain the FSM shown in Figure 2. From now on, we will assume that the address of cell *i* is less then the address of cell *j*.

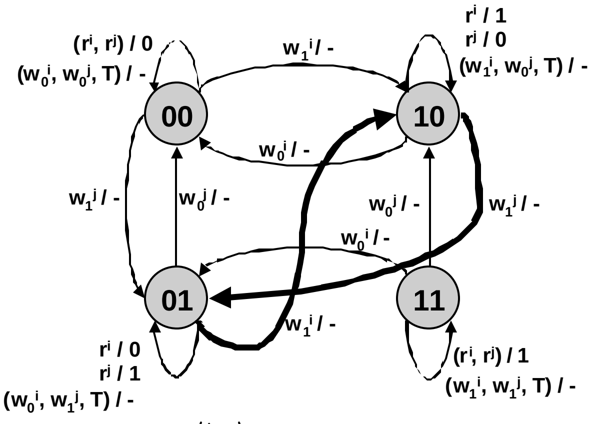

Figure 2: *M1:* $\langle \uparrow, 0 \rangle$ *Idempotent Coupling Fault*

As previously mentioned, since the fault involves two cells only, the cardinality of Q_i is four. The difference between the M_0 and M_1 machine is in the δ function, as pointed out by the two-bolded edge shown in Figure 2.

Looking at the M_1 machine we can split each fault into a set of *Basic Fault Effect (BFE)* [5] [6]. A BFE can be described by a M_i FSM with a d_i function that differs from d_0 by one transition only, or with a l_i function that differs from l_0 by one output value only. Considering the example proposed in Figure 2, it is possible to identify two different BFEs modeled by the two FSM shown in Figure 3. For the sake of simplicity only the relevant edges are represented.

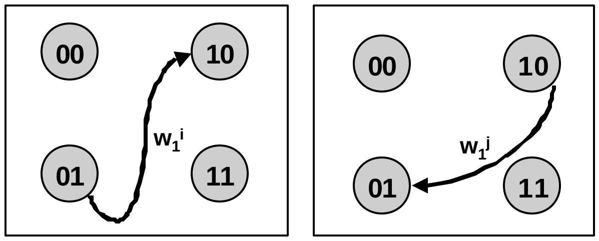

Figure 3: *BFE model for* $\langle\uparrow,0\rangle$ *Coupling Fault*

Each BFE can be covered generating a Test Pattern (TP) defined as a triplet:

$$TP = (I, E, O) \quad \text{(f.2.3)}$$

where:

- $I = \{(0,1)^k \mid 0 \le k \le n-1\}$ is the *initialization state*;
- $E = \{e \mid e \in X\}$ is the operation needed *to excite* the BFE;
- $O = \{r_d^k \mid d \in (0,1), 0 \le k \le n-1\}$ is the operation needed *to observe* the fault effect. We introduce here the concept of *Read and Verify* operation. The notation r_d^i means "read the content of the cell *i* and verify that its value is equal to *d*".

For the proposed example the two BFEs can be tested by the following two TPs:

- $TP_1 = (01, w_1^i, r_1^j)$
- $TP_2 = (10, w_1^j, r_1^i)$

4. March Test Generation Algorithm

In this section the proposed approach will be presented. It exploits the possibility of automatically generating March Tests without exhaustive searches. The algorithm, starting from an unconstrained list of target BFEs, generates a non-redundant march test that covers all of them.

In a first phase, the algorithm analyzes the set of test patterns needed to cover each target BFE, and generates a weighted graph named *Test Pattern Graph (TPG)*. Each TPG node is associated to a TP. The graph is strongly connected, i.e. each node is connected to all the others.

The *weight* of each edge represents the number of memory operations needed to reach the initialization state of the target node (S_T), starting from the observation state of the source node (S_S). In a formal way it can be defined as [10]:

$$\textit{weight} = \textit{hamming-distance}\ (S_S, S_T) \quad \text{(f.4.1)}$$

Lets consider as an example the $FaultList = \{\langle\uparrow,1\rangle,\langle\uparrow,0\rangle\}$ [9]. Using the model proposed in Section 2 we obtain four different BFEs, respectively tested by the following set of TPs:

- $TP_1 = (01, w_1^i, r_1^j)$
- $TP_2 = (10, w_1^j, r_1^i)$
- $TP_3 = (00, w_1^i, r_0^j)$
- $TP_4 = (00, w_1^j, r_0^i)$

The proposed set of test patterns generates the TPG shown in Figure 4.

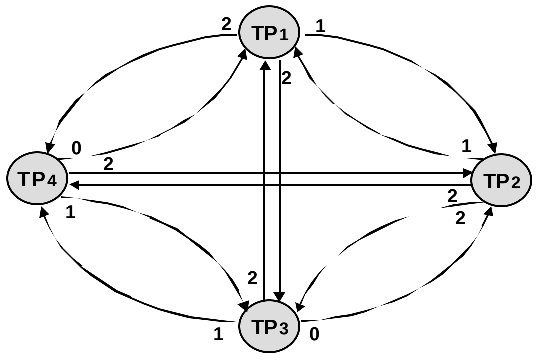

Figure 4: *TPG for* $\{\langle\uparrow,1\rangle,\langle\uparrow,0\rangle\}$

Starting from the TPG the algorithm extracts a so-called *Global Test Sequence (GTS)*. A GTS is a set of memory operations able to detect all the target BFEs. Different GTSs can be obtained by simply concatenating all the different TPs in multiple ways, i.e., to make different visits of the TPG. Since the TPG is strongly connected, the total number of possible GTS can be calculated as follow:

$$\#GTS = V! \quad \text{(f.4.2)}$$

where *V* is the number of nodes in the TPG.

In a fault-list containing a large amount of BFEs, the space of all the possible GTS becomes unmanageable. It is therefore necessary to identify a particular subset of GTSs able to generate non-redundant March Tests. A possible solution is to consider TPG visits with minimum weight only. Thanks to the function used to weight the TPG edges (see (f.4.1)), these visits generate GTSs with minimum number of test operation. Considering two nodes connected by a 0 weight edge, the test sequences obtained by their concatenation does not need the initialization part of the second TP.

The use of GTSs with minimum number of operation seems a good choice since there is a strict correlation between the GTS length and the March test complexity.

The generation of minimum length GTSs is a typical instance of the *Asymmetric Traveling Salesman Problem (ATSP)*[11]. The ATSP is probably the most well known member of the wider field of the *combinatorial optimization problem.* In a general instance of the ATSP, one is given V nodes and a matrix $d_{i,j}$ storing the distance or cost function to go from node i to node j . A "tour" consists of a list of V nodes, (*tour[i]*) where each node appears once and only once . In the ATSP, the problem is to find the tour with the minimum length, where the length is defined to be the sum of the lengths along each step of the tour,

$$length = \sum_{k=0}^{V-1} d_{tour[k],tour[k+1]} \quad \text{(f.4.3)}$$

and *tour[V]* is identified with *tour[0]* to make it periodic.

The main difference with respect to our problem is that the solution of the ATSP is a cycle whereas a GTS is identified by a non-cyclic path (i.e., the first and last node do not need to be the same). To solve the problem we introduced two dummy nodes used to close the cycle. Despite the ATSP is a NP-hard problem, several algorithms able to give an exact solution with very low computation time in problems with low number of nodes (50 nodes), can be found in literature [12].

The GTSs obtained by solving the ATSP problem are able to test all the addressed BFE but are not yet March Tests. A March Test is a particular Test Sequence respecting a set of conditions [1]. It is therefore necessary to apply a set of modification to transform a GTS into an equivalent March Test.

Before applying the modifications (defined in Section 4.1), it is possible to perform a further optimization. We observed that GTS starting with a “00” or “11” initialization state allow to obtain March Tests of the lowest possible complexity. This optimization, which allows reaching a minimal solution considering all the minimum length GTSs, can be expressed as an additional constraint in the ATSP:

$$\begin{aligned} & length = \sum_{k=0}^{V-1} d_{tour[k],tour[k+1]} \\ & s.t. \qquad \text{(f.4.4)} \\ & TP_{tour[0]} = (\{00,11\},...,...) \end{aligned}$$

Looking at the example of Figure 4, a possible ATSP solution is the following GTS:

$$GTS = w_0^i, w_0^j, w_1^i, r_0^j, w_1^j, r_1^i, w_0^i, w_0^j, w_1^j, r_0^i, w_1^i, r_1^j$$

The process of March Test generation from a GTS passes through three different steps:

- GTS reordering
- GTS minimization
- March Test Generation

Each step corresponds to a different set of *Rewrite Rules* [13]. Since a GTS can be considered as a string where each symbol is a memory operation, the rewrite rules can be effectively represented resorting to the *Regular Expression* formalism [14]. All the possible memory operations are defined by the X alphabet defined in (f.2.1).

For the sake of simplicity we define two subsets of instructions:

- $w = \{w_d^i, w_d^j\}$ is the set of possible memory write operations;
- $r = \{r_d^i, r_d^j\}$ is the set of possible memory read operations.

The regular expression formalism is extended introducing three new operators:

- *End Symbol Operator:* $\hat{s}$ marks the symbol *s* as not furtherly modifiable (*terminal symbol*);
- *Red Operator:* $[s]_R$ marks the symbol *s* with the red color;
- *Blue Operator:* $[s]_B$ marks the symbol *s* with the blue color.

The use of colored symbols is useful during the March Test generation phase to identify the boundaries of the different March Elements. The next subsections summarize the rewrite rules used during the three different phases.

4.1 GTS Reordering

The reordering phase reorders the GTS memory instructions taking into account the constraints needed to obtain a March Test [1]. In this phase each modification is defined by a *Pattern* and by a *Rewrite Rule* (see Table 1). The pattern is a regular expression that identifies all the strings on which the rewrite rule must be applied. The reordering process stops when all the GTS symbols are modified into terminal ones.

Pattern	***Rewrite Rule***
$(\hat{w} \mid \hat{r})* w_d^i w_d^j (w \mid r)*$	$w_d^i w_d^j \xrightarrow{M1} \hat{w}_d^i \hat{w}_d^j$
$(\hat{w} \mid \hat{r})* w_d^i w_d^i (w \mid r)*$	$w_d^i w_d^i \xrightarrow{M2} \hat{w}_d^i w_d^i$
$(\hat{w} \mid \hat{r})* w_d^i w_{\bar{d}}^j (w \mid r)*$	$w_d^i w_{\bar{d}}^j \xrightarrow{M3} \hat{w}_d^i w_{\bar{d}}^j$
$(\hat{w} \mid \hat{r})* \hat{r}_d^i \underbrace{(\hat{w}_d^i \mid \hat{w}_d^j \mid \hat{w}_{\bar{d}}^j)*}_{s1} \underbrace{(w_d^j \mid w_{\bar{d}}^j)*}_{s2} r_d^i (w \mid r)*$	$\hat{r}_d^i s_1 s_2 r_d^i \xrightarrow{M4} \hat{r}_d^i [\hat{r}_d^i]_R [s_1 s_2]_B$

Table 1: *Reordering Rewrite Rules*

Appling the reordering rules on the GTS in Section 4 we obtain the following reordered sequence:

$$GTS_R = \hat{w}_0^i, \hat{w}_0^j, \lfloor\hat{r}_0^i\rfloor_R, \lfloor\hat{w}_1^i\rfloor_B, \hat{w}_1^j, \hat{r}_1^i, \hat{w}_0^i, \hat{w}_0^j, \lfloor\hat{r}_0^j\rfloor_R, \lfloor\hat{w}_1^j\rfloor_B, \hat{w}_1^i, \hat{r}_1^j$$

4.2 GTS minimization

The minimization phase deletes redundant subsequences in order to reduce the sequence to the minimum set of absolutely necessary operations only. The rewrite rules applied in this phase consider the GTS starting from left to right (see Table 2). This phase is repeated until no further minimization can be applied. In this context the $ symbol is used to denote the end of the GTS and the color of the symbols (see Section 4) does not affect the application of the rules.

Rewrite Rules	
$\hat{w}_d^i \hat{w}_d^j \xrightarrow{R1} \hat{w}_d^i$	$\hat{r}_d^i \hat{r}_d^j \xrightarrow{R1} \hat{r}_d^i$
$\hat{w}_d^i \hat{w}_d^i \xrightarrow{R2} \hat{w}_d^i$	$\hat{r}_d^i \hat{r}_d^i \xrightarrow{R2} \hat{r}_d^i$
$\hat{r}_d^i \hat{w}_{\bar{d}}^i \hat{w}_d^i \hat{r}_d^j \hat{w}_{\bar{d}}^j \hat{w}_d^j \xrightarrow{R3} \hat{r}_d^i \hat{w}_{\bar{d}}^i \hat{w}_d^i \hat{r}_d^j$	
$\hat{r}_d^i \hat{w}_{\bar{d}}^i \hat{w}_d^i \hat{r}_d^j \hat{w}_{\bar{d}}^j \$ \xrightarrow{R3bis} \hat{r}_d^i \hat{w}_{\bar{d}}^i \hat{w}_d^i \hat{r}_d^j \$$	

Table 2*: Modification Rewrite Rules*

Appling the minimization rewrite rules on the reordered GTS_R (see Section 4.1) we obtain the following minimal sequence:

$$GTS_M = \hat{w}_0^i, \lfloor\hat{r}_0^i\rfloor_R, \lfloor\hat{w}_1^i\rfloor_B, \hat{r}_1^i, \hat{w}_0^i, \lfloor\hat{r}_0^j\rfloor_R, \lfloor\hat{w}_1^j\rfloor_B, \hat{r}_1^j$$

4.3 March Test Generation

This last phase uses the minimized GTS to generate a March Test. The input sequences are analyzed from left to right and the March Elements are generated according to the following rules:

- *Rule 1:* subsequences identified by $(\hat{w}_d^i \mid \hat{r}_d^j)(\hat{w}_d^j \mid \hat{r}_d^j)$ regular expression close a March Element and open a new one;
- *Rule 2:* subsequences identified by $\lfloor\hat{r}\rfloor_R (\lfloor\hat{w}\rfloor_B *)$ regular expression are joined in a single March Element despite they are executed on *i* or on *j*. The last blue marked operation closes the March Element.

The addressing order is generated using the following rules:

- *Rule 3:* March Elements starting with colored operation performed on *i* cells have addressing order $\Uparrow$;
- *Rule 4:* March Elements starting with colored operation performed on *j* cells have addressing order $\Downarrow$;
- *Rule 5:* March Elements starting with non-colored operations have addressing order $\Updownarrow$.

Applying the generation rules on the GTS_M (see Section 4.2) we obtain the following *8n* non-redundant March Test:

$$M = \Uparrow w_0 \Uparrow r_0 w_1 \Uparrow r_1 w_0 \Downarrow r_0 w_1 \Downarrow r_1$$

5. BFEs equivalence

In some cases it is possible to obtain a BFE modeling a fault already covered by another BFE. A typical case is the $\langle\uparrow,\updownarrow\rangle$ Inversion Coupling Fault [1]. It can be split into two BFEs tested by the following TPs:

- $TP_1 = (00, w_1^i, r_0^j)$
- $TP_2 = (01, w_1^i, r_1^j)$

Although two TPs are generated, only one of them is necessary to cover the fault. Therefore, the ATSP problem must be modified to take into account only the necessary test patterns. This goal can be achieved grouping the TPG nodes into equivalence classes (C_i).

In case of a TPG with k equivalence classes, using the $|C_i|$ notation to indicate the cardinality of the C_i class it is possible to generate $E = \prod_{i=0}^{k-1} |C_i|$ different TPG. On each one of the obtained graphs the ATSP problem must be solved identifying *E* possible GTS. The minimum length GTS is considered as the best one.

6. Experimental Results

This section reports some experimental results obtained applying the proposed algorithm to automatically generate March Tests to cover different sets of faults.

The algorithm has been implemented in about 5000 lines of C code. The ATSP has been solved using a Fortran code able to give exact solutions to the problem [12]. For each generated March Test, we report the computation time needed to generate it, its complexity, and the complexity of the equivalent March Test found in literature. All the experiments are performed on a *Compaqä Presario 17XL370, PIII 650Mhz* based Laptop with 128 MB of RAM. The source code has been compiled with the gcc C compiler and the g77 Fortran compiler [15].

Table 3 shows the March tests obtained to cover some combinations of Stuck-At Faults (SAF), Transition Faults (TF), Address Decoder Faults (ADF), and Inversion and Idempotent Coupling Faults (CFin and CFid). All generated March Tests have been verified using an ad hoc memory fault simulator able to validate their correctness w.r.t. the target BFE list. The fault simulator is also used to check the non-redundancy of each generated March Test.

Fault List					**Generated March Tests and their complexity**		**CPU Time(s)**	**Equivalent Known March Test**
SAF	**TF**	**ADF**	**CFin**	**CFid**				
•					$\{\Uparrow w_1 \Downarrow r_1 w_0 \Downarrow r_0\}$	4*n*	0.49	MATS (4*n*)
•		•			$\{\Uparrow w_1 \Uparrow r_1 w_0 \Downarrow r_0 w_1\}$	5*n*	0.53	MATS+ (5*n*)
•	•	•			$\{\Uparrow w_0 \Uparrow r_0 w_1 \Downarrow r_1 w_0 \Uparrow r_0\}$	6*n*	0.61	MATS++ (6*n*)
•	•	•	•		$\{\Uparrow w_0 \Downarrow w_1 \Downarrow r_1 w_0 \Uparrow r_0 w_1\}$	6*n*	0.69	MarchX (6*n*)
•	•	•	•	•	$\{\Uparrow w_1 \Uparrow r_1 w_0 \Uparrow r_0 w_1 \Downarrow r_1 w_0 \Downarrow r_0 w_1 \Downarrow r_1\}$	10*n*	0.85	March C- (10*n*)
			•		$\{\Uparrow w_0 \Uparrow r_0 w_1 w_0 \Downarrow r_{01}\}$	5*n*	0.57	Not Found

Table 3: *Experimental Results*

Each March test is split into *elementary blocks*. An elementary block is a portion of March Test composed by a fault excitation and a fault observation. These blocks are used to build a *Coverage Matrix* (CM). The row of the matrix represents the elementary blocks whereas the columns the target BFEs. A matrix cell is set to the value one if the corresponding elementary block is able to test the BFE represented by the column, otherwise is set to zero. A March Test is able to detect all the target BFEs if for each CF column exist at least one row containing a cell set to one. The March Test is non-redundant if all the matrix rows are needed to cover the target BFE.

This is a typical instance of the *Set Covering* problem applied on the CM matrix. The Set Covering finds the minimum number of CM rows needed to cover all the CM columns. If this number corresponds with the total number of rows, then the March Test can be considered non-redundant.

This approach has been successfully applied on all the March Tests shown in Table 1 and redundant blocks have never been found.

7. Conclusions

This paper presented a methodology to automatically generate March Tests. A general model has been used to represent known memory faults, and to possibly add new user-defined faults. With respect to previously presented approaches our methodology allows generating the optimal March Tests in a very low computation time, and without exhaustive searches.

The generation process is based on four steps: memory fault modeling, TPG generation, minimum length GTS search, and the application of a set of rewrite rules to transform the minimum length GTS in a March Test. Some preliminary experimental results have been presented in order to demonstrate the applicability and efficiency of the proposed approach.

On going activities are focused on the extension of the model to multi-port memory faults, and to more complex user-defined fault models.

8. References

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

[2] A. J. van de Goor, B. Smit, "*Generating March Tests Automatically*",IEEE International Test Conference, pp. 870-877, 1994

[3] A. J. van de Goor, B. Smit, "*Automatic the Verification of March Tests*",IEEE VLSI Test Symposium, pp. 312-318, 1994

[4] A. J. van de Goor, B. Smit, "*The Automatic Generation of March Tests*", IEEE International Workshop Memory Technology, pp. 86-91, 1994

[5] K. Zarrineh, S. J. Upadhyaya, S. Chakravarty, "*A New Framework for Generating Optimal March Tests for Memory Arrays*", IEEE International Test Conference, pp. 73-82, 1998

[6] D. Niggemeyer, M. Redeker, E. M. Rudnick, "*Diagnostic Testing of Embedded Memories based on Output Tracing*", IEEE International Workshop Memory Technology, pp. 113-118, 2000

[7] J.A. Brzozowski, H. Jurgensen "*A Model for Sequential Machine Testing and Diagnosis*" J. Electronic Testing: *Theory and Application*, Vol. 3, No. 3, pp. 219-234, August 1992

[8] J.A. Brzozowski, B.F. Cockburn "*Detection of Coupling Faults in RAMs*" J. Electronic Testing: *Theory and Application*, Vol. 1, No. 2, pp. 151-162, May 1990.

[9] A. J. van de Goor, "*Using March Tests to Test SRAMs*", IEEE Design & Test of Computers, Volume: 10 Issue: 1, March 1993 pp: 8 –14

[10] R. W. Hamming, "*Coding and Information Theory*", II Edition, Englewood Cliffs, NJ: Prentice-Hall 1986.

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

[12] G.Carpaneto, E. Dell'Amico, I. Toth, "*A Branch-and-Bound Algorithm for large scale Asymmetric Traveling Salesman Problems*", Technical Report, Modena University 1990, *ACM Collected Algorithms no. 750, 1994,* ftp://netlib2.cs.utk.edu/toms/index.html

[13] G. Rozemberg, "*Handbook of Graph Grammars and Computing by Graph Transformation*", Vol. I: Foundation, World Scientific, 1997.

[14] A.V. Aho, R. Sethi, J. D. Ullman, "*Compilers: Principles, Techniques and Tools*", Addison-Wesley, 1986.

[15] http://www.gnu.org GNU web site.