Efficient Test-Relaxation-Based Static Compaction Techniques for Combinational and Sequential Circuits

Final Report

Project No. FT-2004/07

by

Aiman H. El-Maleh and Sadiq M. Sait

Computer Engineering Department

College of Computer Science and Engineering

King Fahd of Petroleum & Minerals

Dhahran 31261, Saudi Arabia

Contents

List of Tables

List of Figures

List of Algorithms

ABSTRACT

Testing system-on-a-chip (SOC) involves applying huge amount of test data, which is stored in the tester memory and then transferred to the circuit under test (CUT) during test application. Therefore, practical techniques, such as compression and compaction, are required to reduce the amount of test data in order to reduce both the total testing time and the memory requirements of the tester. In this project, the problem of static compaction for sequential and combinational circuits is investigated.

ملخص الأطروحة

يتضمن فحص النظام على الشريحة تطبيق كمية ضخمة من بيانات الاختبار, التي يتم تخزينها في ذاكرة الفاحص (أداة الاختبار) ومن ثم تحويلها إلى الدائرة الالكترونية تحت الاختبار أثناء تطبيق الاختبار.
ولكي يتم تخفيض كلا من وقت الاختبار الكلي ومتطلبات ذاكرة الفاحص, نحتاج إلى تقنيات عالية, مثل الضغط والدمج. في هذا المشروع يتم دراسة مشكلة الدمج في الدوائر الثابتة والمتتالية.

Chapter 1

Progress Report

A circuit must be tested to guarantee that it is working and continues to work according to specifications. Such testing detects failures due to manufacturing defects. It can also detect many field failures due to aging, environmental changes, power supply fluctuations, and so on.

Electronic companies spend a considerable proportion of production cost and engineering resources on testing [4]. Therefore, although testing incurs a lot of effort, it is finally an important means to significantly reduce the overall cost. For example, while the actual material cost is only a negligible proportion of the product value, the cost of repair increases by a factor of 10 with each production stage [4]. Thus, it is much cheaper to reject several dies than to locate and exchange a defective chip in a complete system.

Currently, with the System-on-Chip (SoC) design paradigm, classical testing problems, such as compression and compaction, are attracting more attention. A SoC is constructed using pre-designed and pre-verified cores as building blocks (see Figure 1.1). In fact, system integrators can purchase cores from various vendors. This in turns creates a competitive environment where multiple vendors are trying to sell cores with similar functionality. A challenging problem in testing SoCs is to deal with the large amount of test data that must be loaded from the tester memory, transferred to the SoC, and applied to the individual cores. The amount of time required to test a chip depends on the size of the test data that needs to be transferred and the speed of the transfer operation, i.e., channel bandwidth. The cost of automatic test equipment (ATE) increases significantly with the increase in its speed, channel capacity, and memory size. Therefore, reducing the amount of test data and thus test time is a major challenge for the SoC industry.

The amount of test data can be reduced using compression and compaction. The objective of compression is to compress a given test set T_D to a much smaller test set T_E that is stored in the tester memory [5, 6, 7]. During test application, T_E is loaded from the tester memory and decompressed on the Chip Under Test (CUT)

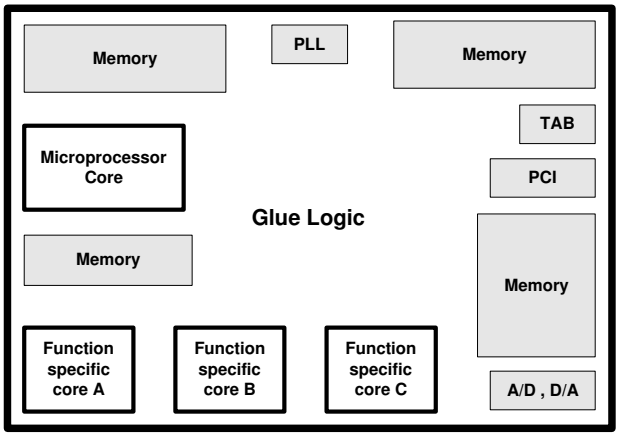


Figure 1.1: A typical SoC.

such that the original test set T_D is restored. Then, T_D is applied to the CUT. On the other hand, the objective of compaction is to reduce the number of test vectors without compromising fault coverage. In this work, we address the problem of static test set compaction for sequential and combinational circuits.

1.1 Introduction

This report covers in detail all the tasks accomplished during FT 2004/07 project. The report highlights the time duration of the reported period, tasks completed with the contribution of each investigator, budget utilization and list of equipment ordered for purchase, and difficulties faced.

1.2 Tasks Accomplished

Below we list all the tasks stated in the original proposal and what portion of it has been accomplished thus far.

Task 1: Modify the test relaxation algorithm for extracting a self-initializing test sequence for a set of faults. (Completed)

Task 2: Develop the Reverse-Order test vector restoration algorithm based on the self-initializing test extraction technique implemented in (Task 1). (Completed)

Task 3: Explore the possibility of reducing the size of the extracted self-initializing test sequences by considering the removal of the possibly redundant initializing sequence due to sequence concatenation. (Completed)

Task 4: Explore reducing the size of the extracted self-initializing test sequences by state traversal technique, based on inert and recurrent sub-sequence removal. (Completed)

Task 5: Develop the test sequence compaction technique based on Reverse-Order restoration of faults and subsequence merging. (Completed)

Task 6: Experiment with the developed techniques to explore the possibility of developing an algorithm that achieves the best compaction level than any of the techniques separately. (Completed)

Task 7: Implement the proposed test vector clustering technique based on the frequency of fault detection and compare it to IFC. (Completed)

Task 8: Increase the size of Class 0 test vectors by attempting to eliminate the conflicting components of a fault and by generating alternative components from other test vectors detecting the fault. (Completed)

Task 9: Improve the efficiency of blockage value computation for Class 0 processing. (Completed)

Task 10: Explore more efficient heurestics for processing of class 0 test set. (Completed)

Task 11: Document the developed software. (Completed)

Task 12: Generate periodical project reports and author publications for conferences/journals. (Published, In progress)

1.2.1 Task Achievement Diagram

Referring to Table 1.1, we reported the completion of Tasks 1-3 in first progress report (June 2004 - December 2004). Test Relaxation algorithm was modified to return self-initializing subsequences for a group of faults (Task 1). This was then used to implement Reverse-Order-Restoration (ROR) algorithm (Task 2). The restored self-initializing subsequence is further reduced by removing test vectors that take the circuit to states, which are already reached by previously restored subsequences, therefore do not contribute to fault detection (Task 3).

Similarly we reported the completion of Tasks 4-6 in second progress report (January 2005 - May 2005). We have successfully implemented State Traversal algorithm to further reduce the size of test set by locating inert and recurrent subsequences and improved its performance (Task 4). We also completed the implementation of Subsequence Merging (MR), (Task 5). The team also completed the experimental work with the developed techniques to explore the possibility of developing an algorithm that achieves the best compaction level than any of the techniques separately.

Table 1.1: Task Achievement Diagram and Contributors. (AM = Aiman El-Maleh, SMS = Sadiq M. Sait, GS = Graduate Student)

	(Months)						Contributors
Tasks	0–3	4–6	7–9	10–12	13–15	16–18	
1	x						AM, SMS, GS
2	x	x					AM, SMS, GS
3		x					AM, SMS, GS
4		x	x	x			AM, SMS, GS
5		x	x	x			AM, GS
6			x	x			AM, SMS, GS
7				x	x		AM, SMS, GS
8					x	x	AM, SMS, GS
9					x	x	AM, SMS, GS
10					x	x	AM, SMS, GS
11	x	x	x	x	x	x	AM, GS
12		x		x		x	AM, SMS, GS

In this regard, we have proposed three algorithms, which clearly show the trade-off between compaction quality and timing, (Task 6).

We have now completed the remaining tasks, i.e., Tasks 7-10. We have successfully implemented FFC (Fault detection Frequency based Clustering) algorithm, which has improved the performance of IFC as reported in [1]. It reduces the size of test set and has shown improvement in the runtime as well, (Task 7). We have also completed the implementation of Class Based Clustering (CBC), (Task 8-10).

In addition, we have documented the software that has been implemented, (Task 11).

Task 12 is also progressing well. The details of all the papers is discussed in the section on publications.

1.2.2 Additional Tasks

In addition to the proposed tasks listed above, we have also implemented subsequence fault coverage increasing LROR for Sequential circuits. This algorithm has shown overall best published results for sequential circuits.

1.3 Budget Utilization

1.3.1 Man Power

Total Amount Allocated = SAR 60,100/-

Budget Utilized

- Principal Investigator (Dr. Aiman H. El-Maleh) @ SAR 1,200/- per month = SAR 7,200/-
- Co-Investigator (Dr. Sadiq M. Sait) @ SAR 1,000/- per month = SAR 6,000/-
- Remaining budget for Graduate Student/Research Assistants = SAR 800/-
- Secretary = SAR 700/-

Total utilization during this period = SAR 14,700/-
Total Amount Allocated on manpower= SAR 49,600/-
Total funds utilization as reported in first progress report= SAR 20,200/-
Total funds utilization as reported in second progress report = SAR 14,700/-
Total utilization as reported in this report = SAR 14,700/-

1.3.2 Books and References

Total Amount Allocated = SAR 2,500/- Total utilization during the project = SAR 0/-

1.3.3 Stationary and Miscellaneous

Total Amount Allocated = SAR 2,000/- Total utilization during the project = SAR 600/-

1.3.4 PC/Desktop

Total Amount Allocated = SAR 6,000/- Total utilization during the project = SAR 0/-

Total utilization of funds during the project = SAR 50,200/-

1.4 Difficulties Faced

The project is delayed due to unavailability of graduate student(s) to help in the implementation of the remaining tasks, i.e., Tasks 7-10. Furthermore, a bug in the implementation required thorough debugging and repetition of some of the experiments.

1.5 Publications

The following research papers have been published from the work in this project. In addition, three more publications are expected from the work on test compaction for combinational circuits.

1.5.1 Journal Publications

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Efficient Static Compaction Techniques for Sequential Circuits based on Reverse Order Restoration Based and Test Relaxation*, accepted to appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

1.5.2 Conference Publications

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Reverse Order Restoration Based on Test Relaxation and Subsequence Merging for Efficient Static Compaction for Sequential Circuits*, IEEE European Test Symposium, 22nd to 26th May '05 at Tallinn, Estonia.

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Efficient Static Compaction Techniques for Sequential Circuits based on Reverse Order Restoration and Test Relaxation*, IEEE Asian Test Symposium, 18th to 21st Dec'05 Salt Lake City, Kolkata, India.

1.5.3 Presentations

- Aiman H. El-Maleh. *Test set compaction for sequential circuits based on Test Relaxation and Subsequence Merging.* Presented at University of Southern California, May 05, 2005, California, USA.

1.6 Acknowledgment

The research team acknowledges King Fahd University of Petroleum & Minerals for all support.

Chapter 2

Static Test Compaction Algorithms for Combinational Circuits

In this chapter, an overview of static compaction algorithms for combinational circuits is given.

2.1 Overview

Static compaction algorithms for combinational circuits can be divided into three broad categories: (1) Redundant Vector Elimination, (2) Test Vector Modification, and (3) Test Vector Addition and Removal. Figure 2.1 shows a taxonomy of static compaction algorithms for combinational circuits [1]. In the first category, compaction is performed by dropping redundant test vectors. A redundant test vector is a vector whose faults are all detectable by other test vectors. Static compaction algorithms falling under this category can be further classified into two classes. The first class contains algorithms based on set covering in which faults are to be covered using the minimum possible number of test vectors. On the other hand, the second class contains algorithms based on test vector reordering in which fault simulation, fault distribution, and double detection are used to identify redundant test vectors and then drop them.

In the second category, compaction is performed by modifying test vectors. Algorithms belonging to this category can be further classified into three classes. The first class contains algorithms based on merging of compatible test cubes. A test cube is a test vector that is partially specified. A test vector is made partially specified by unspecifying the unnecessary primary inputs. This process is referred to as relaxation. Relaxation can be performed using an ATPG or a stand-alone algorithm

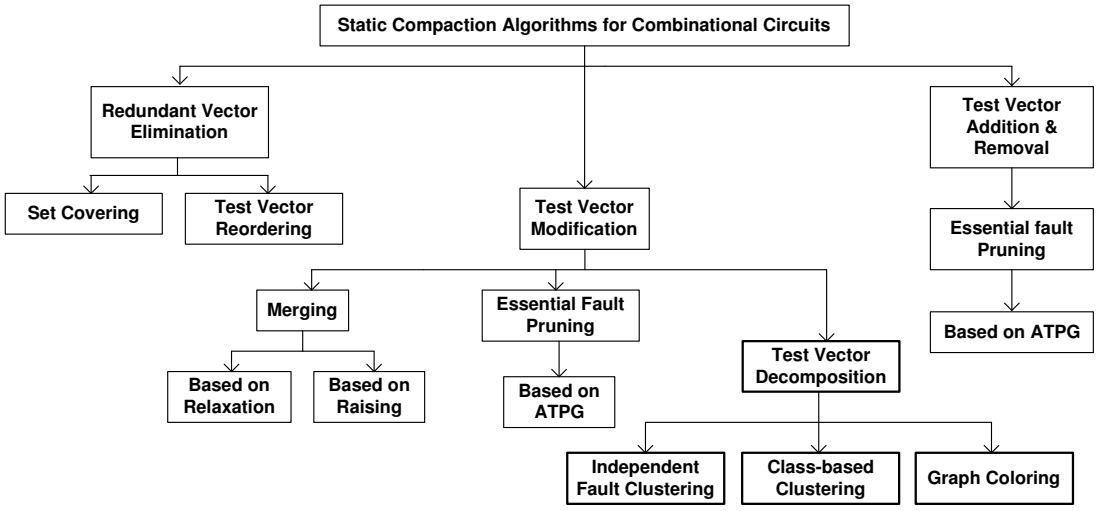


Figure 2.1: Taxonomy of static compaction algorithms for combinational circuits.

[17], [18]. In addition to relaxation, raising can be used to enhance the compatibility among relaxed test vectors. If two relaxed test vectors conflict at one or more bit positions, they can be made compatible by raising one of them at the conflicting bit positions.

The second class contains algorithms that employ essential fault pruning to make some test vectors redundant. A test vector is redundant if it detects no essential faults. A fault is essential if it is detected only by a single test vector. Essential faults of a test vector can be pruned, i.e., made detected by some other test vectors, by re-assigning values to those bits that are originally unspecified and have been randomly assigned values to detect additional faults.

The third class contains algorithms that are based on test vector decomposition. Test vector decomposition is the process of decomposing a test vector into its atomic components. An atomic component is a child test vector that is generated by relaxing its parent test vector for a single fault f. In [1], test vector decomposition is proposed as a new class of static compaction algorithms that modify test vectors to perform compaction.

Finally, the third category of static compaction algorithms consists of compaction algorithms that add new test vectors to a given test set in order to remove some of the already existing test vectors. The number of the newly added test vectors must be less than the number of test vectors to be removed. An ATPG is used to generate the new test vectors.

2.2 Set Covering

Test compaction for combinational circuits can be modeled as a set covering problem. The set cover is set up as follows. Each column of the detection matrix corresponds to a test vector and each row corresponds to a fault. If a test vector j detects fault i, then the entry (i, j) is one; otherwise, it is zero. In this setup, the total amount of memory required for building the detection matrix is $O(nf)$, where n is the number of test vectors and f is the number of faults.

Static compaction procedures based on set covering were described in [42], [43] and [44]. It should be pointed out that this approach has not been used much in the literature due to the huge memory and CPU time requirements.

2.3 Test Vector Reordering

Identification of redundant test vectors in a test set is an order dependent process. Given any order, redundant test vectors can be identified using fault simulation, fault distribution, or double detection. Hence, there are four variations of Test Vector Reordering (TVR) based static compaction algorithms.

2.3.1 TVR with Fault Dropping Simulation

Fault simulation of a test set in an order different from the order of generation is used as a fast and effective method to drop redundant test vectors. Under Reverse-Order Fault simulation (ROF) [45], [46], a test set is fault simulated with fault dropping in reverse order of generation. That is, a test vector that was generated later is fault simulated earlier. A test vector that does not detect any new faults, when it is simulated, is removed from the test set.

The intuitive reason for this phenomenon is simply that test vectors that are further down the list detect faults that are most difficult to detect. Therefore, if we first fault simulate a test vector that is at the end of the list, it not only detects a hard fault right away, it also detects many others by pure chance. This way hard faults are out of the way early.

2.3.2 TVR with Forward-Looking Fault Simulation

The forward-looking fault simulation is an improved version of ROF [46]. It is based on the idea that information about the first test vector that detects every fault can be used to drop test vectors that would not be dropped by ROF. That is, the yet-

Test	Faults
t_1	f_1,f_4
t_2	f_2,f_4
t_3	f_2,f_3

Figure 2.2: Test vectors and their associated faults.

Fault	Test
f_1	t_1
f_2	t_2
f_3	t_3
f_4	t_1

Figure 2.3: First test vector that detects every fault.

undetected faults have lower indexed test vectors that detect them. So, some test vectors are skipped over and consequently dropped from the test set.

Let us consider the following example. Let the test set T be $\{t_1, t_2, t_3\}$ and the fault set F be $\{f_1, f_2, f_3, f_4\}$. Figure 2.2 shows the test vectors with their associated faults and Figure 2.3 shows the first test vector that detects every fault. Conventional ROF first simulates t_3. This test will be retained in the test set to detect f_2 and f_3. Next, t_2 is simulated. Since it detects the new fault f_4, it is retained in the test set. Finally, t_1 is simulated and retained in the test set since it detects a new fault, f_1. No tests are dropped by ROF in this case.

Now, let us start ROF again taking into account the information given in Figure 2.3. ROF starts by simulating t_3. This test is retained in the test set to detect f_2 and f_3. Next, t_2 is simulated. t_2 detects the new fault f_4. However, f_4 is first detected by t_1. Therefore, we conclude that t_2 is not necessary for the detection of any yet-undetected fault and we drop it from the test set. Finally, when t_1 is simulated, the remaining undetected faults f_1 and f_4 become detected and the detection process completes. In this case, one test vector is dropped from the test set.

2.3.3 TVR with Fault Distribution

In TVR with fault distribution, a test vector is fault simulated without fault dropping. Faults detected by every test vector are recorded. Besides, the number of test vectors that detect every fault is recorded. Then, given any order, a test vector whose number of essential faults is zero, i.e., the faults it detects can be distributed among other test vectors, is considered redundant and thus can be dropped. After a test vector is dropped, the number of test vectors that detect every one of its faults

is reduced by one.

In [47], compaction based on fault distribution was used to compact test sets as a part of a dynamic compaction algorithm. The motive behind the proposed algorithm is the fact that ROF cannot identify a redundant test vector if some of the faults detected by it are only detected by the test vectors generated earlier. ROF can only identify a redundant test vector if all the faults detected by it are also detected by the test vectors generated later.

2.3.4 TVR with Double Detection Fault Simulation

Double Detection (DD) was first proposed in [48] as a dynamic compaction algorithm. Basically, when generating a new test vector, a yet undetected fault called a primary target fault is selected and a test vector t_i is generated to detect it. Next, other faults called secondary target faults are selected one at a time and the unspecified values in t_i are specified appropriately to detect the secondary target faults until no unspecified values remain in t_i or no additional secondary target faults can be detected. In choosing the secondary target faults, faults that are not detected are first considered and then faults detected at most once by earlier generated test vectors are considered. Faults are dropped from the list of target faults when they are detected twice. Test vectors that detect faults that are detected only once, i.e., essential test vectors, are marked. After the test generation is complete (when all the faults are detected at least once or aborted or proved to be untestable), the following static compaction procedure is used to reduce the test set size. The generated test vectors are simulated with fault dropping in the following order. First, all the essential test vectors are simulated in the order they were generated. The essential test vectors are followed by the non-essential test vectors in the order opposite to the order in which they were generated. During the fault simulation process, a test vector that does not detect any new fault is dropped. It should be pointed out that essential test vectors cannot be dropped and thus simulating them first maximizes the ability to drop other test vectors.

DD can be used in static compaction procedures [49]. However, since most test generators do not attempt to target faults for a second detection and do not use non-fault dropping simulation, they do not collect all the information necessary for static compaction based on DD. Therefore, the necessary information must be collected in a pre-processing step.

Table 2.1: Definition of the merging operation.

$\circ$	0	1	x
0	0	ϕ	0
1	ϕ	1	1
x	0	1	x

2.4 Merging

Static compaction algorithms in this class can be divided into two groups. The first group contains algorithms based on the very simple and efficient approach of merging compatible test cubes. A test cube is a relaxed test vector. A test vector is relaxed by unspecifying the unnecessary primary inputs. A test vector can be relaxed using an ATPG or a stand-alone algorithm [17] and [18]. A merging procedure employing relaxation proceeds as follows. Given a test set T, test vectors in T are all relaxed. Then, an iterative search is performed for pairs of compatible test vectors. Two test vectors t_i and t_j are compatible if they do not specify complementary values in any bit position. If any two vectors, say t_i and t_j, are compatible, they are replaced by the vector $t_i \circ t_j$, where $\circ$ represents the merging operation (see the definition in Table 2.1). The new test vector $t_i \circ t_j$ has all the binary values of both t_i and t_j. Hence, by a repetitive application of the above compaction operation, many test vectors (two or more) can be combined into fewer test vectors. As a result the total number of test vectors that need to be applied with the same fault detection capabilities is reduced. Examples of this approach can be found in [17], [50] and [51] .

In the second group, algorithms employ in addition to the relaxation operation a *raising* operation. For a test vector t, the raising operation $raise(t, i)$ tries to set the i^{th} bit of t to x while preserving the coverage of the essential faults of t. The raising operation was proposed in [52]. Raising is used to enhance compatibility among relaxed test vectors. For example, if two relaxed test vectors, say t_i and t_j, conflict at one or more bit positions, they can be made compatible by raising one of them at the conflicting bit positions. Typically, raising is used to resolve conflicts when a test set contains no compatible test vectors.

2.5 Test Vector Decomposition

Test Vector Decomposition (TVD) is the process of decomposing a test vector into its atomic components. An atomic component is a child test vector that is generated by relaxing its parent test vector for a single fault f. That is, the child test

vector contains the assignments necessary for the detection of f. Besides, the child test vector may detect other faults in addition to f. For example, consider the test vector t_p = 010110 that detects the set of faults $F_p = \{f_1, f_2, f_3\}$. Using the relaxation algorithm in [17], t_p can be decomposed into three atomic components, which are (f_1,01xxxx), (f_2,0x01xx), and (f_3,x1xx10). Every atomic component detects the fault associated with it and may accidentally detect other faults. An atomic component cannot be decomposed any further because it contains the assignments necessary for detecting its fault.

Static compaction based on merging (see Section 2.4) is a very simple and efficient technique. However, it has the following problems. First, for a highly incompatible test set, merging achieves little reduction. Second, raising is a costly operation. Third, a test vector must be processed as a whole. Therefore, it is proposed in [1] that a test vector be decomposed into its atomic components before it is processed. In this way, a test vector that is originally incompatible with all other test vectors in a given test set can be eliminated if its components can be merged with other test vectors.

By decomposing a test vector into its atomic components, a merging based compaction algorithm will have a more degree of freedom. This is because of the fact that the number of unspecified bits in an atomic component is much larger than that in a parent test vector. Thus, the probability of merging a component is higher than that of merging its parent test vector.

2.5.1 Graph Coloring

The problem of static compaction based on TVD can be modeled as a graph coloring problem. Basically, given a test set T with single stuck-at fault coverage FC_T, the set of atomic components C_T is first obtained. Then, a graph G is built. In G, every node corresponds to a component and an edge exists between two nodes if their corresponding components are incompatible. Our objective is to partition C_T into k subsets such that k is as small as possible and no adjacent nodes belong to the same subset. The fault coverage of the new test set T^* whose size is k should be greater than or equal to FC_T.

It is well known that graph coloring is an NP-hard problem [53]. Thus, efforts of researchers are devoted to heuristic methods, rather than exact ones. Heuristic methods are simple schemes in which nodes are colored sequentially according to some criteria.

2.5.2 Independent Fault Clustering

In [1], two compaction algorithms i.e., Independent Fault Clustering (which is based on Independent Fault Sets) and Class Based Clustering are explored.

Some important definitions are given next followed by the description of the two algorithms i.e., IFC followed by CBC.

Independent faults were defined in [54]. Basically, given a combinational circuit, let T_i be the set of all possible test vectors that detect f_i and T_j be the set of all possible test vectors that detect f_j. Then, two faults f_i and f_j are independent if and only if $T_i \cap T_j = \phi$. Independence among faults can also be defined with respect to a test set T. Let T_i' be the set of test vectors in T that detect f_i and T_j' be the set of test vectors in T that detect f_j. Then, two faults f_i and f_j are independent with respect to T if and only if $T_i' \cap T_j' = \phi$. In [1], the term independent faults is used to mean independent faults with respect to a test set.

Similarly a component of a test vector, is defined as the essential input assignment to detect a certain fault. For example, a test vector $T_i = 1001$, detects the fault f_j. T_i is then relaxed while still detecting f_j by changing the maximum number of input assignments to 'don't care', thereby making the component of f_j as 1xx1.

In Independent Fault Clustering (IFC), IFSs (Independent Fault Sets) are first derived. Then, a fault matching procedure is used to find sets of compatible faults, i.e., faults that can be detected by a single test vector. In the IFS derivation phase, independent faults are identified with respect to a test set. In the fault matching phase, compatible components, corresponding to compatible faults, are mapped to the same compatibility set. Whenever a component is mapped to a compatibility set, it is merged with the partial test vector of that compatibility set. At the end, every compatibility set represents a single test vector.

IFC's results lead to an important observation that the formation of IFS consumes a good fraction of total CPU time and gives an upper bound on the possible size of final test set after compaction. However, the compatibility set (formed by matching the essential and non-essential faults to a set) gets more realistic and closer to final test set size. Therefore one may exclude the IFS formation step to get better CPU time using the compatibility set based on essential and non-essential faults matching. This is in fact the motivation of one of the two algorithms proposed in this work. The algorithm is optimized to give better results both in terms of compaction and CPU time.

2.5.3 Class Based Clustering

The Class Based Clustering (CBC) algorithm [1], is based on the idea of dividing test vectors into classes and then heuristically processing test vectors of every class. A test vector is eliminated if its components can be all moved to other test vectors. Eventually, in the final test set, every test vector represents a cluster whose components originally belong to test vectors in different classes. This is why the technique is called Class Based Clustering.

The test vectors are first classified to different classes based on the number of components that can't be moved to other test vectors. For example, a test vector T_i detecting *three* faults and having *three* components that can all be moved to different test vectors in the test set, belongs to Class 0. Similarly a test vector T_j having total *three* components but only one of them can not be moved to any other test vector is called a class 1 test vector.

The algorithm first processes class 0 test vectors in a certain order, by moving all its components to different test vectors and thereby eliminating the vector from the test set. The processing of class 0 test vector follows the computation of blockage value of that test vector, which computes the number of test vector (belonging to class 0) that will be blocked by moving the vector to a particular test vector. The components of the test vectors are moved to those test vector that result in minimum blockage value.

The algorithm then processes class 1 test vectors. Class 1 test vectors are processed by moving the conflicting component to one of the candidate test vectors, whose conflicting component can be moved to some other test vector. Thereby removing the component that conflicts with the conflicting component of the test vector. This step is followed by eliminating class 1 test vector, whose all components are moved to other vectors. Similar procedure is executed for class 2 test vectors.

Experimental results show that the two algorithms give comparable level of compaction, however the computation time of CBC is higher than IFC. In CBC, an important observation is that, most of the compaction is achieved by processing class 0 test vectors and only marginal compaction is observed by processing test vectors belonging to higher classes. Another important observation in CBC is that the major portion of CPU time is used in calculating the blockage values and processing class 0 test vectors.

2.5.4 Essential Fault Pruning

Generally speaking, pruning a fault of a test vector decreases the number of its faults by one. A test vector becomes redundant if all of its faults are pruned. Fault Pruning (FP) is implemented as follows. Given a test vector t, an attempt is made to detect each of its faults by modifying the other test vectors in the test set. A fault of t is said to be pruned if it becomes detected by another test vector after the modification. If all the faults of t are pruned, then t can be removed from the test set.

The above operation of modifying a test vector, say $t^{'}$, to further detect an additional fault f of another test vector t is basically achieved by generating a new test vector $t^{''}$ such that DET($t^{''}$) = DET($t^{'}$) $\cup$ f, where DET(t) is the set of faults detected by t. Multiple Target Faults Test Generation (MTFTG) is used for this purpose. In MTFTG, a test vector is to be found for a set of target faults. MTFTG will fail if there exists at least two independent faults in the set of target faults. Two faults are independent if they cannot be detected by a single test vector.

The runtime of an FP-based static compaction procedure can be greatly improved by considering only essential faults. A fault is defined to be an essential fault of a test vector t if it is detected only by t. The set of essential faults of t is denoted by ESS(t). It should be pointed out that whenever a test vector t is eliminated, for every fault belonging to the set DET(t) - ESS(t), the number of test vector detecting it is reduced by one.

Few FP-based static compaction algorithms have been reported in the literature. Generally, they fall into two categories. In the first category, a test vector is modified such that it detects the new additional faults. The test vector already detects its essential faults. Therefore, the test generation time for the essential faults is eliminated. Examples of such static compaction algorithms can be found in [47], [52], [55] and [56]. On the other hand, in the second category, a set of N test vectors is replaced by a set of $M < N$ new test vectors. The basic idea is to determine the faults that are detected only by one or more test vectors among the N test vectors to be replaced and find $M < N$ test vectors that detect all theses faults. Examples of such static compaction algorithms can be found in [48] and [57].

Chapter 3

Static Test Compaction Algorithms for Sequential Circuits

Compaction of test sequences for Sequential Circuits is achieved by Dynamic and Static Compaction techniques. Dynamic Compaction techniques [22], [23] incorporate heuristics aimed at producing short test sequences into the test generation process. On the other hand Static Compaction procedure is applied as a post-processing step to test generation process.

Static Compaction offers the following unique opportunities in sequential circuits test generation.

- It may be applied to test vectors generated by any ATPG tool. Thus it does not modify the test generation procedure.
- It may be applied after dynamic compaction to further reduce the test size.
- It can be applied on test sequences generated by simulation based techniques.
- The shortest test sequences for sequential circuits are generated by static compaction techniques.

For the above reasons, Static Compaction is more popular in sequential circuits than Dynamic Compaction.

The following section reviews some of the known techniques for static test compaction for sequential circuits.

3.1 Subsequence Merging

In [2], three algorithms are given to merge self-initializing test sequences. The first algorithm merges aligned test sequences as shown in Figure 3.1 (a). If aligning two

sequences will result in a conflict between one or more vectors, a second algorithm is used to merge two sequences with a skew as shown in Figure 3.1 (b). The third algorithm improves the compatibility of test sequences using stretching. A sequence is stretched if some of its vectors are repeated several times without changing their order. For example the sequence (101x, 1x01, 111x) can be replaced by (101x, 1x01, 1x01, 111x). This will add one more degree of freedom to the process of compaction as shown in Figure 3.1 (c). Merging two test sequences using the last algorithm may affect the fault coverage. Therefore, a fault simulation step is performed after the merging process.

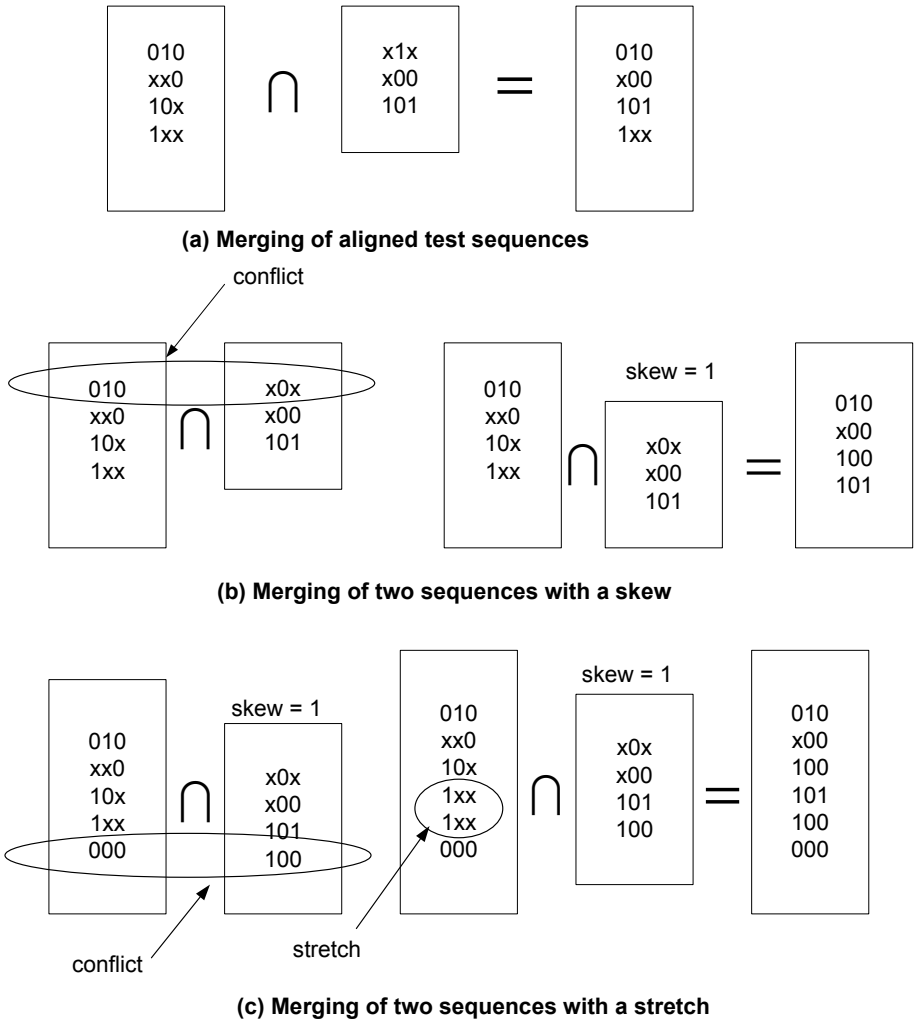


Figure 3.1: Merging of Subsequences illustrated [2]

3.2 Insertion, Omission and Selection

Pomeranz *et. al* proposed three different techniques for Static Test Compaction in [24] and [25] respectively. The techniques are *Insertion, Omission and Selection.*

Insertion reduces the test length by removing the states that repeat themselves while detecting a single fault. Thus removing such test vectors reduces the test length without reducing the fault coverage. Insertion operation can be better understood by the following example. Consider a fault $f \in F_{det}$ with detection time $u_{det}(f)$ (faults having higher detection time are preferred). Let u_j and u_k be two time units such that $u_j < u_k \leq u_{det}(f)$ and such that $S_j/S_j^f = S_k/S_k^f$ (i.e., $S_j = S_k$ and $S_j^f = S_k^f$). Since $S_j/S_j^f = S_k/S_k^f$, the time units from u_j to u_{k-1} only serve to take the fault-free/faulty circuit back to the states at time unit u_j and the test set T detects the fault f even if the subsequence is removed from T. The sequence obtained by omitting T[u_j, u_{k-1}] from T is T[u_0, u_{j-1}] $\circ$ T[u_k, u_{L-1}], where u_L is the time unit of the last test vector in the test set and $\circ$ denotes the concatenation of subsequences.

Using this approach a new test sequence is defined where the fault f is detected earlier, as follows: The subsequence T[u_k, $u_{det}(f)$] is duplicated and inserted at time unit u_j. As a result, the detection time of f is reduced from $u_{det}(f)$ to $u_{det}(f)$-(u_k-u_j). The remaining part of the sequence, T[u_j, u_{L-1}], is pushed to the right. Therefore the new subsequence is as follows:

$$T' = \text{T}[u_0, u_{j-1}] \circ \text{T}[u_k, u_{det}(f)] \circ \text{T}[u_j, u_{L-1}]$$

The above operation is known as insertion operation. The insertion operation increases the total length of the test sequence; however, it allows us to reduce its effective length by reducing the highest detection times. The shorter sequence T[u_0, $u_{L_{eff}-1}$] is then used instead of T.

The above step is the basic insertion operation as applied to the fault having the highest detection time. This operation is repeatedly applied to all the faults in decreasing order of detection time, using the test sequence formed by consecutive insertion operations. The algorithm applies the insertion operation (repeatedly) until further compaction or increase in fault coverage is possible. However, this is also bounded by an upper limit on the length of test set.

Experiments show that insertion reduces the test size and because of concatenation of subsequences, improves the fault coverage. But the drawback is the large number of fault simulations, which increase the execution time of the algorithm.

Omission considers the removal of a test vector t_i followed by fault simulation (considering all the faults) of the new sequence formed by the omission of vector t_i. If fault simulation shows reduction in fault coverage, the algorithm restores the test vector t_i and moves to the next vector t_{i+1}. Thus test vectors whose removal does not affect the fault coverage are removed from the sequence.

The algorithm considers test vectors in original order (as generated by ATPG) of

their appearance for removal. It achieves highest level of compaction in comparison to *Insertion and Selection*. *Omission* checks each and every vector for its possible removal which requires prohibitively large number of fault simulations (equal to the number of vectors in the test set), making the algorithm expensive in terms of execution time.

In [24], binary search technique is proposed to reduce the number of fault simulations, the algorithm quickly traces an inert subsequence (inert subsequence is one whose removal does not reduce the fault coverage). Binary search technique is applied to locate a removable test vector and checks the removal of inert subsequence (found by each step of binary search) thus reducing the number of fault simulations.

Selection begins by fault simulating the whole circuit without fault dropping, it then locates those subsequences that detect maximum number of faults. It then uses a covering procedure to determine the minimum number of test vectors that detect all the faults in the circuit. The best subsequence is one that detects maximum number of faults with minimum number of test vectors.

The algorithm finds the starting and ending point of a test sequence for each fault by simulating the circuit L-1 times without fault dropping, where L is the length of test sequence. Thus each time unit is used as the starting point of fault simulation without fault dropping. The algorithm then solves a covering problem to return the smallest possible test set size.

Selection uses fault simulation without fault dropping which is a very expensive step in terms of storage requirement. A large number of fault simulations makes it slow as well.

Experiments conducted on the three algorithms show that *Omission* gives the highest level of compaction followed by *Selection* and then *Insertion*. The execution time is not reported, but as discussed, all the algorithms rely heavily on fault simulations making the procedures slow and therefore infeasible for large industrial circuits.

3.3 State Traversal

The idea of *Insertion* is extended by Hsiao *et. al.* in [3] and [26]. The author has criticized the techniques in [24] and [25] as they require large number of fault simulations and therefore are impractical for large circuits. The proposed static compaction technique relies on the fact that a test set generated by ATPG goes through a small number of states, and some states are frequently revisited.

The number of states visited by a test set are small in comparison to the total number of test vectors for most circuits. The authors concluded that many subsequences that start and end on same states exist within these test sets. Test sets generated by various test generators exhibit similar phenomena. The subsequences

that start and end on the same state may be removed from a test set if certain conditions are met. The algorithm is not effective for circuits having few repeated states. The technique is fast as it only requires two fault simulation passes through the test set for compaction.

Some of the important definitions stated in the work include the following:

- A *State-Recurrence Subsequence* T_{rec} is a subsequence of vectors T[v_i, v_{i+1},...,v_j] such that the fault free states reached at the end of vectors v_{i-1} and v_j are identical.
- An *Inert Subsequence*, T_{inert} is a state-recurrence subsequence T_{rec}[v_i, v_{i+1},...,v_j] such that no additional faults are detected within the subsequence T_{rec}.
- Given a fault-free state S, the error vector E_f for a particular fault f is equal to S $\oplus$ S_f, where S_f is the corresponding faulty state for the same time frame.
- Given two identical fault-free states S, the error vector E_f for a fault f covers another error vector E'_f for the same fault and state if $E_f \bigcup E'_f = E_f$.

A number of criterion considered by the algorithm **Inert Subsequence Removal, ISR** to reduce the test set size are mentioned, which are as follows:

1. For an inert subsequence T[v_i, v_{i+1},...,v_j], if faulty state S_f^{i-1} at the end of time frame *i-1* and faulty state S_f^j at the end of time frame j are identical for every undetected fault f which is activated at time frames *i-1* and j, T_{inert} can be removed.
2. For an inert subsequence T_{inert}[v_i, v_{i+1},...,v_j], if error vector E_f^j at the end of time frame j covers E_f^{i-1} at the end of time frame *i - 1* for every activated fault f and the additional fault effects propagated at time frame j do not lead to detection, T_{inert} can be removed. The point is illustrated by figure 3.2.
3. For an inert subsequence T_{inert}[v_i, v_{i+1},...,v_j], if error vector E_f^{i-1} at the end of time frame *i-1* covers E_f^j at the end of time frame j for every activated fault *f*, T_{inert} can be removed if the additional fault effects propagated at time frame *i-1* do not cause fault masking in time frames starting at frame *j+1* as shown in figure 3.3.
4. For an inert subsequence T_{inert}[v_i, v_{i+1},...,v_j], if neither error vectors E_f^{i-1} and E_f^j cover the other, conditions imposed on activated faults in both criteria 2 and 3 (mentioned above) need to be satisfied in order for the inert subsequence T_{inert} to be removed.

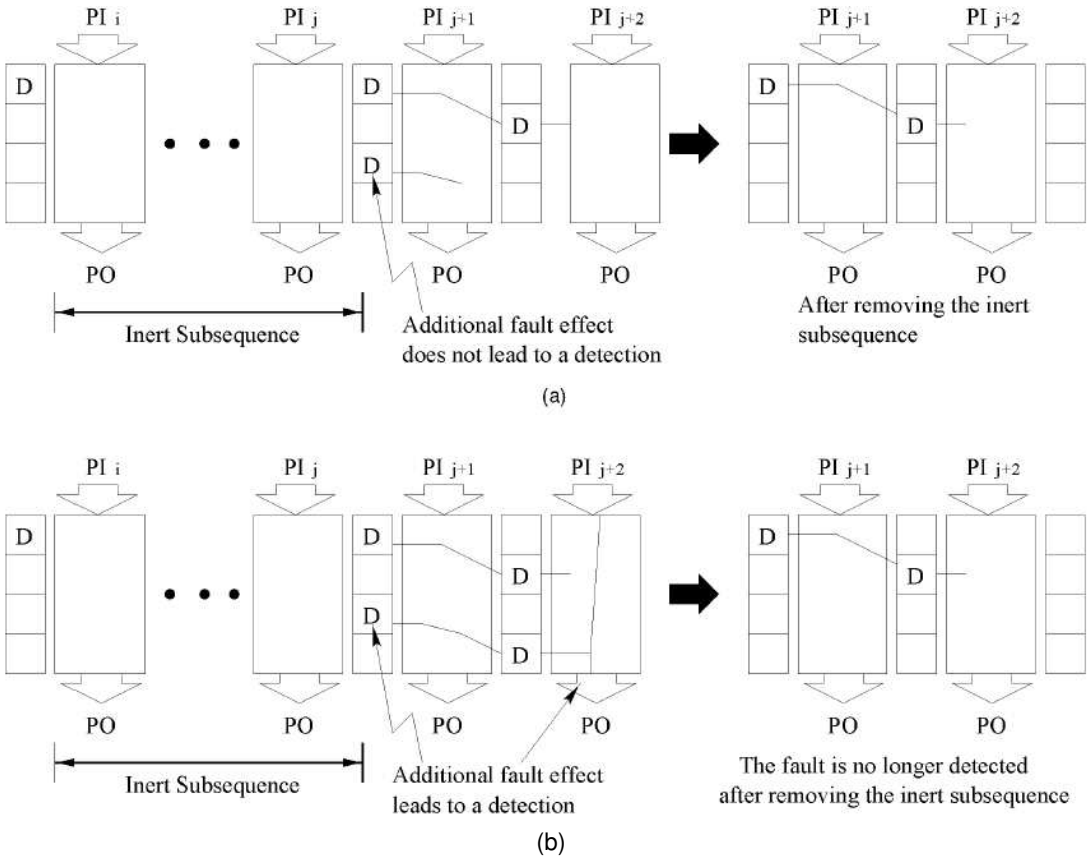


Figure 3.2: Criteria 2: Inert Subsequence Removal [3]

Authors have identified easy faults as ones which have multiple detection times, and require few constraints on value of input and memory elements. Since these faults are detected multiple times, subsequences exclusively detecting such faults (after being detected once) can be safely removed. The algorithm **Recurrence Subsequence Removal (RSR)**, is proposed to remove such subsequences, it first fault simulates without fault dropping (to know exactly the number of times each fault is detected) and then checks for the following four constraints to remove a state-recurrence subsequence $T_{rec}[v_i, v_{i+1},...,v_j]$:

1. All faults within T_{rec} have detection subsequences that do not overlap with T_{rec}.

2. For each fault active at the end of time frame j, if error vector E_f^j at the end of time frame j covers error vector E_f^{i-1} at the end of time frame *i-1*, and the additional fault effects propagated at time frame j do not lead to detection.

3. For each fault active at the end of time frame *i-1*, if error vector E_f^{i-1} at the

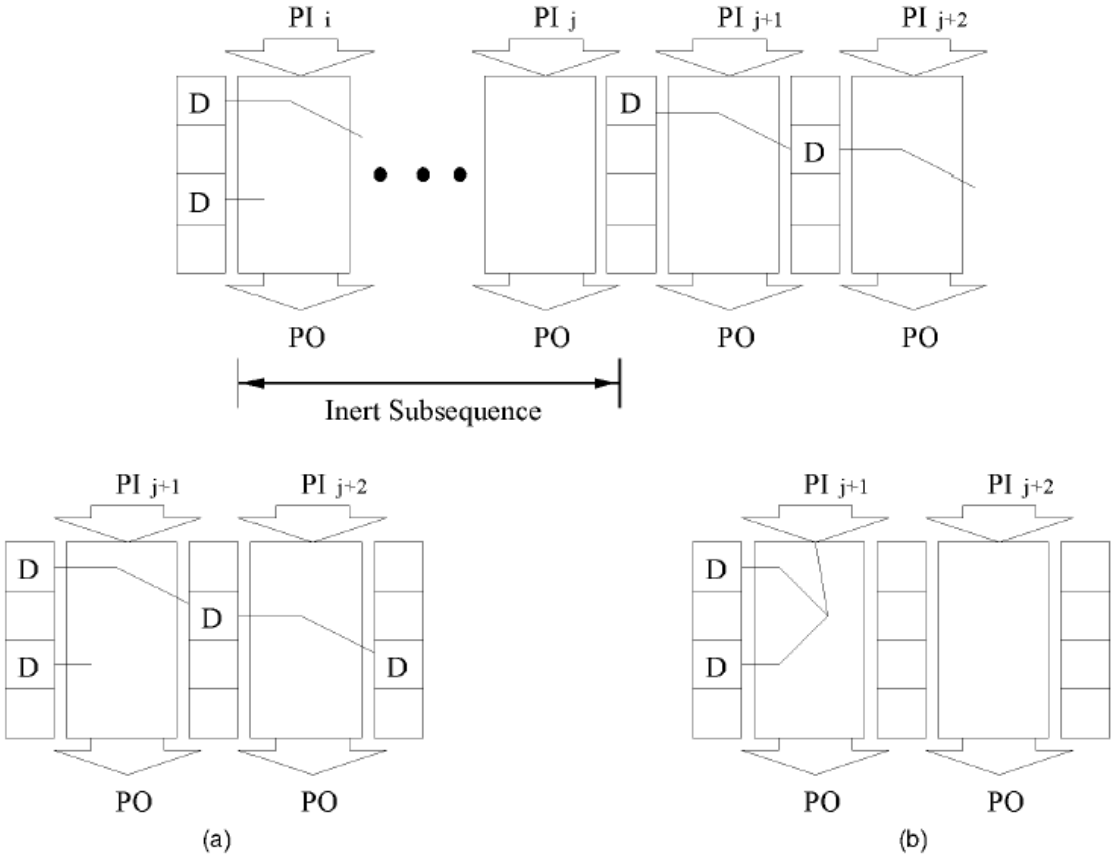


Figure 3.3: Criterion 3 illustrated [3]

end of time frame *i-1* covers the error vector E_f^j at the end of time frame j, the additional fault effects propagated at time frame *i-1* do not cause fault-masking in time frames starting at time frame *j+1*.

4. For each fault active at the end of time frame *i-1* and j, if neither error vector E_f^{i-1} nor error vector E_f^j covers the other, conditions imposed on activated faults in 2 and 3 (mentioned above) are satisfied.

The last three conditions are not necessary conditions, since faults which violate these conditions may be detected multiple times.

The results of algorithms ISR, RSR and their combination CSR show that the compaction achieved is marginal as compared to Omission and other algorithms known for producing compact sequences, but the run time is far less than those algorithms. Amongst the three algorithms, CSR mostly produced more compact test sequences consuming marginally higher time.

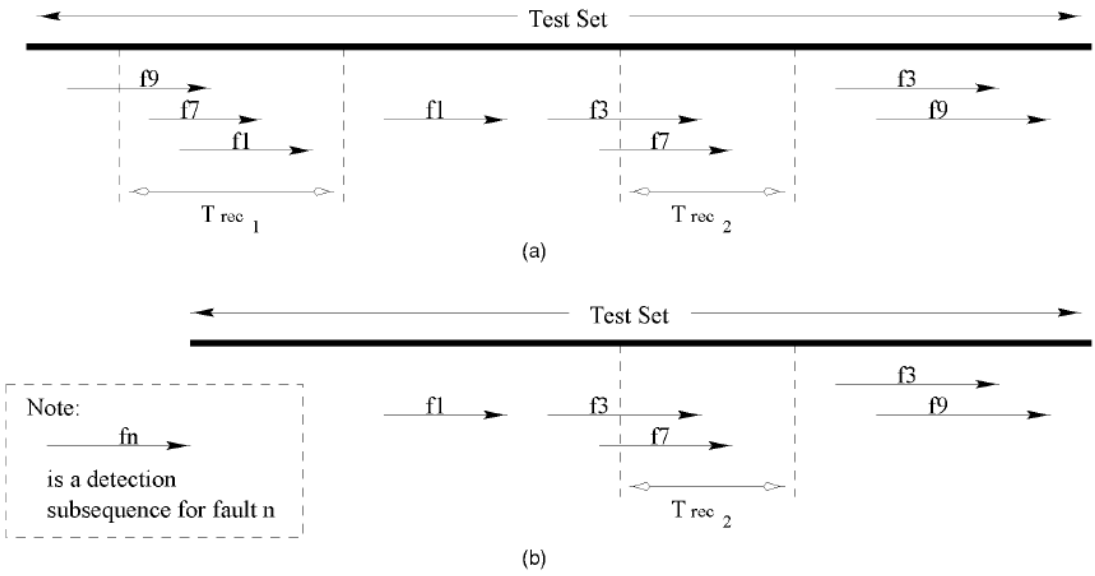


Figure 3.4: RSR Algorithm: Criterion 5 illustrated [3]

The above idea is extended in [26] by Hsiao *et. al.* This algorithm takes into account the fact that State Relaxation gives even more opportunity of finding inert and recurrent subsequences. State relaxation refers to relaxing the flip-flops of the circuits without reducing the fault coverage. The procedures in [3] are applied after getting the relaxed state of the circuit, which allows for even more compaction.

Experimental results show significant improvement in compaction achieved by the algorithm as compared to that in [3] for most of the circuits. The runtime of the algorithm is marginally higher than those in [3].

3.4 Vector Restoration

Vector Restoration based compaction procedure, proposed by Pomeranz *et. al.* [27, 16, 28], has given a new direction to static compaction of test vectors for sequential circuits. Several techniques have been developed based on the idea proposed in this work.

The main motivation behind this work came from the vector omission technique. For many test sequences it was observed that the test length after compaction was less than half of the original test length. This suggested that it might be faster to decide which test vectors must be *retained* in the test sequence in order to maintain the fault coverage, instead of deciding on the test vectors that might be *omitted*. This technique in principle first omits all the test vectors and then restores only those necessary to maintain the fault coverage.

Restoration of test vectors refers to the fact that all test vectors are first removed from the list and then are restored considering each fault one after the other, until the fault under consideration is detected.

The algorithm (procedure 1) proceeds as follows:

- The algorithm first fault simulates the circuit and marks the detection time of each fault together with the fault being detected.
- It then keeps the initial vectors that completely specify the state of the circuit and omits the rest, this is also called *Synchronizing Sequence*. Thus the initial vectors ensure that the circuit states are fully specified. This sequence is used to restore test vectors for every fault, since faults are not considered in a specific order, therefore the states of the flip-flops for each subsequence is specified by the synchronizing sequence.
- It then omits all the test vectors from the list of compacted test vectors.
- The algorithm then restores as many test vectors as required to detect the targeted fault. Different measures to select a fault for restoration are considered, for e.g. it could be selected: randomly, faults detected in decreasing order of detection time; procedure 1 uses the second criteria. The algorithm restores consecutive vectors near the test vector that detects the fault until the fault(s) under consideration is detected.
- Once test vectors for all the faults are obtained, fault simulation is performed to ensure that all faults are detected.
- If there are faults that were originally detected but are not detected by the compacted sequence, then additional test vectors are restored to detect them. It is possible that faults detected earlier by the compacted sequence may become undetected at later stage. This is because of subsequence concatenation necessary to create the final test set; the subsequence looses the synchronizing sequence that was initially used to restore the subsequence after it is concatenated to other test vectors in the final test set.
- Once all the faults are detected by the compacted test set, the algorithm physically omits all the test vectors which are no longer needed for detection of any fault in the circuit.

Few modifications to the above procedure are also suggested in the same work [27], by introducing algorithms 1E and 1R. Procedure 1E is the same as the above procedure, however it does not use the synchronizing sequence. Procedure 1R, on the other hand selects faults (for vector restoration) in random order, however it uses the synchronizing sequence introduced in procedure 1 (described above).

The fact that some earlier detected faults by the compacted sequence may become undetectable is the motivation of procedure 2 [27]. Procedure 2 considers a constant number of faults in one pass and restores test vectors for them. If any of the fault detected earlier becomes undetected, the procedure restores additional vectors right away before considering next group of faults. A group of five faults and synchronizing sequence is used with the procedure.

Experimental results show that *Procedure 1R* gives the highest level of compaction amongst procedure 1, 1E, 1R and 2, but has the highest execution time, however 1E has the smallest execution time. Procedure 2 gives more compaction than procedure 1 in many cases but the time is larger than 1 and 1E.

The paper [27] has also proposed two schemes to speed up the restoration process. In these procedures several faults having the same detection time are considered in parallel using parallel fault simulator HOPE [29].

Two schemes are proposed REST-RO64 and REST-SO64. The first considers 64 faults in random order while the second considers 64 faults in sorted order of decreasing time for restoration.

Experimental results show that the level of compaction is highest, even higher than omission-based compaction. The time to execute, however, is not reported. It is shown that the combination of REST-RO64 and REST-SO64 gives the best level of compaction.

Some of the advantages of restoration are:

- It is faster than vector omission technique, and some derivatives of restoration even achieve higher level of compaction than omission.
- An undetected fault is considered and vectors are restored to detect this fault only, which is cheaper than omission where a vector is removed and then fault simulation is carried out considering all the faults.
- The restoration of vectors considering faults in decreasing order of their detection time, depicts covering problem thus vectors for hard-to-detect faults (hard-to-detect faults tend to have higher detection time) take easy-to-detect faults into account. Therefore easy-to-detect faults have a good probability of being detected during restoration of test vectors for hard-to-detect faults.
- Concatenation of vectors may allow detection of faults which were not detected originally by the ATPG.

3.5 Hardware Reset Scheme

Hardware reset scheme is proposed by Higami *et. al.* [30] for sequential test set compaction. The authors have defined reset state by moving 0 to all flip-flops in the

circuit. Two schemes are proposed i.e. High-Cost and Low-Cost approach. The first approach does fault simulation without fault dropping while the second approach drop faults during fault simulation.

Both approaches categorize test vectors as removable and un-removable test vectors. In the High-Cost approach the number of removable vectors found are higher than the Low-Cost approach. A subset of removable vectors are then replaced by a reset state by logic simulation followed by fault simulation. The level of compaction achieved by the high-cost approach is consequently higher than the low-cost approach.

The two approaches classify the test vectors as First Fault Detecting vector (FFD: the first test vector that propagates the fault to some output of the circuit) and Fault Propagating vector (FP: the test vector that either excites the fault or propagates it to another FP or FFD vector). The authors then define either FP or FFD as un-removable while the rest as removable vectors.

Once the set of removable and un-removable vectors are obtained, the removable vectors are replaced by Reset State and logic simulation is done to check the state of un-removable vectors. The replacement is accepted only if the state of un-removable vectors is not changed by the replacement of removable vectors with the reset state. Thus logic simulation helps replacing removable vectors which do not affect the current state of un-removable vector. The removable test vectors which affect the current state of un-removable vectors are checked by fault simulation to ensure whether they are essential or not. Fault simulation also helps validating the fault coverage and thus attesting the removal of test vectors after logic simulation.

In the case of high-cost approach a detection matrix is made using fault detecting vectors and a covering problem is solved using a greedy algorithm to get a minimum number of test vectors detecting all the faults. Since fault simulation without fault dropping is an expensive procedure, the authors have used a threshold value to detect a fault a certain number of times, thus the fault is dropped after the threshold value.

The compaction results are compared with Reverse Order Restoration (ROR) technique. The compaction achieved by the low-cost approach is comparable to ROR for most of the circuits, but with an overhead of introducing reset states. On the other hand compaction achieved by the high-cost approach is significantly better than ROR with an expense of introducing reset states and fault simulation without fault dropping. The run time of the high-cost approach is more than double to that of the low-cost approach.

3.6 Vector Replacement

Pomeranz *et. al.* have proposed a number of schemes in [20], [31] to take a compaction procedure out of saturation. Any static compaction procedure applied to a

circuit is said to be saturated when its consecutive passes do not result in any further compaction. This procedure can be applied with any static compaction procedure to help overcoming saturation thus allowing higher level of compaction.

Two main schemes VERSE-C and VERSE-H are presented in this paper, each scheme comprises of a number of sub-schemes. VERSE-C (VERSE-Combined) replaces a vector in compacted test set T_C with another vector (from a set C) generated by the scheme proposed in [32]. The vectors generated by [32] give a wider domain of vectors together with states that may be used to replace certain test vectors in T_C.

The vectors in T_C have a specific state S_i on its memory elements and similarly the vectors in C have states C_{sj} for each test vector.

The algorithm first compares and determines the distance **D** between the states S_i to that of states C_{sj}. For example, suppose that the state and input vector of test vector from T_C is $S_i = (01x)$ and $t_i = (01)$, and $C_{sj} = (000)$ and $C_j = (00)$ are the states and input vector respectively from C, then the distance **D** will be 2, as the states differ from each other in two places. This is beacause un-specified state (x) is considered different from a specified state.

The value of **D** has an upper limit called D_{MAX} which is the maximum number of states the algorithm allows to differ in between vectors from the two sets.

VERSE-C has many different versions, which are explained as:

- The first procedure replaces test vectors in compacted set in order one after the other and tries to replace with vectors from C, using the value of D_{MAX} in increasing order, from *zero* onwards, ensuring that $t_i \neq C_j$. The two vectors are replaced.

- This is followed by fault simulation, therefore it fault simulates the circuit after replacing every test vector. A vector is restored whenever the fault coverage is reduced due to vector replacement.

- This step is followed by the application of compaction algorithm (which has to be taken out of saturation) and then higher values of **D** are also tried.

- The second version (Procedure 2) of VERSE-C is the same as the first, but vectors are replaced in reverse order of their appearance. The test vectors towards the end, usually detect hard-to-detect faults and thus replacing them allows detecting easy-to-detect faults and therefore more compaction.

- The third version (Procedure 3) replaces those test vectors at which the fault is detected i.e. propagated to one of its outputs. This reduces the number of replacements and thus fault simulations.

- The fourth version (Procedure 4) is identical to third but the faults are considered in reverse order of detection.

Pomeranz *et. al.* experimented with various procedures and various values of **D** indicate that procedure 2 is effective during the first few iterations since it achieved relatively low test lengths at relatively lower run times during these iterations. Procedure 1 or 3 are typically preferred for the later iterations. Based on these conclusions, procedures 1, 2 and 3 are combined. Procedure 4 did not have any advantage over the other procedures and is not used. The main algorithm of VERSE-C combines the three algorithms and applies them in the following order: procedure 2, 1, and then 3.

VERSE-H (VERSE-Holding) on the other hand does not make use of any test set produced by another algorithm for replacement of test vectors but it uses the same compacted set. VERSE-H also has two versions, which are as follows:

- The first version replaces a vector t_i by t_{i-1} for every $i \in$ the given test set. The algorithm then restores the test vector whose replacement reduces the fault coverage. For example the sequence (00,01,10,11) will be changed into (00,00,10,11) and then into (00,00,00,11).
- The second version of VERSE-H replaces the vectors starting from the end of the sequence. Therefore the sequence (00,01,10,11) will be changed into (00,01,11,11) and then into (00,11,11,11).

The main algorithm of VERSE-H combines the two approaches and starts with the second version and then switches to the first (in this way keeps switching between the two algorithms). This is done until the compaction algorithm can not further reduce the test sequence in allowable number of passes, after which the algorithm terminates.

Results show that VERSE-C has performed better than VERSE-H in most of the cases. However, there are few circuits for which the opposite is true.

The authors propose the use of VERSE-C and VERSE-H together, by applying one after the other.

These schemes use a large number of fault simulations and should be used only when the level of compaction is highly desirable and time to reach the solution may be sacrificed.

3.7 Sequence Re-Ordering

Sequence Re-ordering is proposed by Pomeranz *et. al.* in [21] as another static compaction technique. The scheme can be applied alone or as a preprocessing step applied before some other compaction algorithm. This can also be used to take an algorithm out of saturation and therefore a continuation of the work proposed in [20] and [31].

The procedure reorders the test vectors with the aim of achieving higher level of compaction while maintaining the fault coverage of the test set generated by the ATPG.

The procedure consists of two phases, the first phase, called *Sequence Reordering*, divides the vector set into equal sized partitions called subsequences. The optimal number of partitions, found by experiments is 7.

The partitions are then used to fault simulate the circuit starting from unknown initial states. The faults detected by each subsequence is recorded.

These subsequences are then concatenated, subsequences in consecutive order are concatenated together, to detect remaining faults which require larger number of test vectors for detection. This step requires fault simulation without dropping, to know which faults are detected by concatenating a unique pair of consecutive subsequence.

The above step reduces the number of subsequences in comparison to the original number, generated by the initial vector partitioning step.

The second phase of algorithm called *Subsequence Reordering* takes the subsequences from the previous step and then permutes them. The optimal permutation is one which detects all the faults using minimum number of test vectors.

For example, phase 1 of the algorithm gave 3 self-initialized subsequences. The second phase would permute, therefore giving 3! combinations of subsequences of test vectors. It would then find the optimal arrangement such that all the faults are detected using minimum number of test vectors. Thus compacting the test sequence.

The procedure applied alone gives little compaction. However, it gave better results when applied as a preprocessing step before *Sequence Counting* [33] and *Restoration* [27] based approach.

As mentioned above, the technique applied alone does not give good results and therefore should be applied as preprocessing step or to take an algorithm out of saturation. The run time of compaction when two or more algorithms are applied together increase tremendously and therefore is neither shown in the paper nor is the objective of the scheme proposed. Therefore, the algorithm is only preferred when the level of compaction is the only objective.

3.8 Chronological Order Enumeration

An improvement on the level of compaction achieved by vector omission technique is shown by Pomeranz *et. al.* in [33], [34]. In this paper, the authors have experimented with reordering of test vectors to achieve better level of compaction.

This approach, referred to as *Chronological Order Enumeration* and *Sequence Counting* in [34], omits test vectors from the test sequence and reintroduces them at a later time. Reintroduction of vectors helps reduce the compacted test sequence

length beyond the length that can be achieved if vectors are omitted permanently.

The algorithms follow the following sequence of steps:

- The basic step of the proposed procedure *Chronological Order Enumeration* consists of replacing a vector t_i at time unit i by a vector t_j, where $j > i$. The selection of t_j is random.
- If the above step reduces the fault coverage of the test, then the original sequence is restored, otherwise the change is accepted. In this way, each step attempts to replace a lower indexed test vector with a higher one.
- A variation proposed to this scheme is *Sequence Reduction* in [34]. The main motivation came from the fact that replacing a vector with only higher indexed vector gives very little opportunity for replacing test vectors existing towards the end of the set. For e.g., there is no way to replace the last test vector, since there is no higher indexed vector in the test set. Similarly, the first test vectors can't be replaced by lower indexed vectors. Finally, last few test vectors can be replaced by very few vectors.
- *Sequence Reduction* allows the vectors in compacted test set to be replaced by lower indexed vectors. This allows *Chronological Order Enumeration* more chances of compacting the test sequence. Therefore, replacing a vector with a lower indexed vector contributes to more shuffling and therefore provides more chances of compaction.
- The above procedures are called a number of times, thus if the test length is not reduced during a pre-selected constant, the algorithm terminates.

Results show improved level of compaction as compared to *Omission* and *Restoration* but the time to execute is higher than both of them.

3.9 Accelerated Restoration and Segment Pruning

A number of algorithms are proposed on the concept of *Restoration* [27]. Segment Reordering and accelerated restoration is discussed in [35]. The authors have divided restoration in two phases i.e. **Segment Validation** and **Segment Refinement**, which are as follows:

1. The algorithm begins with the initial fault simulation of the circuit and records the states and time units whenever a fault is detected.

2. In the validation phase, a target fault is selected and the algorithm tries to locate a near-accurate starting point of the subsequence detecting the fault. The algorithm initially starts locating the starting point of the fault either from last fault's starting vector (in case there exists a detected fault before the current target fault) or it simply starts from the last test vector in the test set (if the fault under consideration is the first target fault).

3. The algorithm then jumps back by subtracting 2^i, where i = 0,1,2,..., from the starting point until it detects the target fault.

4. It keeps track of the point where it last made the unsuccessful (kept as *min*) and successful (kept as *max*) attempt while fault simulating for the target fault.

5. The Validation is followed by Refinement phase, where the algorithm exactly locates the starting point of subsequence detecting the fault.

6. The variables *min* and *max* are used by refinement phase to fine tune its search. The algorithm keeps moving to the middle of the two values until the exact starting point is obtained. This is why the second phase is called Refinement (the algorithm refines the starting point of the subsequence detecting the fault).

7. This scheme, also known as 2-ϕ (two-phase) restoration, therefore gets the self-initializing subsequences of each fault, which are concatenated to get the compacted test set. Thereby accelerating the restoration process considerably.

The idea is extended by the same group of authors in [36]. This paper extends the idea of two-phase restoration to *Overlapped Restoration* which is followed by *Segment Pruning* procedures.

Overlapped Restoration comprises of overlapped validation and overlapped refinement. The idea of overlapped validation is simply to target faults together, that have overlapping subsequences. Therefore, two faults f_2 being detected by subsequence v_3 to v_{12} and f_3 being detected by subsequence v_2 to v_5, have overlapping subsequences and they may be targeted together in the validation phase of the algorithm. Thus the target fault list is kept flexible and additional faults are added to the list if they have overlapping subsequences.

The target faults and the segments found in the previous phase are passed on to the overlapped refinement phase (second phase of restoration). It again tries to optimize the segment size (size of subsequence detecting the target faults) considering the target faults. However, during the refinement phase the algorithm fine tunes the detection time of the segment, thus considering all the target faults, by similar method as described in [35].

The overlapped restoration is followed by segment pruning procedures. Two segment pruning procedures are defined by the authors, which are described as follows:

- Basic Segment Pruning algorithm which takes a subsequence/segment detecting a fault(s) and starts removing/clipping vectors from the beginning of the segment. The algorithm keeps removing vectors as long as all faults are detected.
- The second algorithm *Advanced Pruning* however, drops vectors from the segment rather than removing them only from the boundaries. For each iteration, simulation is done starting from a known initial state, to check the detection of fault if certain vectors are removed from the segment. This can be understood as application of *Omission* technique inside each segment; this gives more compact test segments.
- Advanced Pruning results in self-initializing segments that detect a number of target faults. Therefore, segments considered afterwards are independent of previous segments and thus need not to be considered during subsequent fault simulation passes. This reduces the overall number of fault simulations.

Experiments are conducted to compare the algorithms presented in [35] and [36] called SECO, to that of *Restoration* [27]. The following conclusion can be drawn based on the results presented by the author:

- In comparison of Overlapped Restoration with Restoration, Overlapped Restoration has produced similar results in terms of the level of compaction but the execution time is significantly smaller.
- Restoration was applied to large industrial circuits and could not produce results in 2 CPU days while SECO produced results in reasonable amount of CPU time.
- SECO is 5 to 30 times faster than Restoration on ISCAS circuits while for large industrial circuits it is 20 to 50 times faster than restoration.

3.10 Reverse Order Restoration

Reverse-Order-Restoration (ROR) is one of the most effective techniques known, proposed by Guo *et. al.* in [13] by extending the ideas in [15], [37] and [38]. The main difference between earlier work on *Restoration* proposed in [27] and the one discussed here include the following:

- Restoration is done in reverse order, i.e. in decreasing detection time of each fault and restored subsequences create a new test set having vectors in reverse order of their inclusion in the original test set.

- This helps save simulation efforts as fault simulation is done on the restored subsequences only.

- The algorithm includes a prefix that completely specifies the fault-free circuit.

The algorithm specifies k which selects integer time units that are to be restored i.e., all the faults in that time unit are targeted in single restoration phase. Different values of k are used, however when k=1, the algorithm is called *Linear Reverse Order Restoration*. The following paragraph together with the illustration shown in figure 3.5 explains the algorithm.

Let's suppose that the size of the test set T, to be compacted, is of length l. We denote the compacted test set as C, the Good and Faulty states as S_g and S_f, respectively.

- The algorithm begins with fault simulation to store the detection time of each fault. It then restores a number of test vectors for flip-flop initialization. In Fig. 3.5, this is shown by vector 1 in the beginning. The initial version of LROR [15] used 20 test vectors as synchronizing sequence in case of test size of more than 300 vectors and *l/16* otherwise.

- Fault simulation is then performed to check if there is any fault detected by the prefix; in Fig. 3.5, fault $f1$ is detected. The states of the circuit are stored, therefore subsequent vector restoration requires only to reach the S_g/S_f reached by the already restored subsequence. This speeds up the restoration process as test vectors can be appended towards the end.

- The algorithm targets all the faults in a single time frame to restore subsequences necessary for their detection. The targeted faults are selected in decreasing order of their detection time to produce a covering effect. In Fig. 3.5, *f7* is targeted as it appears in the last time frame.

- Vectors are restored for detecting the targeted faults; vectors closer to the time of detection are restored first, followed by fault simulation. If the fault is not detected, additional vectors are restored followed by a pass of fault simulation until the target fault(s) is detected.

  In Fig. 3.5, the vector at time unit 12 is restored first, which is followed by the restoration of vectors at time frame 11 and then 10 that finally detected *f7*. The example also shows the covering effect as faults *f3 & f5* are also detected.

- The algorithm proceeds (moves to the next time frame having undetected faults) until all the faults that were detected by the original test set are detected.

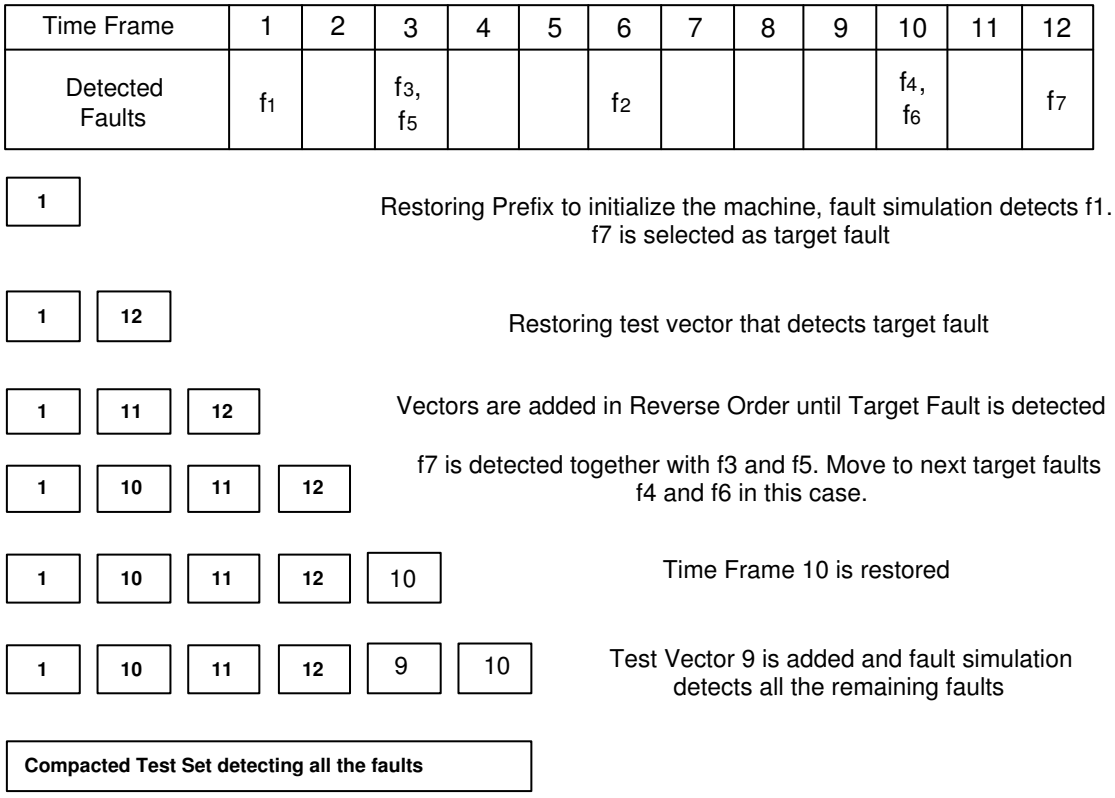


Figure 3.5: Reverse-Order-Restoration

The only difference between this basic ROR [15] scheme and the one proposed in [37] is illustrated in Fig. 3.6. During restoration, if a time frame is reached with still un-detected faults, then these faults are added to the target fault list and restoration includes test vectors for these faults also. Therefore, from Fig. 3.5 as soon as restoration for $f7$ reaches time frame 10, it includes *f4 & f6* in the target list and restoration will continue until time frame 9, subsequently detecting all the faults and giving a higher level of compaction.

Figure 3.6: Modified Reverse-Order-Restoration

The experiments show that the highest level of compaction is achieved with *k=1* as compared to other values of k and *Restoration* procedure in [27]. The algorithm is also compared with SECO [35] & [36] and showed higher level of compaction.

To speed-up the restoration process another algorithm Radix Reverse-Order-Restoration (RROR) is also proposed.

The algorithm has proposed Radix Search which is based on binary search technique i.e. r^{i-1}, where $1 \leqslant r \leqslant 2$. Radix-ROR selects a target fault, restores the vector that detects it, and then depending on the value of r it jumps to the vector located by the values of i, where i=1,2,3,... until the target fault is detected thus reducing the number of fault simulations and optimizing the execution time of the algorithm.

The algorithm is compared with LROR in the paper. Results show that compaction achieved is highest using *r=1* or LROR technique, however the execution time is reduced using higher radix values.

Mixed-Mode Algorithm is also discussed in the paper. The algorithm applies *Omission* after obtaining a subsequence using ROR and before appending it to the test set (compact set). Thus trying to achieve further compaction by removing un-necessary vectors from each subsequence.

Experiments to compare various algorithms show that the highest level of compaction is achieved by *MISC-ITE* i.e. Mixed Mode algorithm applied iteratively, followed by LROR-ITE which achieves comparable compaction at significantly less time, about $1/4^{th}$ of MISC-ITE.

3.11 SIFAR

SIngle FAult Restoration (SIFAR) is proposed by Lin *et. al.* in [14]. It uses the basic idea of restoration of test vectors (subsequences) to detect the fault, which are then concatenated to the compacted test set. The algorithm uses parallel simulation to speed up the process.

SIFAR considers a single target fault (in decreasing order of their detection time) and restores test vectors for each fault until the fault is detected. If there is more than one fault detected by the original test sequence at a single simulation time, then only one of them is considered as a target fault. Test vectors that detect the target fault are restored and concatenated to the compacted test set. SIFAR may not restore test vectors for all the faults in a single pass i.e., some of the faults may remain un-detected. Such faults are considered in the subsequent passes of SIFAR. The algorithm is iterated as many times as required to restore test vectors for all the faults, i.e., to restore the fault coverage. Most of the benchmarks required only two passes to restore the fault coverage as most of the faults are detected at more than one instant during fault simulation of the test set.

Experimental results show that SIFAR produces more compact test sets than

	f_1	f_2	f_3	f_4
S_1	1	0	3	0
S_2	2	5	0	0
S_3	0	3	1	4

Figure 3.7: Detection Matrix

SECO in almost all the cases with a considerable reduction in CPU time. SIFAR when compared with REST-SO64 gave better level of compaction and CPU time. However, it gave better level compaction as compared to ROR [15] for most of the circuits, but its CPU time exceeded to that of ROR in most of the cases. This is due to the fact that SIFAR is implemented using PROOFS [39], while ROR uses HOPE [29] for fault simulation. HOPE is twice as fast as PROOFS [14]. In addition, results of ROR and SIFAR are reported on different machines so direct comparison is not possible. Later, Linear Reverse Order Restoration and MISC [13] performed better than SIFAR, in terms of compaction quality.

3.12 Iterative Approach

Given a test sequence T that is composed of n self-synchronizing subsequences, iterative algorithms, such as genetic, tabu search, and simulated annealing algorithms, can be used to reorder the n subsequences such that some of them are shortened or eliminated. The optimization problem can be formulated as follows.
Given:

1. Set of faults F,
2. Test sequence T composed of a set of n self-synchronizing subsequences, and
3. Detection matrix for T.

Objective:

Order the n subsequences in T such that the test length, i.e. the sum of the effective test lengths of all subsequences, is minimum.

The detection matrix is an $n \times m$ integer matrix, where n is the number of subsequences and m is the number of faults. If S_i detects f_j, then the entry (i,j) corresponds to the first detection time of the fault. On the other hand, if S_i does not detect f_j, the entry (i,j) is zero. The effective test length of a subsequence S_i is equal to the maximum detection time for a fault in the set of faults detected by S_i.

Let us consider the following example. Let $T = \{S_1, S_2, S_3\}$ and $F = \{f_1, f_2, f_3, f_4\}$. S_1, S_2, and S_3 are composed of three, five, and four test vectors, respectively. Figure 3.7 shows the detection matrix. The test length of T is 12.

If the new order $\{S_3, S_2, S_1\}$ is considered, one can see that S_1 is not useful any more. This is because all the faults it detects are already detected by S_3 and S_2. Furthermore, the effective length of S_2 is reduced to 2 since f_2 is already detected by S_3. As a result, the effective length of the same test sequence in the new order is 6.

Genetic algorithms for static compaction of test sequences were proposed in [40, 41]. They are based on the optimization model described above. The input to the algorithms is the detection matrix, which can be easily computed using fault simulation with fault dropping enabled just during the simulation of each subsequence, during test generation process by ATPG. The self-initializing subsequences are also found as part of the test generation process by the ATPG. A solution is encoded as a string of integers. Every solution is assigned a fitness value that represents the number of test vectors eliminated with respect to the original test sequence. The adopted operators are the uniform cross-over, selection, mutation, and heuristic. The heuristic operator implements a simple local optimization procedure on the best solution. The algorithm achieved a test length reduction that varies from 50% to 62%.

Chapter 4

Proposed Static Test Compaction Techniques for Sequential Circuits

This chapter details the proposed test compaction algorithms for sequential circuits, tabulate results and justifies their behavior. It first begins with the discussion on the proposed techniques, which is followed by the limitations of Justification algorithm.

Some of the important attributes of a test set generated by any ATPG are summarized next. These are crucial in understanding the behavior of sequential circuit test generation and therefore contribute to efficient compaction.

- Hard-to-Detect faults are defined as those faults that are either difficult to excite, difficult to propagate, or both [58]. Therefore, a larger subsequence detects such faults. These faults are distributed usually towards the end of the test set generated by the ATPG.
- Easy-to-Detect faults, on the other hand require relatively fewer necessary assignments on the inputs as well as on the flip-flops, therefore can be detected by a relatively smaller subsequence [3]. Such faults are detected a number of times during the test generation process and are evenly distributed on the time frames generated by the test set.
- This distribution of faults points to an important fact; the subsequences for hard-to-detect faults may produce a covering effect, thus giving a high probability of detecting easy-to-detect faults. Therefore subsequences detecting easy-to-detect faults may be removed.
- A subsequence detecting a set of faults having relaxed state assignments can be further reduced by an inexpensive *State Traversal* step, as discussed in [26].
- A set of relaxed input vectors, provides another degree of freedom. Relaxed test vectors may merge in each other using techniques similar to [2], thereby contributing to reduction in test size.

- Removal of subsequences is possible by increasing the fault coverage of test vectors that have already been included in the Compacted Test Set (i.e., previously restored test vectors). Fault coverage can be increased by a number of techniques, one way is by relaxation of subsequences. Relaxation is done by considering all the faults detected by fully specified test set. The un-specified bits, found by relaxation algorithm are then randomly filled. This step is reiterated a fixed number of times and may result in increasing the fault coverage of test vectors that have (previously) been restored.

  This allows for detection of yet un-detected faults while keeping the size of the compacted test set. Therefore, test vectors required for detecting those faults (which are detected by increasing the fault coverage) can be completely eliminated, thereby fuelling compaction.

The proposed algorithms capitalize on an efficient test relaxation technique for sequential circuits [19]. The relaxation algorithm returns the relaxed assignment on inputs as well as on the flip-flops of the circuit, targeting all the (yet un-detected) faults, in a single time frame. The relaxation technique has the advantage of CPU time saving on simulation based techniques, when it comes to restoring the self-initializing subsequences. As observed in all the algorithms, the self-initializing subsequence requires a good number of fault simulations and therefore consumes a high percentage of the overall execution time of the algorithm. The proposed techniques capitalize on relaxed state assignments (produced by the algorithm in [19]) which returns the self-initializing subsequence by simply locating the time frame having all un-specified states.

In this work, a number of algorithms are proposed. The algorithms encompass all the attributes of Static Compaction algorithms, discussed earlier in this chapter and are based on Test Relaxation algorithm, which is their novelty. The proposed algorithms include the following:

1. Linear-Reverse-Order-Restoration based on Test Relaxation (RX-LROR).

2. Merging Restoration (MR) based on Test Relaxation.

3. Linear-Reverse-Order-Restoration using State Traversal (ST) algorithm. State Traversal has also been implemented in two ways, referred as ST and ST2. Their implementation with RX-LROR is referred as RX-LROR-ST and RX-LROR-ST2.

4. A hybrid scheme comprising of RX-LROR-ST and Relaxation of compacted test set is referred as Hybrid-I.

5. A hybrid scheme comprising of Hybrid-I followed by MR. This is referred as Hybrid-II.

6. Linear-Reverse-Order-Restoration (RX-LROR-ST) based on increasing the Fault Coverage of subsequences is referred as SFC-LROR.

7. A hybrid scheme comprising of SFC-LROR and MR is referred as Hybrid-III.

8. Merging Restoration based on increasing the Fault Coverage of subsequences is referred as SFC-MR.

9. The iterative versions of these algorithms are prefixed by ITE, followed by their respective reference.

4.1 Linear-Reverse-Order-Restoration based on Test Relaxation (RX-LROR)

The algorithm follows the same flow as the one proposed by Guo *et al.* [15]. The key difference in between the two algorithms lies in restoration of test vectors. As mentioned earlier, LROR [15] uses fault simulation targeting a group of faults, and keeps on restoring test vectors until all the faults are detected. However, the proposed algorithm uses Test Relaxation [19] to restore test vectors.

Similar to LROR [15] implementation, the proposed algorithm doesn't include new faults into the target fault set during subsequence restoration, while restoring test vectors for a group of faults in a single time frame.

Algorithm 4.1 illustrates our implementation of the Reverse-Order-Restoration technique based on test relaxation. Let's suppose that the size of the test set to be compacted T, is of length l. We denote the compacted test set as C; initially $C = \emptyset$. The Good and Faulty states are S_g and S_f, respectively. Given a time frame i, we denote the set of faults detected at i by F_i. We also denote the required flip-flop values for justifying the faults F_i by $(S_g/S_f)_i$. F_{target} holds all the faults detected by T.

Let S_i and S_j indicate the flip-flop values (required or reached) at time frame i and j, respectively. Then, the state justification requirements of S_j are covered by those of S_i, if $S_j \supseteq S_i$. For e.g., let S_j be 1X and S_i be 10. Then, $S_j \supseteq S_i$ and this means that the required values on S_j are satisfied by S_i. Finally, & is a concatenation operator.

Algorithm 4.1 starts by restoring a self-synchronizing sequence of length k vectors, where k is user-specified. Then, it starts the restoration process from the last time frame in the test sequence in which some faults are yet un-detected. Test restoration is shown in Algorithm 4.2. A test subsequence for a set of faults is restored by justifying the required values for detecting the faults frame-by-frame. The restoration process of the test subsequence terminates if the required values on the flip-flops at a time-frame are all X's or are covered by the flip-flop values reached by

the previously restored sequence. It should be noted that (S_g/S_f) holds the states for all undetected faults that reached to the flip-flops after fault simulation in step 3 of algorithm 4.1. On the other hand $(S_g/S_f)_i$, shown in Algorithm 4.2 indicates the required good and faulty values for time frame i. However, it should be observed that the good and faulty values in $(S_g/S_f)_i$ and (S_g/S_f) are compared only with respect to the faults being justified i.e. F_i.

Once a test subsequence is restored, an attempt to reduce its size is made by State Traversal algorithm, which is discussed in the next subsection. Finally, the reduced subsequence is concatenated to the previously restored sequence and only the concatenated sequence is fault simulated, and detected faults are dropped. The process continues until all the faults are detected.

Algorithm 4.1 Reverse Order Restoration (RX-LROR)

1. Fault Simulate the circuit using the given test set.
   Collect the detection time of each fault.
2. Restore the first k test vectors as a synchronizing
   sequence from the given test set T. C=$\{v_1,v_2,v_3,.....v_k\}$.
3. Fault simulate the restored sequence C and drop all
   the faults detected from F_{target}. Store the (S_g/S_f)
   values of all the flip-flops for all undetected faults.
4. **if** $(F_{target} = \emptyset)$ Return C **else** Go to Step 5.
5. V = Test Restoration(n, F_n), where n is the last
   time frame having undetected faults.
6. $C = C$ & V; Go To Step 3.

Algorithm 4.2 Test Restoration (n,F_n)

1. Let $i = n$, and $V = \emptyset$ be the sequence currently
   restored.
2. $(S_g/S_f)_i$ = *Justify(F_i,i)* and let $j = i$.
3. **while**$(((S_g/S_f)_j \neq X)$ and $((S_g/S_f)_j \not\supseteq (S_g/S_f)))$ {
   $V = V_j$ & V //add current time frame to V
   $j = j - 1$ //move back single time frame
   $(S_g/S_f)_j$ = Justify(F_i, j) //get the required
   //values for all flip-flops in this time frame
   } //end while
4. Return(V)

LROR [15] used 20 test vectors as synchronizing sequence in case of test size more than 300 vectors and *l/16* otherwise. In our implementation of RX-LROR,

the synchronizing sequence is kept the same for a fair comparison. During cost function computation for flip-flops, it applies a multiplicative weight of 10.

The results of proposed RX-LROR are shown in table 4.1 and is compared with LROR [15], on STRATEGATE [9] test sequences. It can be seen that the proposed algorithm has performed better in terms of CPU time than LROR [15] both in one-shot and iterative versions. The one-shot version of the proposed algorithm has shown better results than LROR [15] on 10 out of 16 circuits. It resulted in longer test sequences for few circuits (specifically s444, s526, s1423 and s5378), due to which the overall savings are lesser than LROR [15]. This deviation in performance is due to current limitations of used justification algorithm, the limitations and possible solutions are discussed in section 3.6. The next two columns compare the iterative versions of two algorithms. Iterative version of proposed RX-LROR keeps on re-iterating until results are not improved in four consecutive iterations. However, the iterative version of LROR [15] stops as soon as a single iteration is unable to reduce the test size. From table 4.1, it can be seen that LROR [15] has performed better on 11 out of 16 circuits with higher overall savings. It can be noticed that the proposed RX-LROR has suffered from quick saturation and is unable to reduce the test size for many circuits.

Table 4.1: Compaction Results of proposed RX-LROR on STRATEGATE Test Sequences.

		LROR [15]	RX-LROR	ITE LROR [15]	ITE RX-LROR
Ckt	TS	TS (sec)	TS (sec)	TS (sec)	TS (sec)
s298	194	**138 (0.14)**	152 (0.06)	**112 (0.74)**	152 (0.24)
s344	86	62 (0.09)	**44 (0.03)**	51 (0.18)	**44 (0.12)**
s382	1486	606 (1.49)	**593 (0.26)**	**545 (4.26)**	593 (0.86)
s444	1945	**642 (2.15)**	839 (0.38)	**607 (5.48)**	839 (1.26)
s526	2642	**1269 (8.86)**	1855 (0.67)	**1269 (67.81)**	1854 (3.34)
s641	166	**118 (0.13)**	133 (0.07)	**117 (0.32)**	133 (0.33)
s713	176	139 (0.16)	**115 (0.07)**	**103 (0.61)**	114 (0.43)
s820	590	489 (0.79)	**469 (0.27)**	471 (1.94)	**457 (2.44)**
s832	701	543 (0.89)	**534 (0.31)**	**443 (4.5)**	462 (2.35)
s1196	574	277 (0.28)	**268 (0.3)**	**260 (1.2)**	266 (1.26)
s1238	625	285 (0.31)	**268 (0.33)**	270 (1.09)	**266 (1.68)**
s1423	3943	**1031 (12.87)**	1287 (3.14)	**836 (50.43)**	1171 (12.26)
s1488	593	501 (1.79)	**466 (0.56)**	474 (14.89)	**466 (2.54)**
s1494	540	468 (1.71)	**453 (0.52)**	**422 (21.92)**	445 (3.36)
s5378	11481	**677 (38.71)**	760 (45.34)	**585 (71.55)**	690 (85.67)
s35932	257	137 (56.93)	**131 (20.8)**	137 (119.76)	**125 (128.7)**
Total (sec)	25999	**7382** (127.27)	8367 **(73.11)**	**6702** (366.68)	8077 **(246.84)**

4.2 RX-LROR with State Traversal

State traversal is implemented in two ways, we refer to it as ST and ST2. This section discusses the two algorithms and shows experimental results.

4.2.1 State Traversal

The state traversal algorithm is called after a sequence is restored and is shown in Algorithm 4.3. In Algorithm 4.3, it is assumed that the restored subsequence V, consisting of n vectors, detects F_n faults. It is also assumed that i and j are variables corresponding to time frames i and j, respectively.

Initially, during restoration, the algorithm stores for each fault the S_g/S_f requirements that have to be justified in previous time frames. Next, for each time frame j, the algorithm checks for the earliest possible time frame i such that the justification requirements of time frame j are satisfied by the justification requirements of time frame i. If such a time frame i is found, then the vectors from i to $j-1$ are redundant and can be removed.

Algorithm 4.3 is illustrated in Fig. 4.1. As shown in Fig. 4.1, the algorithm stores S_g/S_f for each fault in a list. Since, $(S_g/S_f)_4 \supseteq (S_g/S_f)_2$ for fault f_1, thus the state requirements at time frame 4 are satisfied by the state requirements at time frame 2. Therefore, test vectors 2 and 3 can be removed from the restored subsequence without affecting the fault coverage. It should be observed that algorithm 4.3 takes into account all the faults in F_n when comparing (S_g/S_f) values. Therefore, the algorithm removes redundant vectors, just by state comparison without doing any additional fault simulation.

Algorithm 4.3 State Traversal(V,F_n)

1. Let i=2 and j=n.
2. **while**($j > 2$) {
   if((for each fault k $\in F_n$ $((S_g/S_f)_i \subseteq (S_g/S_f)_j)$))){
   Clip Vectors V_i to V_{j-1} from V
   j=i-1 ; i=2 }
   else if($i < j-1$) $i++$
   else { $j--$; $i=2$ }
   } // End while
3. Return (V).

The modified RX-LROR with State Traversal is shown by algorithm 4.4, which refines the restored subsequence before appending it to the compacted test set.

The experimental results of proposed RX-LROR-ST are shown in table 4.2 and are compared with LROR [15] on STRATEGATE [9] test sequences. It can be seen

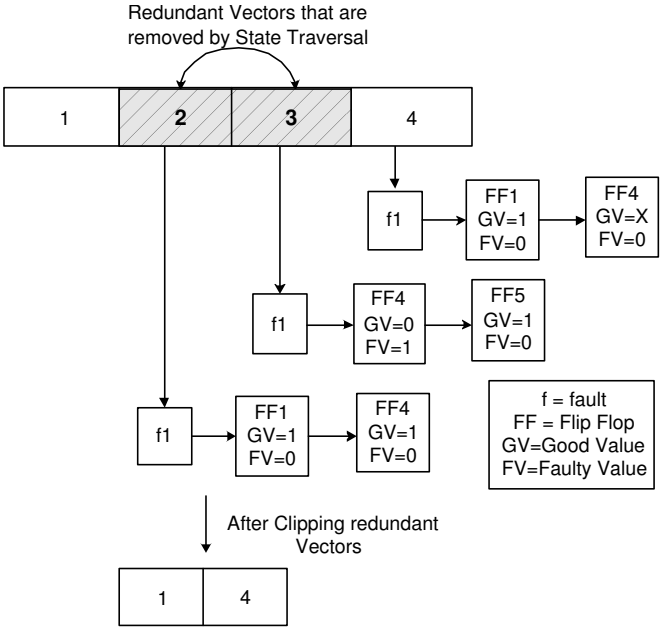


Figure 4.1: Compaction by State Traversal Algorithm.

that RX-LROR-ST has performed better in terms of CPU time than LROR [15], both in one-shot and iterative versions. It should also be noticed that State Traversal has shown significant reductions and results have improved in comparison to RX-LROR, shown in table 4.1, at the expense of a very small penalty of CPU time.

The one-shot version of RX-LROR-ST has shown better results than LROR [15] on 10 out of 16 circuits. It resulted in longer test sequences for few circuits (again due to current limitations of justification algorithm), due to which the overall savings are lesser than LROR [15]. The next two columns compare the iterative versions of the two algorithms. Similar to ITE-RX-LROR, Iterative version of RX-LROR-ST keeps on re-iterating until results are not improved in four consecutive iterations. From table 4.2, it can be seen that ITE-LROR [15] has performed better on 11 out of 16 circuits with higher overall savings. It can be noticed again that ITE-RX-LROR-ST has also suffered from quick saturation and is unable to reduce the test size for many circuits.

4.2.2 State Traversal-2

It can be observed from the results of RX-LROR-ST that for some of the circuits, the algorithm increased the size of compacted test set, as compared to the test size achieved by RX-LROR, the fact is highlighted by table 4.3. The behavior is shown by circuits s382, s641, s713, s1423 and s1488. This points to the fact that some of the faults have their subsequence completely or partially inside test vectors,

Algorithm 4.4 Reverse Order Restoration with State Traversal

1. Fault Simulate the circuit using the given test set.
   Collect the detection time of each fault.
2. Restore the first k test vectors as a synchronizing
   sequence from the given test set T. C=$\{v_1,v_2,v_3,.....v_k\}$.
3. Fault simulate the restored sequence C and drop all
   the faults detected from F_{target}. Store the (S_g/S_f)
   values of all the flip-flops for all undetected faults.
4. **if** $(F_{target} = \emptyset)$ Return C **else** Go to Step 5.
5. V = Test Restoration(n, F_n), where n is the last
   time frame having undetected faults.
6. V = State Traversal(V, F_n)
7. $C = C$ & V; Go To Step 3.

clipped by ST. Therefore, fault-simulation phase is not able to detect those faults, hence test vectors have to be restored again. This results in redundancy of test vectors and therefore, overall size of the test increased in comparison to RX-LROR implementation.

This observation motivated the development of State Traversal-2 (ST-2), which is shown in algorithm 4.5. It follows the same steps as ST with a difference that it removes vectors if no fault is detected within those vectors. This heuristic was found experimentally useful in reducing the overall restored test sequence by state traversal and not resulting in longer test sequences.

Algorithm 4.5 State Traversal-2(V,F_n, F_{target})

1. Let i=2 and j=n.
2. **while**$(j > 2)$ {
   if((for each fault k $\in F_n$ $((S_g/S_f)_i \subseteq (S_g/S_f)_j)$) **&**
   (No fault $\in F_{target}$ detected in Time Frames i to $j-1$)){
   Clip Vectors V_i to V_{j-1} from V
   j=i-1 ; i=2 }
   else if$(i < j-1)$ $i++$
   else { $j--$; $i = 2$ }
   } // End while
3. Return (V).

The performance of different versions of RX-LROR is compared in Table 4.3. For the circuits mentioned earlier, which resulted in higher test size by RX-LROR-ST than RX-LROR, it can be noticed that RX-LROR-ST2 has reduced the test size of s641, s713 and s1488. However, it is unable to reduce the test size of s382 and

Table 4.2: Compaction Results of RX-LROR-ST on STRATEGATE Test Sequences

		LROR [15]		RX-LROR-ST		ITE LROR [15]		ITE RX-LROR-ST	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	138	0.14	**134**	**0.05**	**112**	0.74	134	**0.24**
s344	86	62	0.09	**44**	**0.05**	51	**0.18**	**44**	0.21
s382	1486	**606**	1.49	625	**0.35**	**545**	4.26	625	**1.25**
s444	1945	**642**	2.15	668	**0.39**	**607**	5.48	668	**1.26**
s526	2642	**1269**	8.86	1385	**0.81**	**1269**	67.81	1385	**2.76**
s641	166	**118**	0.13	157	**0.12**	**117**	**0.32**	134	0.7
s713	176	139	0.16	**134**	**0.09**	**103**	**0.61**	107	0.86
s820	590	489	0.79	**466**	**0.41**	471	1.94	**447**	**1.84**
s832	701	543	0.89	**470**	**0.44**	**443**	4.5	469	**2.17**
s1196	574	277	**0.28**	**268**	0.38	**260**	**1.2**	266	1.43
s1238	625	285	**0.31**	**268**	0.41	270	**1.09**	**266**	1.88
s1423	3943	**1031**	12.87	1327	**2.75**	**836**	50.43	1271	**11.43**
s1488	593	501	1.79	**479**	**0.72**	**474**	14.89	**474**	**5.36**
s1494	540	468	1.71	**401**	**0.77**	422	21.92	**401**	**3.22**
s5378	11481	**677**	**38.71**	710	45.02	**585**	**71.55**	659	81.84
s35932	257	137	56.93	**131**	**22.15**	137	**119.76**	**125**	134.85
Total	25999	**7382**	127.3	7667	**74.91**	**6702**	366.68	7475	**251.3**

s1423. It should be noticed that the overall savings of RX-LROR-ST are highest, which is due to s526, for which RX-LROR-ST2 increased the size significantly rather than reducing it. In terms of CPU time, it should be noticed that different implementations of State Traversal algorithm are inexpensive and contribute to test size reduction.

The overall performance of RX-LROR-ST2 in comparison to LROR [15] is shown in Table 4.4. It can be noticed that one-shot RX-LROR-ST2 has performed better than LROR [15] on 10 out of 16 circuits. However, LROR [15] performed significantly better on s526 and s1423 (limitations of justification algorithm), due to which it resulted in higher overall savings than RX-LROR-ST2. However, the iterative version of LROR [15] has shown better performance on 12 out of 16 circuits with higher overall savings. It can be noticed that ITE-RX-LROR-ST2 has also suffered from quick saturation, and for many circuits it is unable to reduce the test size.

4.3 Merging Restoration

Merging Restoration (MR) follows the same flow as Algorithm 4.1 and is shown as algorithm 4.6. It takes advantage of the unspecified assignments at the inputs of the extracted subsequence and merges it with previously restored subsequences rather

Table 4.3: Different versions of State Traversal on STRATEGATE Test Sequences

		RX-LROR		RX-LROR-ST		RX-LROR-ST2	
Ckt	TS	TS	sec	TS	sec	TS	sec
s298	194	152	0.06	134	0.05	152	0.06
s344	86	44	0.03	44	0.05	44	0.04
s382	1486	593	0.26	*625*	*0.35*	*625*	*0.26*
s444	1945	839	0.38	668	0.39	614	0.36
s526	2642	1855	0.67	1385	0.81	*1890*	*0.67*
s641	166	133	0.07	*157*	*0.12*	119	0.08
s713	176	115	0.07	*134*	*0.09*	112	0.08
s820	590	469	0.27	466	0.41	456	0.29
s832	701	534	0.31	470	0.44	498	0.36
s1196	574	268	0.3	268	0.38	268	0.31
s1238	625	268	0.33	268	0.41	268	0.36
s1423	3943	1287	3.14	*1327*	*2.75*	*1328*	*2.72*
s1488	593	466	0.56	*479*	*0.72*	453	0.56
s1494	540	453	0.52	401	0.77	434	0.57
s5378	11481	760	45.34	710	45.02	726	45.87
s35932	257	131	20.8	131	22.15	131	21.85
Total	25999	8367	73.11	7667	74.91	8118	74.44

than concatenating it.

The idea of merging is similar to the one proposed by Roy *et al.* [2]. The subsequences can be merged in different ways i.e., from Start and End. Merging from Start is shown by Algorithm 4.7. It checks the compatibility of the two test sequences (currently restored V and compacted test set C), and tries to merge the two test sequences starting from the last test vectors of V and C towards the beginning of the test sequences. Merging from End, on the other hand is exactly the opposite, it checks the compatibility of the two test sequences C and V, and tries to merge the two sequences starting from the first test vectors towards the end of the test sequences. The flow chart of the algorithm is shown in Fig. 4.2. Similarly, another scheme uses a more greedy heuristic and decides on merging the subsequence wherever savings are higher. However, experimental results showed that Merging from Start gave overall best results. Therefore, our work uses Merging from Start only. State traversal (ST) is not applied in MR as higher compaction is achieved without it.

Algorithm 4.6 Merging Restoration

1. Fault Simulate the circuit using the given test set.
   Collect the detection time of each fault.
2. **if** ($C \neq \emptyset$) Fault simulate the restored sequence C and drop all
   the faults detected from F_{target}.
3. **if** ($F_{target} = \emptyset$) Return C **else** Go to Step 4.
4. Let $i = n$, where n is the last time frame having
   undetected faults. Let $V = \emptyset$ be the sequence
   currently restored.
5. $(S_g/S_f)_i$ = *Justify(F_i,i)* and let $j = i$.
6. **while** ($(S_g/S_f)_j \neq X$) {
   $V = V_j$ & V //add current time frame to C
   $j = j - 1$ //move back single time frame
   $(S_g/S_f)_j$ = Justify(F_i, j) //Get the states for all
   //flip-flops reached in this new time frame
   } //end while
7. $C = MergeStart\,(C, V)$; Go To Step 2.

Algorithm 4.7 *MergeStart* (C,V)

1. Let n_c and n_v be the number of test vectors in C
   and V.
2. **if** ($n_c < n_v$) swap C with V and n_c with n_v.
3. Let i=last test vector in C, SM=i
4. Let j=last test vector in V
5. **if** (i $\geq$ 1)
   while(j $\geq$ 1 **and** i $\geq$ 1) {
   if($C[i]$ and $V[j]$ are compatible) { j = j-1; i = i-1;
   if(i = 0 **OR** j = 0) Merge C and V, starting from
   $C[SM]$ and $V[n_v]$
   } // end if
   else { SM = SM-1; i = SM
   goto step 4 } //breaking the while loop
   } // end-while
6. **else** $C = V$ & C;
7. Return(C)

Table 4.4: Performance of RX-LROR-ST2 on STRATEGATE Test Sequences

		LROR [15]		RX-LROR-ST2		ITE LROR [15]		ITE RX-LROR-ST2	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	**138**	0.14	152	**0.06**	**112**	0.74	152	**0.25**
s344	86	62	0.09	**44**	**0.04**	51	**0.18**	**44**	**0.18**
s382	1486	**606**	1.49	625	**0.26**	**545**	4.26	622	**1.13**
s444	1945	642	2.15	**614**	**0.36**	**607**	5.48	614	**1.12**
s526	2642	**1269**	8.86	1890	**0.67**	**1269**	67.81	1714	**3.36**
s641	166	**118**	0.13	119	**0.08**	**117**	**0.32**	118	0.57
s713	176	139	0.16	**112**	**0.08**	**103**	0.61	111	**0.5**
s820	590	489	0.79	**456**	**0.29**	471	**1.94**	**428**	1.98
s832	701	543	0.89	**498**	**0.36**	**443**	4.5	460	**2.44**
s1196	574	277	**0.28**	**268**	0.31	**260**	**1.2**	266	1.34
s1238	625	285	**0.31**	**268**	0.36	270	**1.09**	**266**	1.8
s1423	3943	**1031**	12.87	1328	**2.72**	**836**	50.43	1207	**11.93**
s1488	593	501	1.79	**453**	**0.56**	**474**	14.89	423	**4.36**
s1494	540	468	1.71	**434**	**0.57**	**422**	21.92	434	**2.53**
s5378	11481	**677**	**38.71**	726	45.87	**585**	**71.55**	651	92.97
s35932	257	137	56.93	**131**	**21.85**	137	**119.76**	**125**	132.9
Total	25999	**7382**	127.3	8118	**74.44**	**6702**	366.68	7635	**259.36**

Merging from Bottom

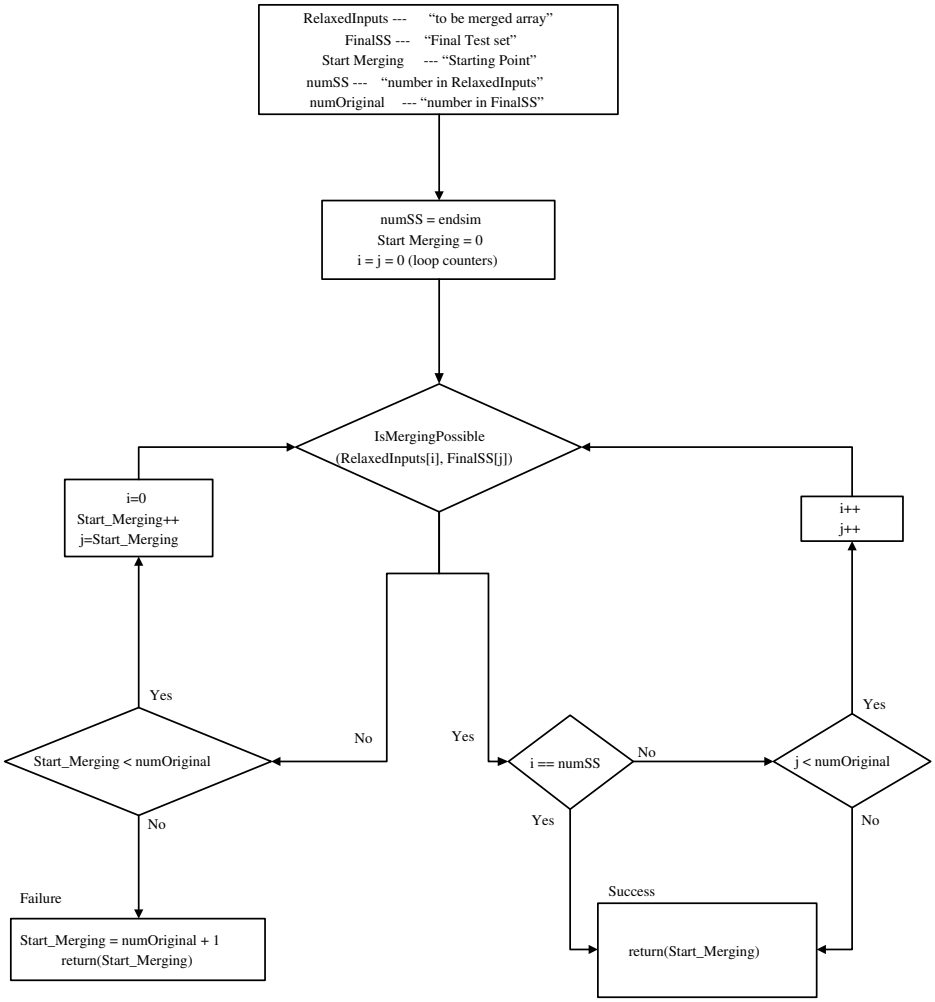


Figure 4.2: Merging from End

Table 4.5 shows the performance of MR and compares it with LROR [15] and the proposed RX-LROR algorithms. It can be seen that MR has performed better on only 3 out of 16 circuits compared to LROR [15] and our implementations of RX-LROR, with significantly higher CPU time. Our experimental analysis has shown that MR has the potential of squeezing the size of test set and significant savings are possible if it is applied after RX-LROR, as it gives another level of freedom to the test compaction procedures. This is discussed in the subsection on Hybrid schemes.

After a single run of MR Algorithm, there is a large percentage of un-specified bits. These bits can be randomly filled for subsequent iterations.

Table 4.5: MR on STRATEGATE test sequences

		LROR [15]		RX-LROR		RX-LROR-ST2		MR	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	**138**	0.14	152	**0.06**	152	**0.06**	154	0.08
s344	86	62	0.09	**44**	**0.03**	44	0.04	61	0.06
s382	1486	606	1.49	593	**0.26**	625	**0.26**	**567**	0.28
s444	1945	642	2.15	839	0.38	**614**	**0.36**	1029	0.39
s526	2642	**1269**	8.86	1855	**0.67**	1890	**0.67**	1669	1.07
s641	166	**118**	0.13	133	**0.07**	119	0.08	148	0.67
s713	176	139	0.16	115	**0.07**	**112**	0.08	140	0.59
s820	590	489	0.79	469	**0.27**	**456**	0.29	531	3.38
s832	701	543	0.89	534	**0.31**	**498**	0.36	568	3.39
s1196	574	277	**0.28**	268	0.3	268	0.31	**242**	1.86
s1238	625	285	**0.31**	268	0.33	268	0.36	**248**	2.25
s1423	3943	**1031**	12.87	1287	3.14	1328	**2.72**	1650	27.61
s1488	593	501	1.79	466	**0.56**	**453**	**0.56**	533	5.46
s1494	540	468	1.71	453	**0.52**	**434**	0.57	501	4.88
s5378	11481	**677**	**38.71**	760	45.34	726	45.87	1549	231.73
s35932	257	137	56.93	**131**	**20.8**	131	21.85	188	380.7
Total	25999	**7382**	127.3	8367	**73.11**	8118	74.44	9778	664.4

The large CPU time of MR can be understood by comparing the number of subsequences restored by MR and RX-LROR-ST2, which is shown in Table 4.6. It can be seen that MR restores higher number of subsequences than RX-LROR-ST2. It is because of the fact that the restored subsequences are un-specified and result in detection of lesser number of faults than a subsequence of same size but having specified bits. This coupled with the fact that the merged subsequences need to be fault simulated in contrast to fault simulating only the newly restored subsequence, which results in comparatively higher number of fault simulations (in case of MR than RX-LROR algorithms). This in turn increases the time taken by MR.

4.4 Hybrid Schemes

In this section we propose two hybrid schemes, which reduce the inherent limitation of Vector Restoration algorithms of quick saturation and are found useful when these algorithms are re-iterated.

Hybrid-I is composed of two primary steps. In the first step (step-I), the proposed RX-LROR-ST2 algorithm (Algorithm 4.4) is run for two iterations and if there is any reduction in test sequence length in any of these two iterations, the algorithm runs for one more iteration/improvement. The algorithm re-iterates by running an

Table 4.6: Number of Subsequences restored by MR and RX-LROR-ST2 on STRATEGATE test sequences

Ckts	MR No. of SS	RX-LROR-ST2 No. of SS
s298	8	6
s344	18	6
s641	65	9
s713	72	15
s820	87	29
s832	88	25
s1196	192	147
s1238	207	150
s1488	65	16
s1494	62	16
s5378	132	49
s35932	35	7
Total	1031	475

extra iteration as long as the last iteration reduces the test sequence length. This step is followed by Test Relaxation [19] and randomly filling the un-specified bits (using 1 as seed), which forms the second step of Hybrid-I (step-II) of Hybrid-I. Test Relaxation and random filling (step-II) change the composition of test set, while maintaining its fault coverage. This helps moving the algorithm out of local minima and the search space is therefore increased. Furthermore, it allows RX-LROR-ST2 to re-iterate far longer and partially replaces almost every test vector at a very low cost of CPU time. Step-II is again followed by step-I and the process continues (step-I followed by step-II) until four consecutive iterations are unable to reduce the test size.

Hybrid-II is based on the intuition that merging of relaxed subsequences (MR) gives another level of freedom to test compaction, therefore it may further squeeze the size of test set, if applied after Hybrid-I. As mentioned previously, MR requires comparatively large number of fault-simulations than RX-LROR. This drawback makes it vulnerable to large sized test set in terms of CPU time.

Hybrid-II is proposed to keep the advantages offered by MR, while restricting its limitations. It applies MR to the solution found by Hybrid-I. In this algorithm, MR is applied once and is re-iterated until each pass of MR does not further reduce the size of test set. Similar to Hybrid-I, it uses 1 as a seed, to randomly fill the un-specified bits of the test set during first iteration, and in subsequent iterations it adds 50 to the previous value of seed.

The Hybrid schemes are compared with ITE-LROR [13], ITE-MISC [13] and

Table 4.7: Hybrid Schemes on STRATEGATE Test Sequences

	STRATEGATE Test Sequences										
		ITE LROR [13]		ITE SIFAR [14]		ITE MISC [13]		ITE Hyb-I		ITE Hyb-II	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	125	0.6	112	**0.4**	98	3.2	106	1.56	**89**	1.83
s344	86	47	**0.1**	48	0.2	**43**	0.4	48	0.45	48	0.51
s382	1486	524	**3.3**	**498**	5.6	507	17.8	603	3.34	595	3.71
s444	1945	461	7.9	**454**	7.2	455	31.3	621	**4.71**	614	5.24
s526	2642	945	26.6	975	**8.9**	**884**	209.8	1605	14.27	1483	15.95
s641	166	78	0.5	87	**0.4**	**63**	1.7	68	2.14	68	2.32
s713	176	72	**0.6**	94	1.1	**60**	0.8	64	2.02	64	2.23
s820	590	394	**6.4**	388	6.5	**335**	15.2	377	22.17	376	26.24
s832	701	458	8.8	435	**4.5**	**368**	14	418	19.32	401	24.95
s1196	574	221	**1.7**	237	3.4	216	3.2	213	38.49	**184**	46.63
s1238	625	222	2.6	251	**1.5**	222	3.6	222	33.88	**190**	40.7
s1423	3943	843	**108.3**	778	113	**702**	469.2	1096	114.11	1096	128.98
s1488	593	343	27.1	**312**	**8.8**	364	39.4	362	17.72	361	28.56
s1494	540	297	33.8	313	**6.8**	**296**	72.6	408	11.65	403	22.49
s5378	11481	711	339.4	597	**89.5**	**583**	2148	586	401.38	586	473.16
s35932	257	110	752.3	152	**290**	**101**	1177	133	875.76	133	1002.7
Total	25999	5851	1320	5731	**547.8**	**5297**	4207.2	6930	1562.97	6691	1826.2

ITE-SIFAR [14] on STRATEGATE and HITEC test sequence in Table 4.7 and Table 4.8, respectively.

Considering STRATEGATE test sequences shown in Table 4.7, it can be noticed that ITE-Hybrid-I has performed better on 7 out of 16 circuits than ITE-LROR [13], while 1 resulted in a draw, the CPU time of the two algorithms is comparable. When compared to ITE-SIFAR [14], ITE-Hybrid-I has again performed better on 9 out of 16 circuits, while 1 resulted in a draw. In comparison to ITE-MISC, it has performed better on 2 out of 16 circuits, while one circuit resulted in a draw. ITE-MISC has achieved higher overall savings, but its CPU time is significantly higher.

Similarly, it can be noticed from Table 4.7 that ITE-Hybrid-II has performed better on 8 out of 16 circuits than ITE-LROR [13]. In comparison to ITE-SIFAR [14], ITE-Hybrid-II has again performed better on 9 out of 16 circuits, while 1 resulted in a draw. However, ITE-MISC has performed better on 12 out of 16 circuits but the CPU time is again significantly higher than that of ITE-Hybrid-II. ITE-Hybrid-I and ITE-Hybrid-II resulted in longer test sequences for few circuits (specifically s444, s526, s1423 and s1494), due to which the overall savings are lesser than the three algorithms. This deviation in performance is due to current limitations of used justification algorithm, the limitations and possible solutions are discussed in

Table 4.8: Hybrid Schemes on HITEC Test Sequences

	HITEC Test Sequences								
		ITE LROR [13]		ITE MISC [13]		ITE Hyb-I		ITE Hyb-II	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec
s298	322	109	**0.8**	**97**	1.1	161	1.27	143	1.43
s344	127	47	**0.1**	47	0.5	**45**	0.9	**45**	0.93
s382	2074	288	**2.1**	**279**	77.5	871	4	864	4.47
s444	2240	343	**2.3**	**302**	22.4	821	7.26	634	7.96
s526	2258	404	**9.3**	**352**	31.8	1326	20.09	1326	20.67
s641	209	**63**	**1**	72	1.2	66	3.07	66	3.24
s713	173	74	**0.7**	74	1	**71**	2.27	**71**	2.47
s820	1115	578	**13.8**	**432**	28.3	489	24.85	488	28.19
s832	1137	562	**8.3**	**383**	64	497	18.29	495	22.46
s1196	435	226	**2.3**	223	2.5	214	35.98	**180**	42.01
s1238	475	227	**1.9**	225	**1.9**	218	44.15	**184**	49.6
s1423	150	111	**1.1**	**110**	1.6	138	2.88	131	5.2
s1488	1170	**571**	**10.4**	572	354.6	650	41.2	648	50.52
s1494	1245	540	**9.1**	**492**	274	616	57.19	611	68.66
s5378	912	**245**	**108.1**	271	189	262	89.46	262	107.83
s35932	496	142	**227.8**	**117**	1158	187	1039.5	159	1879.1
s3271	709	555	**24.6**	443	265	682	51.77	**366**	95.11
s3384	161	104	**11.6**	92	13.1	104	16.9	**75**	19.4
s4863	518	302	**20.5**	315	25.6	272	376.45	**140**	419.1
Total (sec)	15926	5491	**455.8**	**4898**	2513.1	7690	1837.48	6888	2828.35

section 3.6.

Next, these algorithms (other than ITE-SIFAR) are compared on HITEC test sequences. As Table 4.8 shows, ITE-Hybrid-I has performed better than ITE-LROR [13] on 7 out of 19 circuits while 1 circuit resulted in draw. In comparison to ITE-MISC, it has performed better on 7 circuits. However, these results are improved by ITE Hybrid-II. It can be seen that ITE-Hybrid-II has shown better results than ITE-LROR [13] on 9 out of 19 circuits and higher overall savings than ITE-Hybrid-I. While comparing to ITE-MISC [13], it has shown better performance on 9 out of 19 circuits. The effect of ITE-Hybrid-II is even more pronounced for the circuits s3271, s4863 and s3384. The overall savings are lesser because of few circuits (s382, s444, s526 and s1494), these circuits resulted in longer test size (see section 3.6).

4.5 Subsequence Fault Coverage Increasing based Compaction

In this section, we propose a modification to the RX-LROR compaction algorithm (Algorithm 4.4) to maximize its effectiveness in producing more compacted test sequences.

In this algorithm, instead of only concatenating the restored test vectors for a group of faults, as is done by RX-LROR class algorithms, the currently restored subsequence is first relaxed and then the un-specified bits are randomly filled to increase the fault coverage of restored test sequence. This leads to the complete removal of those subsequences that are otherwise needed to detect the un-detected faults (by increasing the fault coverage). Intuitively this approach (SFC-LROR) seems to achieve compaction quality of RX-LROR based schemes, as a worst-case.

Fig. 4.3 illustrates the behavior of SFC-LROR in comparison to RX-LROR. RX-LROR restores the test sequence (6, 7) to detect the faults f3 and f10, and the test sequence (11, 12) to detect faults f5 and f6. On the other hand, SFC-LROR detects these faults in earlier test sequences. SFC-LROR increases the fault coverage of the test sequence (1, 2, 3) to detect f3. Similarly, the test sequence (4, 5) detects the faults f5 and f10, and the test sequence (8, 9, 10) detects the fault f6, in addition to previously detected faults. Hence, SFC-LROR restores lesser test sequences giving higher level of compaction.

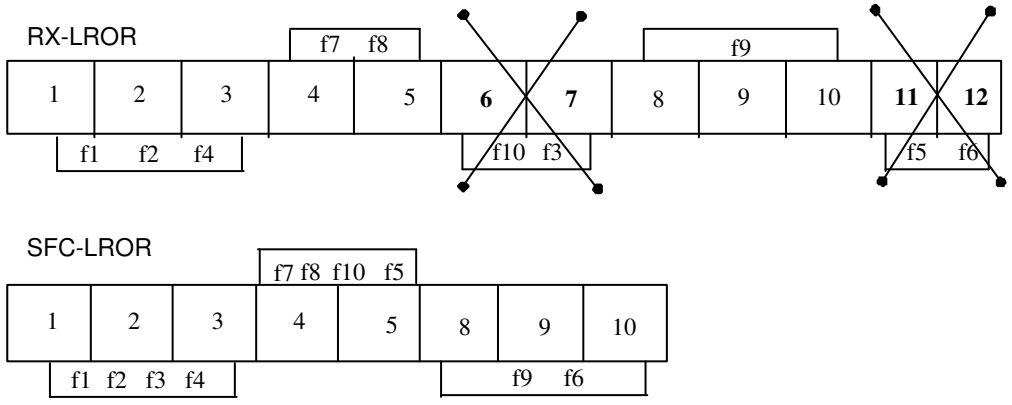


Figure 4.3: Compaction by Subsequence Fault Coverage Increasing LROR (SFC-LROR)

4.5.1 Subsequence Fault Coverage Increasing LROR

The proposed algorithm is called subsequence fault coverage increasing LROR (SFC-LROR) and is shown as Algorithm 4.8. It follows the same steps as RX-LROR-ST2,

Algorithm 4.4, with a difference that after concatenating the newly restored test sequence to the compacted test set, relaxation algorithm [19] is called to return the un-specified input assignments on the currently compacted test set. This step is followed by randomly filling the un-specified inputs. Randomly filling the un-specified inputs is essentially used for increasing the fault coverage as more faults can be detected, which could lead to reducing the number of restored test sequences. These two steps, Relaxation followed by Random filling are done once each time a test sequence is restored and if the fault coverage of the compacted test sequence increases, the process is repeated.

Algorithm 4.8 Subsequence Fault Coverage Increasing LROR

1. Fault Simulate the circuit using the given test set.
   Collect the detection time of each fault.
2. Restore the first k test vectors as a synchronizing
   sequence from the given test set T. C=$\{v_1$,v_2,v_3,.....$v_k\}$.
3. Fault simulate the restored sequence C and drop all
   the faults detected from F_{target}. Store the (S_g/S_f)
   values of all the flip-flops for all undetected faults.
4. **if** $(F_{target} = \emptyset)$ Return C **else** Go to Step 5.
5. V = Test Restoration(n, F_n), where n is the last
   time frame having undetected faults.
6. V = State Traversal(V, F_n, F_{target})
7. $C = C$ & V;
8. **while**(fault coverage of C increases **&** $F_{target} \neq \emptyset$) {
   C=Relaxation(C)
   RandomFill(C) }
9. Go To Step 3.

The performance of SFC-LROR is shown in tables 4.9 to 4.12. Table 4.9 shows the one-shot version of all the best known compaction schemes on STRATEGATE test sequences [9]. In addition it shows the improvement made by SFC-LROR on our RX-LROR-ST2 implementation. It can be seen that SFC-LROR has made significant improvement on our implementation of RX-LROR-ST2. It has shown improvements on 12 out of 16 circuits and overall savings are also significantly higher. This trend is even more pronounced on HITEC [59] test sequences, shown in Table 4.10. On HITEC test sequences, SFC-LROR has again performed better than RX-LROR-ST2 on 15 out of 16 circuits and achieved much higher overall savings. It is worth mentioning that this fault coverage increasing scheme is generic and can be applied on top of any static compaction scheme. These results basically show very strong potential that any scheme can benefit from it and may achieve higher level of compaction.

Table 4.9 also shows the overall performance of SFC-LROR in comparison to the best known compaction algorithms. In comparison to LROR [15], it has shown better results on 11 out of 16 circuits, while one resulted in a draw. In comparison to LROR [13], it performed better on 8 out of 16 circuits. In comparison to SIFAR [14], it performed better on 6 out of 16 circuits. MISC [13] applies vector omission on the restored subsequences found by LROR [13], before appending those to the compacted test set. MISC has performed better than SFC-LROR on 10 out of 16 circuits with higher overall savings.

Next, Table 4.10 shows the performance of various compaction schemes on HITEC [59] test sequences. In comparison to SECO [36], SFC-LROR has performed better on all 16 circuits with savings well over 1700 test vectors. In comparison to SIFAR [14], it has performed better on 8 out of 16 circuits. In comparison to LROR [13], it has performed better on 9 out of 16 circuits. Finally, MISC has again shown better results than SFC-LROR on 10 out of 16 circuits with higher overall savings.

Table 4.9: Comparison of the best known Static Compaction algorithms on STRATEGATE Test Sequences

STRATEGATE Test Sequences													
		LROR [15]		LROR [13]		SIFAR [14]		MISC [13]		RX-LROR-ST2		SFC-LROR	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	138	0.14	125	0.2	**116**	**0.1**	123	0.2	152	0.06	150	0.19
s344	86	62	0.09	47	0	48	0.1	**44**	**0.1**	**44**	**0.04**	52	0.25
s382	1486	606	1.49	**536**	**1.2**	559	1.2	555	2.1	625	0.26	606	2.26
s444	1945	642	2.15	577	1.3	596	1.7	**459**	**2.4**	614	0.36	677	4.1
s526	2642	1269	8.86	1069	3.7	1171	3.1	**1004**	**6.3**	1890	0.67	1445	15.22
s641	166	118	0.13	91	0.1	87	0.2	**74**	**0.2**	119	0.08	80	0.37
s713	176	139	0.16	112	0.2	125	0.2	92	0.2	112	0.08	**85**	**0.77**
s820	590	489	0.79	401	0.9	423	0.9	**356**	**1.5**	456	0.29	449	15.86
s832	701	543	0.89	475	1.4	511	1.1	**375**	**2.9**	498	0.36	444	23.19
s1196	574	277	0.28	234	0.5	251	0.6	234	0.6	268	0.31	**225**	**16.23**
s1238	625	285	0.31	244	0.5	251	0.7	244	0.8	268	0.36	**228**	**17.17**
s1423	3943	1031	12.87	964	13.5	1082	17.2	**763**	**19.7**	1328	2.72	1155	88.8
s1488	593	501	1.79	**363**	**1.6**	390	1.7	370	2.7	453	0.56	456	41.92
s1494	540	468	1.71	417	5.4	**408**	**1.7**	417	9.5	434	0.57	415	34.13
s5378	11481	677	38.71	734	50.1	**597**	**64.3**	704	81.6	726	45.87	615	226.1
s35932	257	137	56.93	148	195.4	186	95.9	174	159.4	**131**	**21.85**	133	105.9
Total	25999	7382	127.3	6537	276	6801	190.7	**5988**	**290.2**	8118	74.44	7215	592.46

Hybrid-III is another powerful compaction scheme, which combines SFC-LROR and MR. The algorithm re-iterates SFC-LROR until 4 consecutive iterations are unable to reduce the test size. This step is followed by MR, which is reiterated until one iteration of MR does not reduce the test size. MR is again followed by SFC-LROR and the process continues as long as each pass (SFC-LROR followed by MR) reduces the test size. The idea is illustrated by Fig. 4.4.

Table 4.10: Comparison of the best known Static Compaction algorithms on HITEC Test Sequences.

HITEC Test Sequences													
		SECO [36]		SIFAR [14]		LROR [13]		MISC [13]		RX-LROR-ST2		SFC-LROR	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec
s298	322	216	0.8	**129**	**0.2**	169	0.1	139	0.2	187	0.05	157	0.3
s344	127	61	0.3	50	0.1	**47**	**0**	48	0.1	54	0.06	55	0.31
s382	2074	878	6	**353**	**1.6**	430	0.8	387	1.7	890	0.44	689	3.11
s444	2240	1005	7.5	480	2	349	0.8	**315**	**1.4**	842	0.43	820	6.04
s526	2258	1526	12	649	3.2	607	1.2	**370**	**3.3**	1544	0.71	1525	15.68
s641	209	125	0.8	112	0.2	105	0.1	102	0.2	135	0.1	**87**	**0.91**
s713	173	106	0.8	93	0.2	89	0.1	88	0.1	105	0.07	**68**	**0.91**
s820	1115	790	5.9	599	1.4	598	0.7	**496**	**1.3**	631	0.5	541	24.2
s832	1137	779	6.5	597	1.5	605	0.7	**484**	**1.4**	636	0.46	548	26.98
s1196	435	281	2.2	256	0.6	251	0.5	252	0.6	291	0.29	**236**	**16.85**
s1238	475	303	2.7	272	0.7	266	0.5	267	0.7	302	0.29	**245**	**19.54**
s1423	150	134	4.2	160	1.3	111	0.5	110	0.9	142	0.25	**109**	**3.2**
s1488	1170	828	10	**613**	**2.8**	647	1.5	643	3.8	774	1	698	54.81
s1494	1245	855	11	640	3	630	1.6	**605**	**3.6**	815	1.1	763	56.77
s5378	912	653	39	456	8	300	9	292	13.2	451	4.5	**287**	**63.11**
s35932	496	202	487	183	59.3	161	48.1	**160**	**65.7**	249	34.31	198	1072.5
Total	14538	8742	596.7	5642	86.1	5365	66.2	**4758**	**98.2**	8048	44.56	7026	1365.22

Table 4.11 and 4.12 compare the iterative versions of the best reported schemes in literature with iterative version of SFC-LROR and Hybrid-III. ITE-SFC-LROR re-iterates until 4 consecutive iterations are unable to reduce the test size.

Table 4.11 compares the iterative versions of best known compaction algorithms on STRATEGATE [9] test sequences. It can be noticed that ITE-SFC-LROR has achieved better results than ITE-RX-LROR-ST2, which are further improved by ITE-Hybrid-III. ITE-Hybrid-III performed better than ITE-SFC-LROR on 14 circuits, while 1 resulted in a draw. In comparison to ITE-LROR [13], ITE-Hybrid-III has performed better on 8 out of 16 circuits. In comparison to ITE-SIFAR [14], it has again performed better on 8 circuits. Finally in comparison to ITE-MISC [13], it has performed better on 5 out of 16 circuits.

Table 4.12 shows the compaction results on HITEC [59] test sequences. It can be seen that ITE-Hybrid-III has performed better than ITE-LROR [13] on 13 out of 19 circuits, while 1 resulted in a draw. In comparison to ITE-MISC [13], it has again performed better in 14 out of 19 circuits. The performance of ITE-Hybrid-III is even more noticeable in circuits s713, s820, s1196, s1238, s1488, s1494, s5378, s3271, s3384 and s4863.

It should be noticed from tables 4.9 to 4.12 that the overall savings of proposed algorithms are lesser than other algorithms primarily because of the following circuits: s382, s444, s526, and s1423. This is due to current limitations of justification algorithms, which are discussed in the next section.

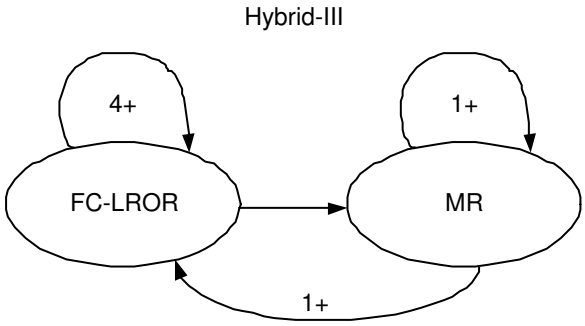


Figure 4.4: Hybrid-III

4.5.2 Subsequence Fault Coverage Increasing MR

This is similar to SFC-LROR with the only difference that subsequences are merged rather than concatenating them into one another. Algorithm 4.9 shows the implementation details of the algorithm. It should be noted that unlike normal MR algorithm, V holds the specified input assignments rather than relaxed assignments. The restored subsequence V is then relaxed considering only the un-detected faults. The relaxed subsequence is then merged from start, in previously restored subsequence C. The subsequence C is then randomly filled and an attempt is made to increase its fault coverage, using relaxation and random filling. The process is re-iterated n times such that there is no further increase in the fault coverage of the subsequence C. It should be noticed that after last pass of relaxation, the un-specified bits are not random filled. This is done to allow Merging algorithm to merge subsequences in C after restoring them for other faults. The algorithm continues until all target faults are detected.

The performance of FC-MR is shown in tables 4.13 and 4.14. Table 4.13 compares the performance of SFC-MR with other best known compaction schemes. It can be noticed that SFC-MR has made significant improvement on MR; saving of more than 1000 vectors is achieved. In comparison to SFC-LROR it has performed better on only 5 out of 11 circuits with significantly lower overall savings. In comparison to all other best known techniques, SFC-MR has shown best results on only 2 out of 11 circuits.

Table 4.14 compares the performance on HITEC [59] test sequences. SFC-MR has again shown significant improvement on MR, it has performed better on all 11 circuits, with overall savings of more than 1,300 test vectors. In comparison to SFC-LROR, it has performed better on only 3 out of 11 circuits with lesser overall savings. Finally, in comparison to all other best known techniques, SFC-MR has shown best results on 3 out of 11 circuits and overall savings are higher than SECO [36] only.

SFC-MR could not show promising results in terms of compaction quality, in comparison to SFC-LROR. Therefore, the scheme is implemented as a prototype

Algorithm 4.9 Subsequence Merging Restoration based on increasing the Fault Coverage

1. Fault Simulate the circuit using the given test set.
   Collect the detection time of each fault.
2. **if** ($C \neq \emptyset$) Fault simulate the restored sequence C and drop all the faults detected from F_{target}.
3. **if** ($F_{target} = \emptyset$) Return C **else** Go to Step 4.
4. Let $i = n$, where n is the last time frame having undetected faults. Let $V = \emptyset$ be the sequence currently restored.
5. $(S_g/S_f)_i$ = *Justify(F_i,i)* and let $j = i$.
6. **while** $((S_g/S_f)_j \neq X)$ {
   $V = V_j$ & V //add current time frame to C
   $j = j - 1$ //move back single time frame
   $(S_g/S_f)_j$ = Justify(F_i, j) //Get the states for all
   //flip-flops reached in this new time frame
   } //end while
7. $V = Relaxation(V, F_{un-detected})$
8. $C = MergeStart\,(C, V)$
9. **for**(k= 1 to n) {
   RandomFill(V) // 1 is used as a seed
   // to the random number generator.
   $C = Relaxation(C)$
   }
10. Go To Step 2.

Table 4.11: Comparison of the Iterative Versions of the best known Static Compaction algorithms on STRATEGATE Test Sequences.

STRATEGATE Test Sequences													
		ITE LROR [13]		ITE SIFAR [14]		ITE MISC [13]		ITE RX-LROR-ST2		ITE SFC-LROR		ITE Hyb-III	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec
s298	194	125	0.6	112	0.4	**98**	**3.2**	152	0.25	116	3.47	101	5.1
s344	86	47	0.1	48	0.2	**43**	**0.4**	44	0.18	52	1.31	49	2.46
s382	1486	524	3.3	**498**	**5.6**	507	17.8	622	1.13	604	18.33	595	32.58
s444	1945	461	7.9	**454**	**7.2**	455	31.3	614	1.12	546	19.35	540	34.26
s526	2642	945	26.6	975	8.9	**884**	**209.8**	1714	3.36	1356	87.56	1350	202.23
s641	166	78	0.5	87	0.4	63	1.7	118	0.57	62	6.19	**59**	**13.76**
s713	176	72	0.6	94	1.1	60	0.8	111	0.5	61	7.5	**57**	**11**
s820	590	394	6.4	388	6.5	**335**	**15.2**	428	1.98	391	98.8	374	204.9
s832	701	458	8.8	435	4.5	**368**	**14**	460	2.44	402	118.27	374	377.24
s1196	574	221	1.7	237	3.4	216	3.2	266	1.34	215	157.6	**180**	**273.31**
s1238	625	222	2.6	251	1.5	222	3.6	266	1.8	202	155.64	**185**	**285.54**
s1423	3943	843	108.3	778	113	**702**	**469.2**	1207	11.93	1082	868.3	1082	2105.2
s1488	593	343	27.1	**312**	**8.8**	364	39.4	423	4.36	402	249.84	396	492.22
s1494	540	297	33.8	313	6.8	**296**	**72.6**	434	2.53	385	111.9	359	234.67
s5378	11481	711	339.4	597	89.5	583	2148	651	92.97	**490**	**1333.61**	**490**	**1985.6**
s35932	257	110	752.3	152	290	**101**	**1177**	125	132.9	125	526.8	125	680.8
Total	25999	5851	1320	5731	547.8	**5297**	**4207.2**	7635	259.36	6491	3764.5	6316	6940.9

only and this is why timing is not shown in the tables.

4.6 Limitations of Justification Algorithm

The proposed compaction scheme is based on the test relaxation algorithm in [19], which involves justification of the required values to detect a fault. The justification algorithm has limitations that may lead to the restoration of longer test sequences than necessary.

The justification algorithm is guided by cost functions in case of having several choices for justifying a value. In order to minimize the length of the extracted sequence, a multiplicative weight of 10 or 100 is applied to flip-flop cost functions whenever a flip-flop is reached during cost function computation. The approximate nature of the computed cost functions may guide the justification algorithm to a choice that leads to the extraction of a longer test sequence.

Another limitation is during justification of faulty values. Since the justification algorithm is based on the cost of good values only, it may lead to the selection of inferior choices. For instance, to justify 1/0 at the output of a gate, the algorithm considers the cost of justifying 0 as a very high number, as the good value of the gate is 1; it may choose this path only when there is no cheaper choice available or it is the only choice.

Table 4.12: Comparison of the Iterative Versions of the best known Static Compaction algorithms on HITEC Test Sequences.

HITEC Test Sequences											
		ITE LROR [13]		ITE MISC [13]		ITE RX-LROR-ST2		ITE SFC-LROR		ITE Hyb-III	
Ckt	TS	TS	sec	TS	sec	TS	sec	TS	sec	TS	sec
s298	322	109	0.8	**97**	**1.1**	148	0.37	157	0.92	153	2.55
s344	127	47	0.1	47	0.5	54	0.17	55	1.45	**46**	**10**
s382	2074	288	2.1	**279**	**77.5**	890	1.32	689	15.41	683	28.55
s444	2240	343	2.3	**302**	**22.4**	842	1.58	820	31.17	815	59.16
s526	2258	404	9.3	**352**	**31.8**	1544	2.97	1327	101.4	1321	160.75
s641	209	**63**	**1**	72	1.2	123	0.5	**63**	**4.77**	**63**	**7.41**
s713	173	74	0.7	74	1	105	0.34	53	4.76	**53**	**8.34**
s820	1115	578	13.8	432	28.3	600	3.78	380	243.1	**359**	**394.74**
s832	1137	562	8.3	383	64	581	3	397	216.28	**381**	**475.52**
s1196	435	226	2.3	223	2.5	281	2.34	212	150	**187**	**221.35**
s1238	475	227	1.9	225	1.9	288	2	215	189.36	**190**	**252.48**
s1423	150	111	1.1	110	1.6	142	0.94	**109**	**15.52**	**109**	**36.14**
s1488	1170	571	10.4	572	354.6	765	4.8	457	796.4	**451**	**902.9**
s1494	1245	540	9.1	492	274	758	4.9	489	655.9	**331**	**2156.8**
s5378	912	245	108.1	271	189	327	35.26	**212**	**561.36**	**212**	**912.72**
s35932	496	142	227.8	**117**	**1158**	225	199.1	159	2547.6	159	2879.6
s3271	709	555	24.6	443	265	683	13.26	537	2304.5	**332**	**3441.43**
s3384	161	104	11.6	92	13.1	131	3.5	95	216.68	**81**	**456.86**
s4863	518	302	20.5	315	25.6	366	23.9	221	797	**139**	**1299.1**
Total	15926	5491	455.8	4898	2513.1	8853	304.03	6647	8853.6	6065	13706.4

This is illustrated in Fig. 4.5. Suppose that it is required to restore a test subsequence for the fault k s-a-0 from the given test sequence (10, 11, 11, 11).

The fault is detected in the last time frame and justifying the value 1/0 on the output leads to the requirement of 1/0 on line m and X/0 on line n. The faulty value on line m is justified through the fault site and this leads to the requirement of 1/X on input A. There are no justification requirements on input B and hence it can be relaxed.

The value X/0 on line n has to be justified in the previous time frame (i.e., time frame 3). This will lead to the justification requirement X/0 on line o in time frame 3. Notice that this value could be justified through line p from the fault site leading to the test sequence (XX, 1X). However, the algorithm uses cost functions based on good values only; the cost of having a 0 on line p is a very high number. Thus, line p is not selected and line q is selected as it has lesser cost. This causes the algorithm to go back until the first time frame, where the value is justified through

Table 4.13: Performance of SFC-MR on STRATEGATE Test Sequences.

Ckt	TS	LROR [15]	SIFAR [14]	LROR [13]	MR	SFC-MR	SFC-LROR	MISC [13]
s298	194	138	116	125	154	141	150	**123**
s344	86	62	48	47	61	50	52	**44**
s641	166	118	87	91	148	79	80	**74**
s713	176	139	125	112	140	87	**85**	92
s820	590	489	423	401	531	497	449	**356**
s832	701	543	511	475	568	509	444	**375**
s1196	574	277	251	234	242	**199**	225	234
s1238	625	285	251	244	248	**212**	228	244
s1488	593	501	390	363	533	591	456	**370**
s1494	540	468	**408**	417	501	460	415	417
s5378	11481	677	**597**	734	1549	809	615	704
Total	15726	3697	3207	3243	4675	3634	3199	**3033**

input B. Thus, the algorithm will return the test sequence (X0, 1X, 1X, 1X), which has redundant vectors.

One way to address this limitation is by computing the cost of faulty values (in addition to good values) for a single time frame. This could be computed by injecting the faulty values at the fault site and processing the circuit level-by-level. These two cost functions would better guide the justification process for good and faulty values separately, using respective cost functions.

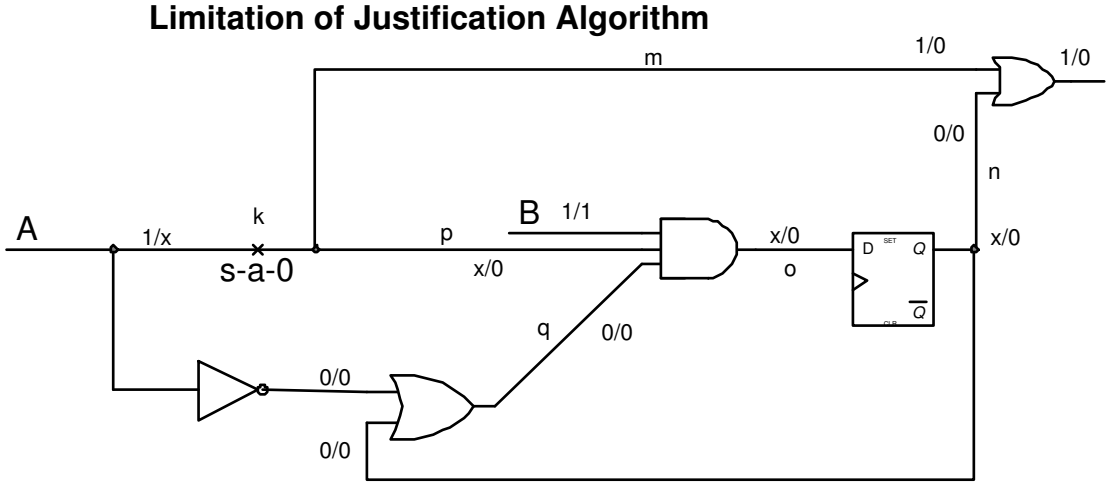
Second and more exact method to address this limitation is by setting an upper limit to the size of restored subsequence. During the justification process, if the restored subsequence reaches that upper limit, it could then be fault simulated using the target fault list. In case of failure in fault detection, the justification algorithm would continue until it justifies the fault or it reaches the upper limit again. In either case a pass of fault simulation would restrict the size of subsequence. This scheme is a compromise between large test size due to in-exact nature of cost functions and expensive vector-by-vector fault simulation to find the exact starting point of the subsequence as used by LROR.

The second limitation of current implementation is the memory requirement. Currently, our technique stores all faults that get excited and propagated, even if they are not detected. The memory usage can be significantly reduced by storing only information about propagated faults from the time of their excitation to detection. A two pass fault simulations scheme, as proposed by Hsiao *et al.* [3] can be used to find exactly those time frames where the faults are excited, propagated and reach to the primary output. For faults that require large test sequence, the algorithm can be altered to run in phases to store only the good and faulty values

Table 4.14: Performance of SFC-MR on HITEC Test Sequences.

Ckt	TS	SECO [36]	SIFAR [14]	LROR [13]	MR	SFC-MR	SFC-LROR	MISC [13]
s298	322	216	129	169	207	175	157	**139**
s344	127	61	50	**47**	69	59	55	48
s641	209	125	112	105	158	**81**	87	102
s713	173	106	93	89	129	72	**68**	88
s820	1115	790	599	598	863	709	541	**496**
s832	1137	779	597	605	879	694	548	**484**
s1196	435	281	256	251	255	**213**	236	252
s1238	475	303	272	266	269	**228**	245	267
s1488	1170	828	**613**	647	911	711	698	643
s1494	1245	855	640	630	974	781	763	**605**
s5378	912	653	456	300	706	357	**287**	292
Total	7320	4997	3817	3707	5420	4080	3685	**3416**

in a predetermined number of frames to reduce the memory requirements. This will require running the fault simulator in phases to determine the required good and faulty values for the set of frames across which the faults are going to be justified. These ideas will be investigated in future work.



	A B	A B	A B	A B
Input Assignments	1 0	1 1	1 1	1 1
Relaxed Assignments	X 0	1 X	1 X	1 X

Figure 4.5: Extraction of longer test sequence.

Chapter 5

Proposed Static Compaction algorithms for Combinational Circuits

This chapter discusses two static compaction algorithms for combinational circuits. The algorithms include the following:

- Fault Detection Frequency-Based Clustering (FFC)
- Class based Clustering (CBC)

This chapter presents the two algorithms, tabulate the results and compares their behavior with other static compaction techniques.

5.1 Fault Detection Frequency-Based Clustering (FFC)

This algorithm is based on test vector decomposition and is an improved version of a combinational algorithm by El-Maleh *et al.* presented in [1].

We first describe the algorithm, Secondly, an illustrative example is given. Thirdly, the iterative version of the algorithm is described. Finally, experimental results are reported.

5.1.1 Algorithm Description

In Independent Fault Clustering (IFC), IFSs are first derived. Then, a fault matching procedure is used to find sets of compatible faults, i.e., faults that can be detected by a single test vector. In the IFS derivation phase, independent faults are

identified with respect to a test set. In the fault matching phase, compatible components, corresponding to compatible faults, are mapped to the same compatibility set. Whenever a component is mapped to a compatibility set, it is merged with the partial test vector of that compatibility set. At the end, every compatibility set represents a single test vector.

The FFC algorithm is shown in Algorithm 5.1 and proceeds as follows. First, the given test set T is fault simulated without fault dropping. This step is performed to find the number and set of test vectors that detect every fault. Second, all the faults are sorted using their detection frequency in ascending order, this step identifies those faults, which are detected by one or fewer test vectors. Next, essential faults are matched. In this step, for every essential fault f detected by t, the atomic component c_f corresponding to f is extracted from t. Then, for every compatibility set CS_i, if c_f is compatible with the partial test vector in CS_i, c_f is mapped to CS_i. On the other hand, if the number of compatibility sets is zero or c_f is incompatible with all partial test vectors in the existing compatibility sets, a new compatibility set is created and c_f is mapped to it.

It should be observed that an essential fault has a single component while non-essential faults have more than one. Therefore, if a component of a non-essential fault f is incompatible with all the partial test vectors in the existing compatibility sets, the other components of f should be tried before creating a new compatibility set. On the other hand, if the component of an essential fault is incompatible with all the partial test vectors in the existing compatibility sets, a new compatibility set must be created. Hence, essential faults are matched first.

Next, the algorithm fault simulates the existing compatibility sets and drops all the faults detected. This step saves the computation time, which is otherwise spent on extracting atomic components of yet un-mapped, non-essential faults and then either mapping them to existing compatibility sets or creating a new compatibility set for such faults.

The algorithm then focuses on remaining un-mapped, non-essential faults. This step exhaustively checks every component of non-essential fault and therefore attempts to minimize creating a new compatibility set. It extracts the atomic component of every fault and attempts to map it to an appropriate compatibility set. For every fault, an atomic component of a fault f is extracted, if it is incompatible with all partial test vectors in the existing compatibility sets, a new component is tried. In this step, a new compatibility set is created only if the number of compatibility sets is zero, which is possible only when there are no essential faults.

At this point, only those non-essential faults remain which require a new compatibility set and none of their atomic component could be mapped to any of the partially filled existing compatibility sets. The algorithm randomly fills the partially filled test vectors of existing compatibility sets and then fault simulates all the compatibility sets. This is done to maximize the chances of detecting yet un-mapped,

Algorithm 5.1 FFC(T)

1. Fault simulate T without fault dropping.
   1.1. Record the number of test vectors detecting each fault.
2. Group the faults by their detection count.
   2.1. Sort the faults in ascending order of their detection count.
3. For every essential fault f that is detected by a test vector t:
   3.1. Extract the atomic component c_f from t.
   3.2. If the number of compatibility sets is zero, create a new compatibility set, map c_f to it, and then go to Step 3.
   3.3. Map c_f to an existing compatibility set, if possible, and then go to Step 3.
   3.4. Create a new compatibility set and map c_f to it.
4. Fault simulate all the compatibility sets and drop all the remaining (Non Essential) faults that are detected.
5. For the remaining non-essential (un-detected) fault(s) f that is detected by a set of test vector T':
   5.1. For every test vector t', where t' is a member of T':
   5.2. Extract the atomic component c_f from t'.
   5.3. If the number of compatibility sets is zero, create a new compatibility set, map c_f to it, and then go to Step 5.
   5.4. Map c_f to an existing compatibility set, if possible, and then go to Step 5, otherwise go to Step 5.1.
6. Random fill test vectors of all the compatibility sets.
7. Fault simulate all the compatibility sets and drop all the remaining (Non Essential) faults that are detected.
8. For the remaining non-essential (un-detected) fault(s) f that is detected by a set of test vector T':
   8.1. For every test vector t', where t' is a member of T':
   8.2. Extract the atomic component c_f from t'.
   8.3. Map c_f to an existing compatibility set, if possible, and then go to Step 8, otherwise go to Step 8.1.
   8.4. Create a new compatibility set and map c_f to it.
9. Random fill all the vectors of T^*.
10. Return T^*.

non-essential faults and therefore save an extra compatibility set. It should be noted that random filling in step 6 does not affect compaction, since it is guaranteed that none of the remaining test vector components could map to any of the existing partially filled test vectors.

Finally, the algorithm creates an additional compatibility set for the remaining un-mapped, non-essential faults, only if all components of a fault f are incompatible with all partial test vectors in the existing compatibility sets. At the end, the algorithm randomly fills the remaining partially filled test vectors and returns the compatibility sets as the new test set.

5.1.2 Illustrative Example

We next illustrate the steps of the proposed FFC algorithm through an example. Table 5.1 shows a set of four test vectors along with their detected faults and the components generated for each fault. Faults f1 and f2 are essential faults and are clustered first resulting in two sets as shown in Table 5.2. We assume in this example that fault simulating the resulting compatibility sets do not detect additional faults. Then, faults with detection count=2 are clustered i.e., faults f3, f4, f5, f7 and f8. The first component of f3=x0xxxx11x1 are attempted for clustering and is found to be incompatible with the existing sets. The second component of f3=x0x1x1xxx1 is then successfully clustered into the second set. The first component of f4=00xx1xx1xx is successfully clustered into the first set. However, none of the components of the faults f5, f7 and f8 could be clustered in the existing sets and hence their clustering is delayed. Next, clustering is attempted for faults of detection count=3 i.e., f6. While neither the first nor the second components of f6 could be clustered into the existing sets, the third component of f6=00xx11xx0x is successfully clustered into the first set. Next, the algorithm randomly fills the merged test vectors of the compatibility sets and fault simulates the remaining undetected faults i.e., f5, f7 and f8. We assume in this example that fault f5 is detected by the randomly filled test vectors. Finally, f7 and f8 are clustered. The first component of f7=1x1x1xx10x is mapped to a new set. Then, the first component of f8= x1xx1xx001 is found incompatible with the third set and hence it's second component is attempted. The second component of f8= xxx11x0101 is then found compatible with third set and is clustered with it creating the merged test vector 1x111x0101, which is randomly filled to create a fully specified test set. Thus, the test set is compacted into the following three test vectors: 0001110100, 1011011001, 1110100101.

5.1.3 Iterative FFC

The level of compaction achievable by the IFC (or FFC) algorithm can be improved in two ways. First, after a component is generated for a fault, the component is

Table 5.1: Example Test Vectors and their components.

Test Vector		**Fault Detected**	**Fault Component**
v1	0000111110	*f1*	0x0xx1xxx0
		f4	00xx1xx1xx
v2	1101101001	*f2*	1xx1xx10x1
		f5	xxxx10x0xx
		f6	1xx1x0x0x1
		f8	x1xx1xx001
v3	1010111101	*f3*	x0xxxx11x1
		f5	xxxx1xx101
		f6	1xxx11x10x
		f7	1x1x1xx10x
v4	0011110101	*f3*	x0x1x1xxx1
		f4	00xxx1x1x1
		f6	00xx11xx0x
		f7	xx1x1x0x0x
		f8	xxx11x0101

fault simulated and the faults detected by it are marked as detected. In this way, a large portion of the faults will not be considered subsequently since they are already detected. Based on our experimental investigations [1], we noticed that this extra step increases the runtime and improves the results very little. Second, the FFC algorithm can be called on a test set iteratively. Basically, the new test set generated is treated as the test set to be compacted. Therefore, FFC is carried out iteratively until the length of the test set cannot be reduced in last six iterations, we call this procedure *iterative* 6+. Unspecified bits in the test set T are assigned random values before every call to the FFC algorithm.

It should be pointed out that any static compaction algorithm can be used after the FFC algorithm. In fact, given a test set T, the FFC algorithm will generate a new test set T^* whose characteristics are different from the characteristics of T. Thus, a static compaction algorithm that cannot compact T may manage to compact T^*.

5.1.4 Experimental Results

In order to demonstrate the effectiveness of the FFC algorithm, we have performed experiments on a number of the ISCAS85 and full-scanned versions of ISCAS89

Table 5.2: Illustration of steps of FFC on the given example.

After mapping faults with detection cnt=1		**After mapping faults with detection cnt=2**		**After mapping faults with detection cnt=3**		**After Merging Components**	**After Random Filling**
Fault	**Fault Component**	**Fault**	**Fault Component**	**Fault**	**Fault Component**	**Test Vector**	**Test Vector**
f1	0x0xx1xxx0	*f1*	0x0xx1xxx0	*f1*	0x0xx1xxx0	000x11x100	0001110100
		f4	00xx1xx1xx	*f4*	00xx1xx1xx		
				f6	00xx11xx0x		
f2	1xx1xx10x1	*f2*	1xx1xx10x1	*f2*	1xx1xx10x1	10x1x110x1	1011011001
		f3	x0x1x1xxx1	*f3*	x0x1x1xxx1		

benchmark circuits. The experiments were run on a Pentium Mobile, with 2.0 GHz processor and 1GB DDR2 RAM. We have used test sets generated by HITEC [59]. In addition, we have used the fault simulator HOPE [60] for fault simulation purposes and the test relaxation algorithm in [17] for component generation.

Table 5.3 summarizes the features of benchmark circuits we have used in our experiments. The first column gives the circuit name. Columns two through eight give the number of primary inputs, number of primary outputs, number of gates, number of Test Vectors (TVs), number of Collapsed Faults (CFs), number of Detected Faults (DFs), and Fault Coverage (FC), respectively.

In Table 5.4, we compare the test compaction results of IFC [1] and FFC algorithms when applied on the original test set. The first column gives the circuit name. The second column specifies the number of test vectors in the original test set before applying any compaction. The third and fourth columns give test set sizes after applying Reverse-Order Fault simulation (ROF) and Random Merging (RM) [17], respectively. ROF is based on applying reverse-order and random order fault simulation for 20 iterations. RM is based on relaxing the test vectors generated by ROF and merging compatible vectors. Columns five and six give the results of the IFC algorithm [1] while columns seven and eight report the results of the proposed FFC algorithm. Test set sizes are given under the column headed TV. The CPU time required by each of the algorithms is given under the column headed 'sec'. It should be noted that the two algorithms (IFC and FFC) are executed on the same machine for ease of comparison. The FFC algorithm has shown better compaction quality on 12 out of 15 circuits, while 2 circuits resulted in a draw. In terms of overall savings, FFC has saved more than 120 test vectors than IFC [1] (with an average compaction improvement of 7%). For example, for the circuits c3540 and c5315, FFC achieved 24% and 25% higher compaction than IFC, respectively. It should also be noted that FFC consumes significantly lesser CPU time. It

Table 5.3: Features of benchmark circuits used in our experiments.

Cct	# Inputs	# Outputs	# Gates	# TVs	# CFs	# DFs	FC
c2670	233	140	1193	154	2747	2630	95.741
c3540	50	22	1669	350	3428	2895	84.452
c5315	178	123	2307	193	5350	5291	98.897
s13207.1f	700	790	7951	633	9815	9664	98.462
s15850.1f	611	684	9772	657	11725	11335	96.674
s208.1f	18	9	104	78	217	217	100
s3271f	142	130	1572	256	3270	3270	100
s3330f	172	205	1789	704	2870	2870	100
s3384f	226	209	1685	240	3380	3380	100
s38417f	1664	1742	22179	1472	31180	31004	99.436
s38584f	1464	1730	19253	1174	36303	34797	95.852
s4863f	153	120	2342	132	4764	4764	100
s5378f	214	228	2779	359	4603	4563	99.131
s6669f	322	294	3080	138	6684	6684	100
s9234.1f	247	250	5597	620	6927	6475	93.475

has shown 13.37 times overall improvement than IFC [1]. In order to increase the level of compaction, FFC can be applied in an iterative manner until no compaction improvement is possible. We have experimented with an iterative version of FFC, called FFC6+, by applying FFC iteratively until the length of the test set cannot be reduced in the last six iterations. Unspecified bits in the test set T are assigned random values before every call to the FFC algorithm. Columns nine and ten in Table 5.4 report the results of an iterative version of IFC applied on the test generated by ROF, called ROF+ITER-IFC [1]. Columns eleven and twelve report the results of FFC6+. It can be seen that FFC6+ has achieved higher test compaction than ROF+ITER-IFC on 12 out of 15 circuits, while 2 resulted in a draw. For example, for the circuit c5315, FFC6+ has achieved 29% more compaction than ROF+ITER-IFC. Furthermore, it has shown higher overall savings (with an average compaction improvement of 8%) in a much more efficient CPU time (ranging from 1 to 14 times less CPU time). It should be observed that FFC6+ consumes more time on s15850 at the expense of more compaction as the algorithm continued on iterating due to more compaction improvements achieved.

It should be pointed out that any static compaction algorithm can be used after the proposed FFC algorithm. In fact, given a test set T, the FFC algorithm will generate a new test set T* whose characteristics are different from the characteristics of T. Thus, a static compaction algorithm that cannot compact T may manage to

Table 5.4: Comparison of Compaction results

	Original	**ROF**	**RM**	**IFC [1]**		**FFC**		**ROF+Itr-IFC[1]**		**FFC 6+**	
Circuits	**TV**	**TV**	**TV**	**TV**	**sec**	**TV**	**sec**	**TV**	**sec**	**TV**	**sec**
c2670	154	106	100	98	0.993	98	0.04	85	42.07	**82**	**3.93**
c3540	350	83	80	99	2.01	75	1.01	75	26.95	**63**	**5.05**
c5315	193	119	106	107	3.97	80	1.96	86	88.04	**61**	**10.94**
s13207.1f	633	476	252	244	34.06	238	10.02	238	473.12	**234**	**69.02**
s15850.1f	657	456	181	142	50.97	144	15.97	129	374.95	**118**	**1365.98**
s208.1f	78	33	33	34	0.001	**32**	**0.001**	**32**	**0.01**	**32**	**0.01**
s3271f	256	115	76	60	1.95	59	1.93	60	18.98	**55**	**3.95**
s3330f	704	277	248	238	3.05	230	0.99	196	30.02	**192**	**4.2**
s3384f	240	82	75	**72**	**1.98**	**72**	**0.96**	**72**	**7.07**	**72**	**2.98**
s38417f	1472	822	187	150	838	130	225.95	120	3775.06	**108**	**2337**
s38584f	1174	819	232	148	4718	138	154.02	124	8217.08	**114**	**1735.17**
s4863f	132	65	59	50	3.02	47	3.95	42	70.88	**38**	**6.96**
s5378f	359	252	145	120	3.05	119	1	117	109	**107**	**13.99**
s6669f	138	52	42	40	7.91	36	5.02	30	175.01	**28**	**12.02**
s9234.1f	620	375	202	182	11.06	170	3.04	155	200.93	**139**	**27.04**
Total	7160	4132	2018	1784	5680.024	1668	425.861	1561	13609.17	**1443**	**5598.24**

Bold face highlights the best results

compact T*.

It should also be noted that FFC consumes significantly lesser CPU time. It has shown 13.37 times improvement than our previous implementation of IFC [1].

5.2 Class Based Clustering

This algorithm is based on test vector decomposition (TVD) [1]. Each test vector is decomposed into components based on the faults the test vector covers. In this research, we defined two types of components, i.e., essential and unessential components. An essential component is the one that covers an essential fault, which is the fault that is covered by only one test vector. The main two ideas in our algorithm are the following:

1. A sufficient condition for eliminating a test vector is to ensure that its essential components can be moved to other test vectors (while previous algorithms requires that all components must be movable).

2. During component movements, we simulate the new vectors using their specified forms, this will cover more faults compared to the un-specified forms. To ensure that the coverage pool is not affected, we fill the un-specified inputs based on the values of the original vectors.

Another new feature of our algorithm is its parametric design, which allows for a compromise between the algorithm computation time and performance. That has been accomplished by dividing the algorithm into iterations. In each iteration, a number of test vectors are eliminated. Running more iterations increases the number of eliminated vectors at the expense of run time and vice versa.

Experimental results for benchmark circuits demonstrate the effectiveness of the algorithm. It also compares the performance and computation time between this algorithm and our previous work [1].

5.2.1 Previous Work: Class Based Clustering

PRELIMINARIES

Given the set of components of every test vector in a test set, a test vector can be eliminated if all its components can be moved to other test vectors. Moving a component to a test vector is implemented by merging the component with the destination test vector. Even though the idea is very simple, it is not always possible to move a component to a new test vector. This is because of two problems: (1) blocking and (2) conflicting components. In the former, a component ci is blocked from being moved to a test vector t when it becomes incompatible with it. ci becomes incompatible with t when another component cj that is incompatible with ci is moved to t. In the later, however, a test vector can not be eliminated if it contains at least one conflicting component. A conflicting component cannot be moved to any other test vector in the given test set.

Definition 1, *Conflicting Component*: A component c of a test vector t belonging to a test set T is called a Conflicting Component (CC) if it is incompatible with every other test vector in T.

The number of CCs in a test vector determines its degree of hardness. The degree of hardness of a test vector is basically a measure of how hard a test vector is to eliminate. Test vectors can be classified based on their degree of hardness.

Definition 2, *Degree of Hardness of a Test Vector*: A test vector is at the nth degree of hardness if it has n CCs.

Definition 3, *Class of a Test Vector*: A test vector belongs to class k if its degree of hardness is k.

A CC can be moved to a test vector t if the characteristics of t are changed. That is, a CC ci is movable to a test vector t, if the components in t incompatible with ci are moved to other test vectors. The set of test vectors to which ci can be moved is referred to as the set of candidate test vectors of ci. A test vector whose CCs are all movable is referred to as a potential test vector.

Definition 4, *Movable CC*: Let ci be a CC in a test vector ts, B be a set of components in a test vector td such that ci is incompatible with every component cj in B, Sj be the set of test vectors compatible with cj. Then, ci is movable to td if and only if $Sj \cap B \neq \emptyset$ for every cj in B.

Definition 5, *Set of Candidate Test Vectors of a CC*: The set of candidate test vectors of a CC ci, denoted by *Scand ci*, contains all test vectors to which ci can be moved.

Definition 6, *Potential Test Vector*: Let α be the set of CCs in a test vector t. t is a potential test vector that belongs to class $|\alpha|$ if and only if for every CC ci in α, ci is movable.

5.2.2 The Original Algorithm Description [1]

The CBC algorithm is based on the idea of dividing test vectors into classes and then heuristically processing test vectors of every class. A test vector is eliminated if its components can be all moved to other test vectors. Eventually, in the final test set, every test vector represents a cluster whose components originally belong to test vectors in different classes. This is why the technique is called CBC.

The CBC algorithm is shown in Figure 5.1 and proceeds as follows. First, the given test set is fault simulated without fault dropping. This step is performed to find the number and set of test vectors that detect every fault. Secondly, test vectors are sorted in increasing order of their number of faults. Then, atomic components of test vectors are generated. Component generation is performed such that components are extracted from essential test vectors. An essential test vector is a test vector that detects at least one essential fault. The component generation algorithm is shown in Figure 5.2 and proceeds as follows. For every fault f detected by t, if the number

of test vectors that detect f is one, that is, f is an essential fault, the component of f is extracted from t; otherwise, the number of test vectors that detect f is reduced by one. Therefore, a test vector that detects no essential faults is eliminated. The sorting step preceding component generation improves the number of eliminated test vectors. Note that a component of a fault is extracted from a test vector that detects a large number of faults.

Algorithm CBC

1. Fault simulate *T* without fault dropping. (By HOPE)
2. Sort test vectors in increasing order of their number of faults.
3. Generate atomic components.
4. Sort test vectors in decreasing order of their number of components.
5. Remove redundant components using fault dropping simulation.
6. For every test vector, merge its components together.
7. Classify test vectors.
8. Process class zero test vectors.
9. For every test vector, merge its components together.
10. Reclassify test vectors.
11. Process class one test vectors .
12. For every test vector, merge its components together.
13. Classify test vectors.
11. Process *i* one test vectors, where *i*>1.

Figure 5.1: CBC algorithm.

After obtaining the set of components of every test vector, test vectors are sorted in decreasing order of their number of components. This helps maximize the number of redundant components. Redundant components are dropped using fault simulation with dropping. After that, every test vector is reconstructed by merging its components together. Finally, test vectors are classified and processed.

Class zero test vectors are processed as shown in the above Figure 5.3. First, test vectors are sorted in increasing order of their number of components. This way a test vector with a small number of components has a higher chance of getting eliminated. After that, for every test vector, its blockage value is computed. The

Algorithm Gen Comp
1. For every test vector t :
1.1. For every fault f detected by t :
1.1.1. If f is *essential* : a. Extract the atomic component c_f from t .
Else b. Decrement the number of test vectors detecting f by one.

Figure 5.2: Component generation algorithm.

Algorithm for processing class zero test vectors:
1. Sort class zero test vectors in increasing order of their number of components.
2. For every class zero test vector, compute its blockage value.
3. For every class zero test vector *t* whose blockage value is zero:
3.1. Move components of *t* to appropriate test vectors.
3.2. Eliminate *t*.
3.3. Update *Scomp* of components belonging to other class zero test vectors.
3.4. Update the blockage values of other class zero test vectors.
4. Sort class zero test vectors in increasing order of their number of components.
5. For every remaining class zero test vector *t*:
5.1. If components of *t* can be all moved:
5.1.1. Move components of *t* to appropriate test vectors.
5.1.2. Eliminate *t*.
5.1.3. Update *Scomp* of components belonging to other
class zero test vectors.

Figure 5.3: The Algorithm for processing Class zero.

blockage value of a test vector t, denoted by *TVB(t)*, can be defined as the sum of the blockage values of the individual components making up t. This can be shown mathematically as follows:

$$TVB(t) = \sum_{i=1}^{Number\_of\_Components} CB_{ci}$$

where *CBci* is the blockage value of component ci belonging to the set of components of t and *NumComp* is the number of components making up t.

CB (ci) is mathematically defined as follows:

CB (ci) = *Min (CB(ci, tj))*

Processing of class one and remaining classes are shown in Figure 5.4 and 5.5.

Algorithm for processing class one test vectors.

1. For every class one test vector *t*:
   - 1.1. Find S_{cand} of the CC.
   - 1.2. If $S_{cand} \neq \phi$, mark *t* as potential.
2. For every class one potential test vector *t*:
   - 2.1. For every test vector in S_{cand}, find the number of class one potential test vectors whose CCs can be moved to it.
   - 2.2. Sort test vectors in S_{cand} according to their types.
3. Sort class one potential test vectors in decreasing order of the number of non potential test vectors in S_{cand}.
4. For every unprocessed class one potential test vector t_p^1:
   - 4.1. Merge t_p^1 . Denote by *td* the test vector to which the CC of *t*1 *p* has been moved.

4.2. If t_p^1 has been eliminated, then for every class one potential test vector t_p^2 whose CC can be moved to *td* , merge t_p^2.

Figure 5.4: The Algorithm for processing Class one.

Algorithm Proc Remaining Classes ($Class_i$, $i >= 2$)

1. For every class *i*, where *i* > 1:
   - 1.1. Find the set of class *i* potential test vectors.
   - 1.2. For every class *i* potential test vector t_p:
     - 1.2.1. For every CC in t_p:
       - a. Move the CC to an appropriate test vector; otherwise, go to Step 1.2.
       - b. Reclassify test vectors.
     - 1.2.2. If all CCs in t_p have been moved:
       - a. Move remaining components.
       - b. Reclassify test vectors.

Figure 5.5: Algorithm for processing remaining classes.

5.2.3 The Proposed Algorithm

DEFINITIONS

- tv_c : Merged test vector, i.e. the result of merging its components together.
- tv_{ec} : Essentially merged test vector, i.e. the result of merging its essential components.
- tv: A test vector that is composed of only {0,1}. It is generated from tv_r by setting each X value to either 0 or 1.
- *fault_list*: The list of all collapsed faults in the circuit. Each *fault_list* $[fault_i]$ is a structure that stores information about $fault_i$. This information includes: how many and which test vectors detect $fault_i$.
- *Essential fault*: A fault that is covered by only one test vector.
- *Essential component*: A component for an essential fault.
- *Unessential component*: A component for unessential fault, which is the fault that is covered by more than one test vector.
- *Compatible test vector*: A test vector is said to be compatible with a component if the component is compatible with all of the components of the test vector.
- *Essentially compatible test vector*: A test vector is said to be essentially compatible with a component if the component is compatible with all of the essential components of the test vector.
- S: The set of all possible eliminated test vectors. A test vector $tv \in S$ if and only if all components of S are movable or can be dropped.

ALGORITHM OVERVIEW

The main problem in the previous work in processing class 1 and higher classes was moving the conflicting components (CCs). To solve this problem, we distinguished between two types of components: *Essential component* and *Unessential component*. An essential component is the one that is associated with an essential fault, that is, a fault that is covered by only one test vector. When a CC (Conflicting Component) is to be moved, we first check, if this component is unessential, then we can just drop the component without affecting the fault coverage (FC_T). If the conflicting component CC is essential, and there is a destination test vector tv such that CC is ensured to be compatible with the essential components of tv, then we can move CC to tv. After CC component movement, we drop all unessential components of

the destination tv which are conflicting with CC. In this way, many of the CC can be dropped or moved.

Furthermore, if CC to be moved is essential and we cannot find a compatible tv_{ec}, then we try to find a new component for the essential fault associated with CC and try to move the new component to a compatible tv_{ec}. The process is as follows: let f be the essential fault associated with the essential conflicting component CC, then we search the original test set (T) for a all test vectors tv that cover the fault f, if we find such, then we generate a new component by relaxing (tv,f) hopefully that the new generated component will be compatible with a tv_{ec} from the current test set. If we could find a tv_{ec} that is compatible with the new component, then we discard the original component and move the new one to the compatible vector.

Based on the component type (essential or unessential) and the component compatibility, we divide the components into 4 types:

Type-1	A component that has a compatible test vector.
Type-2	An essential CC that has essentially compatible test vector.
Type-3	An essential CC which has no essentially compatible test vector. Further, there is a test vector from the original test set (T) from which we can generate a new component covers the same fault associated with CC, this new has an essentially compatible test vector. Type 3 components may happen during the compaction process when some un essential components become essential, due to removal of Type-4 components.
Type-4	Unessential CC. (This component can be simply dropped).

The Algorithm

Figure 5.6 shows the main algorithm. Initially, each test vector must be decomposed into components (steps 1-6). Then, our iterative algorithm starts (7.1-7.3) reducing the size of the test set. In the first steps (1-6), the given test set is fault simulated without fault dropping. This step is performed to find the number and set of test vectors that detect every fault. Secondly, test vectors are sorted in increasing order of their number of faults. Then, atomic components of test vectors are generated as in Figure 5.2. The sorting step preceding component generation improves the number of eliminated test vectors. Note that a component of a fault is extracted from a test vector that detects a large number of faults.

After test vectors are decomposed into components, the iterations start by trying to reduce the original test set T (step 7.1). In the subsequent iterations, the resulting test vectors is changed by generating another test vectors compatible with the last

Processing Vectors (T , *NumIter* **):**

Inputs T : a test set of size $|T|$ with a fault coverage FC_T .
NumIter : the maximum number of iterations the algorithm will run

Outputs T^* : a new test set such that, $|T^*| \leq |T|$ and $|FC_{T^*}| \geq |FC_T|$

1- Fault Simulate the set of test vectors without fault dropping
2- Sort test vectors in increasing order of their number of faults.
3- Generate atomic components.
4- Sort test vectors in decreasing order of their number of components.
5- Remove redundant components using fault dropping simulation.
6- For every test vector, merge its components together.
7- Loop for *NumIter*
  7.1 T^* **= Compact_Test_Vectors (**T**)**
  7.2 $T = T^*$
  7.3 T = **Change_Simulate(**T **)**
- End Loop

Figure 5.6: The Proposed algorithm.

test vectors using the algorithm '*Change_Simulate*' (step 7.3), as shown in Figure 5.7. This process changes the faultlist status and thus, changes essential components. This will hopefully increase the probability of eliminating more test vectors during the next run of *Compact_Test_Vectors* function.

The algorithm: *Compact_Test_Vectors* is shown in Figure 5.9. It works as follows:

First, the essential components are identified, as shown in Figure 5.8. Identification of essential components is done as follows: First, set all component as unessential. Then, search tv for the component of f and then set this component as essential.

The second step in 'Compact_Test_Vectors' is the generation of the set of test vectors that are candidate to be eliminated. A test vector can be eliminated if all of its components can be either moved or discarded. This is done using the 'Eliminable(tv)' function which is shown in Figure 5.10. Since moving components may block some components which were previously movable, we need to re-check to make sure that the vector is still eliminable before starting to move each of its components. This is done using the function '*Still_Eliminable(tv)*' which is shown in Figure 5.12. Since the destination un-relaxed vector *comp_tv* may change because c_i will be moved to the relaxed $comp\_tv_r$, the *fault_list* detected by this vector may change. Therefore, the *fault_list* will be updated, as follows: the vector *comp_tv* will be deleted from the *fault_list*. Then, the component c_i will be merged to the relaxed vector $comp\_tv_r$. If the moved component is essential, then the merge of essential

Change_Simulate(T)

Inputs T : a test set of size $|T|$ with a fault coverage FC_T .

Outputs T^* : a different test set such that $|FC_{T^*}| \geq |FC_T|$ and $|T^*| = |T|$

For all test vectors $tv \in T$ do the following:

Delete tv and update *fault_list* accordingly

Loop n times

1- For all inputs i, if $tv_c(i) = X$ then $tv(i)$ = Random (Zero , One)

2- Simulate the new tv vector, and update *fault_list* accordingly

3 – If FC_T decreases, undo steps 1&2 else exit the loop

Figure 5.7: Change Simulate procedure.

Mark_Essential_Faults_Components

Inputs test vectors set , *fault_list*

Outputs the test vectors set will be updated to include the essential faults components

---

For all test vectors $tv \in T$, for all components $c \in tv$, do the following:

Set c as unessential (initially)

For all essential faults f,

Let tv be the vector that covers f

Search tv for the component of f *and* then set this component as essential

Merge the new component with merged essential components .

Figure 5.8: Mark Essential Faults Components procedure.

components will be updated by merging c_i to it. Also, *comp_tv* is updated to be compatible with $comp\_tv_r$, and finally the new *comp_tv* vector will be simulated and the *fault_list* will be updated accordingly. After moving/discarding all components of the eliminable vector, we check that there is no drop in the fault coverage. If there is a drop in the fault coverage, then we quickly undo components movement (we store the state of destination vectors before try to eliminate a vector). The reason for the fault coverage drop is that we use the specified vector form in simulation (to cover the maximum possible faults and thus decrease essential components count). The specified vector is formed by merging the components of the vector and filling the unspecified entries based on the original values of the vector.

Figure 5.10 shows the '*Eliminable(tv)*' function. This function returns true if all components of the test vector *tv* can be either moved or discarded. This function relies on the core function '*Is_Component_Movable_or_Discardable (c, tv)*' which shown in Figure 5.11.

The core function '*Is_Component_Movable_or_Discardable (c, tv)*' shown in Figure 5.11 tries to find whether a component *c* of a test vector *tv* can be moved or discarded. Also, the algorithm specifies the component type and the destined vector and stores this information in the component record.

The '*Still_Eliminable* function shown in Figure 5.12 is very similar to *Movable* function shown in Figure 5.10. Moving components may block some components which were movable and also some destination vectors for movable components may be eliminated. Therefore, before trying to eliminate a test vector, we should re-check to make sure that the destination vectors of movable components are not eliminated and still compatible with the movable component. This can be done using *Eliminable* function, but this will be time consuming since most the components do not change their states. To speed up the check process, we have introduced the *Still_Eliminable* function. This function will check the results obtained by the previous call of *Eliminable* and make sure that the movable components destination is still compatible and not eliminated, if not, then it will search for a new destinations.

Figure 5.13 shows the function that update the state of essential components. This is important because after moving components and re-simulation, some unessential faults may become essential and vice versa. By storing the state of *fault_list* before component movements, we can compare it with the current state of *fault_list* after movements to get the new essential faults and the previous essential faults that became unessential.

Compact_Test_Vectors (T)

Inputs T : a test set of size $|T|$ with a fault coverage FC_T .

Outputs T^* : a new test set such that, $|T^*| \leq |T|$ and $|FC_{T^*}| \geq |FC_T|$

---

1. Mark_Essential_Faults_Components

2. Generate set of all vectors that are nominative for elimination, as follows:
   2.1 S = Set of eliminable test vectors = { }
   2.2 For all vectors *tv*, if **Eliminable(*tv*)** is true then add *tv* to S

3. Sort S in increasing order of their number of components. Then, For all vectors $tv \in S$,
   3.1 if **Still_Eliminable(*tv*)** is true then try to eliminate the vector as follows:
   3.2 Tag *tv* as eliminated and update *fault_list* accordingly
   3.3 **Update_Essential_Faults_Components**
   3.4 Loop for all components $c_i \in tv$ do the following: *(Components movements)*
   If c_i is type-4, then just discard it
   Else let $comp\_tv$ = the compatible vector, move c_i to $comp\_tv$ as follows:
   - delete $comp\_tv$ from vectors set and update *fault_list*
   - add c_i to the components of $comp\_tv$
   - merge c_i with $comp\_tv_c$
   - if c_i is essential, then merge c_i with $comp\_tv_{ec}$
   - generate $comp\_tv$ (specified vector) by filling the X's of $comp\_tv_c$ based on the previous values.
   - simulate the new $comp\_tv$ vector
   - **Update_Essential_Faults_Components**

   3.5 If fault coverage decreased or one of the components of *tv* becomes unmovable (due to the change in essential components set) then UNDO *tv* elimination by restoring state of the vector *tv* and state of all vectors which the components of *tv*

Figure 5.9: The Compact Test Vectors procedure.

Eliminable(*tv*)

Inputs *tv*: a test vector

Outputs Boolean : TRUE if all components can be either moved or discarded

---

For all components $c \in tv$, do the following:

Is_Component_Movable_or_Discardable (*c* , *tv*)

Return TRUE if all component are either movable or discard able.

Figure 5.10: The Eliminable procedure.

Is_Component_Movable_or_Discardable (*c* , *tv*)

Inputs *c* : a component to be moved or discarded
tv : The test vector which the component resides
Outputs component type , destination vector

// check if the component *c* is type-1
Search for a test vector that is compatible with the component *c*
If a compatible test vector is found then
    Set the component *c* as type-1
    Store the index of the compatible test vector in *c* record
    Exit from the algorithm
End if

// check if the component *c* is type-4
If the component *c* is unessential then
    Set the component *c* as type-4 // discarded //
    Exit from the algorithm
End if

// check if the component *c* is type-2
Search for a test vector that is essentially compatible with the component c
If an essentially compatible test vector is found then
    Set the component *c* as type-2
    Store the index of the essentially compatible test vector in *c* record
    Exit from the algorithm
End if

// check if the component *c* is type-3
Let *f* = the essential fault covered by the essential component *c*
Search for a test vector *tv2* from the original test vectors, such that *tv2* covers *f* and the relaxed component (*tv2*,*f*) has an essential compatible test vector
If such test vector *tv2* is found then
    Set the component *c* as type-3
    Store the relaxed component ((*tv2*,*f*) in *c* record
    Store the index of the essentially compatible test vector (*tv2)* in *c* record
End if

If none of the above searches hit, so, Set the component *c* as not movable nor discard able.

Figure 5.11: Is Component Movable or Discardable procedure.

Still_'Eliminable (*tv*)

Inputs *tv*: a test vector
Outputs Boolean : TRUE if all components still can be either moved or discarded

For all components $c \in tv$, if c is movable (types 1, 2 or 3) , then
if the destination vector is eliminated or not compatible with c any more, then try to move or discard c using the function: **Is_Component_Movable_or_Discardable (*c* , *tv*)**
End if

Return TRUE if all components are either movable or cab be discarded.

Figure 5.12: Still Eliminable procedure.

Update_Essential_Faults_Components

Inputs test vectors set , *fault_list* , *fault_list* state before component movements
Outputs the test vectors set will be updated to include the essential faults components

---

Let F_b = Set of all essential faults before moving a component C
Let F_a = Set of all essential faults after moving the component C

E = set of new essential faults = $F_a - F_b$
U = set of unessential faults that was essential before movements = $F_b - F_a$

For all $f \in E$ // New essential components
Let *tv* is the vector that covers *f*
If *tv* has a component for *f* then set this component as essential
else generate a new component for *f* and set it as essential.
Merge the new component with merged essential components

For all $f \in U$ // Previous essential components that became unessential
Let *c* is the essential component of *f*, and this component is in vector *tv*
Set *c* as unessential.
Re-Merge the essential components of the vector *tv*.

Figure 5.13: Update Essential Faults Components procedure.

5.2.4 Experimental Results

This section compares the performance of CBC with other static compaction algorithms.

In order to demonstrate the effectiveness of the proposed test compaction algorithm, we have performed experiments on a number of the ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits. We have used test sets generated by HITEC [59]. In addition, we have used the fault simulator HOPE [60] for fault simulation purposes and the test relaxation algorithm in [17] for test vector component generation.

Table 5.5 compares the test compaction results of the proposed algorithm with previously proposed CBC algorithm. The second, third, fourth and fifth columns report the number of inputs, outputs, test vectors and fault coverage for each circuit. The sixth and seventh columns report the number of test vectors achieved after test compaction by the CBC algorithm and the proposed algorithm respectively. For all the circuits, the proposed algorithm achieves significantly higher compaction than CBC algorithm. The improvement in compression varied from 5% for circuit s13207 to 50% for circuit c880, with an average improvement of 31%.

Table 5.6 compares the CPU time taken by the proposed algorithm for 1 and 10 iterations in comparison with CBC algorithm. The efficiency of the proposed algorithm is clearly demonstrated by the large benchmark circuits.

Table 5.7 shows the statistics of moved component according to which type they belong. Most of the moved components are either compatible with an existing test vector (Type 1) or compatible with the essential components of an existing test vector (Type 2). It is interesting to observe that a good percentage of the components are dropped due to being unessential. A small percentage of the conflicting components have been regenerated from the original test set due to conflict will essential components of all test vectors.

Figure 5.14 and 5.15 shows the effect of the number of iterations on the compacted test set size for circuits s9234 and c7552. As can be seen, after applying 10 iterations of the algorithm a good level of compaction improvement is achieved.

Table 5.5: Test compaction results comparison with CBC algorithm.

Cct	# Inp	# Out	# TVs	Fault Coverage	#TVs after running CBC alg.	#TVs after running our alg.
c880	60	26	200	0.940552	52	26
c2670	233	140	154	0.957408	106	83
c3540	50	22	350	0.8483	99	65
c5315	178	123	193	0.988972	105	60
c7552	207	108	198	0.893510	101	51
s1512f	29	21	200	0.8879	32	24
s6669f	83	55	138	1.00	51	27
s9234.1f	36	39	620	0.9348	180	141
s13207.1f	62	152	478	0.9846	247	234
s15850.1f	611	684	456	0.966738	152	137 *1i

Table 5.6: Processing time comparison with CBC algorithm.

Cct	processing time for CBC alg.	Processing time for our algorithm (1 iteration)	Processing time for our algorithm (10 iterations)
C880	0.01	0.9	2.7
C2670	3.05	1	7
C3540	0.007	2	9
C5315	7.05	9	25
C7552	3.07	25	49
S1512f	0.04	0.1	1
S6669f	11.08	10.1	26.2
S9234.1f	790	106.9	153
S13207.1f	4345	111	205
S15850.1f	4144.0	176	274

Table 5.7: Component movement statistics after processing original test vector.

Cct	Total Moved Components	(1) Compatible with a TV	(2) Compatible with ess. Components of a TV	(3) Extracted from Org. TV	(4) Dropped Uness components
c880	198	22.7%	29.8%	6.1%	41.4%
c2670	74	23.0%	45.9%	4.1%	27.0%
c3540	775	7.2%	73.8%	1.7%	17.3%
c5315	523	49.5%	26.4%	2.3%	21.8%
c7552	643	42.1%	21.2%	4.7%	32.0%
s1512f	211	67.3%	16.6%	3.3%	12.8%
s6669f	486	60.7%	24.5%	1.4%	13.4%
s9234.1f	1244	78.5%	12.5%	0.5%	8.4%
s13207.1f	592	88.3%	7.9%	0.7%	3.0%
s15850.1f	841	87.6%	8.2%	0.4%	3.8%

Figure 5.14: Test set size vs. iterations for the circuit s9234.1f.

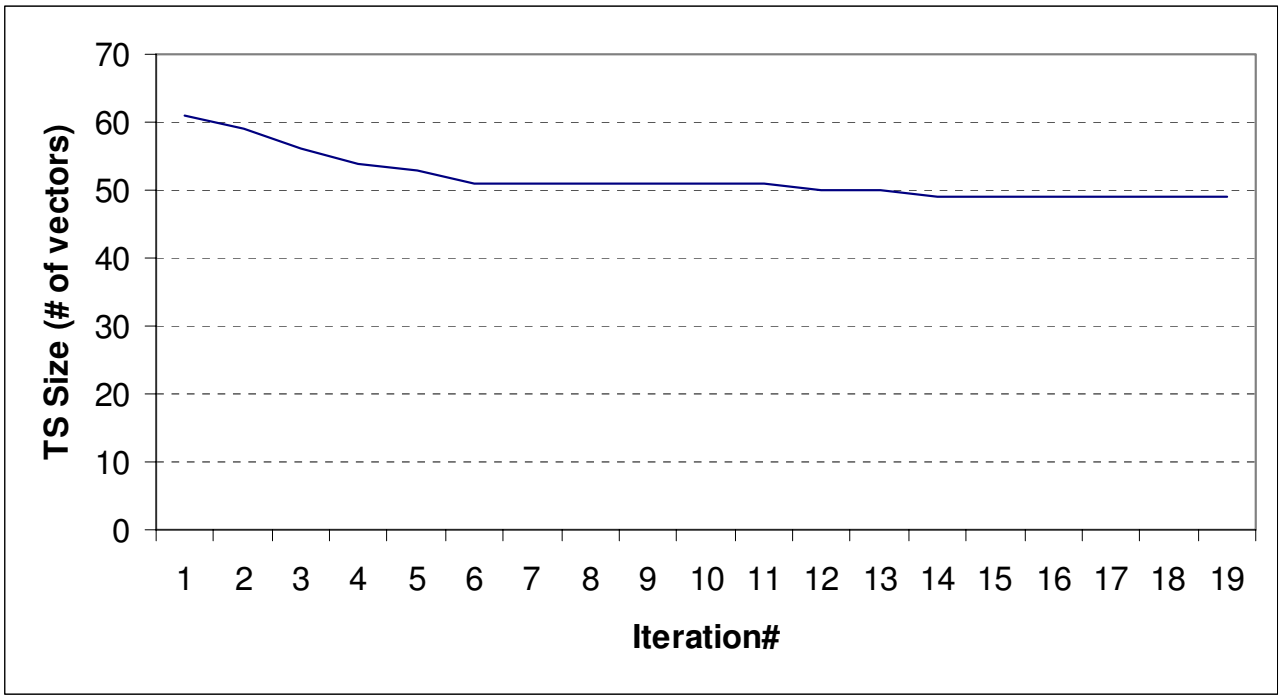


Figure 5.15: Test set size vs. iterations for the circuit c7552.

5.2.5 Conclusion

In this research, we could effectively reduce a given test vectors set using test vectors decomposition. Each test vector is decomposed into components. We distinguished between essential and unessential components. This distinction made more freedom for components to move around vectors, and thus allowed for more vectors elimination.

Chapter 6

Conclusions and Future Research

In this project, we have presented a detailed literature review of most of the static compaction algorithms for combinational and sequential circuits. In addition, we have proposed several static compaction algorithms for combinational and synchronous sequential circuits based on an efficient test relaxation scheme. The proposed work has the advantage of quickly finding a self-initializing test sequence for a set of faults compared to simulation based technique, which rely on fault simulations. The restored subsequence can be further compacted by state traversal algorithm, which allows the removal of redundant vectors without additional fault simulations. These restored subsequences can be either concatenated (having fully specified bits; making RX-LROR) or they can be merged (relaxed input assignments, Merging Restoration). Merging Restoration is found to be more effective after applying RX-LROR as demonstrated by ITE-Hybrid-II and ITE-Hybrid-III. We have proposed a new class of static compaction algorithms based on increasing the fault-coverage of restored subsequences, which have shown the potential of reducing the test set size significantly. Finally, we have also proposed an efficient way of taking any compaction algorithm out of saturation. This is achieved by using test relaxation and randomly filling the unspecified bits before re-iterating the algorithm.

In future, we plan to address the following issues: The results of the proposed algorithms can be improved in terms of test sequence length, CPU time and memory requirement by addressing the limitations of the justification algorithm. Furthermore, the technique used to increase the fault coverage of subsequences is not the best known, for this purpose. In future we plan to use better techniques for increasing the fault coverage and intuitively that should further improve the level of compaction.

For combinational circuits, we suggest the following improvements to CBC algorithm.

- As the experimental results showed, by changing the state of the test vectors randomly, we could eliminate more test vectors from the test set. We changed

the state of a test vector by merging its components and then setting the 'X" value randomly to 0 or 1 and then simulate. Another state change that may be implemented easily is moving components randomly between test vectors. After each iteration, we dedicate some time (say 5-10 seconds) to move the components randomly between vectors. Hopefully, this will solve the problem of some essential conflicting components

- The big task in our algorithm is to find a compatible vector for an essential fault component. We could solve this problem by trying to find an another component for the essential fault from the original fault list (type-3). According to the experimental results, table3, this gives only little percentage of the moved components. We try to solve this problem by generating random vectors, hopefully that this vectors will cover the essential fault, but this method does not work.

A suggested solution (which we do not implement) is as follows: Add another processing level, which implements known heuristics to search for all possible set of vectors that covers the essential fault and extract its component from each vector. If an extracted component has a compatible vector in the test vectors set then the task is done.

Bibliography

[1] Aiman H. El-Maleh and Yahya E. Osais. Test Vector Decomposition Based Static Compaction Algorithms for Combinational Circuits. *ACM Transactions on Design Automation of Electronic Systems*, 8(4):430–459, Oct. 2003.

[2] R. Roy, T. Niermann, J. Patel, J. Abraham and R. Saleh. Compaction of ATPG-Generated Test Sequences for Sequential Circuits. *in Proceedings of International Conference on Computer-Aided Design*, pages 382–385, November 1988.

[3] M. S. Hsiao, E. M. Rudnick and J. K. Patel. Fast Static Compaction Algorithms for Sequential Circuit Test Vectors. *IEEE Transactions on Computer*, 48(3):311–322, March 1999.

[4] Andreas Steininger. Test and Built-In Self Test – A Survey. *Systems Architecure*, 46:721–747, 2000.

[5] Abhijit Jas and Nur Touba. Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs. In *Proceedings IEEE International Test Conference (ITC)*, pages 458–464, Washington, DC, 1998. IEEE Computer Society Press.

[6] Sybille Hellebrand, Steffen Tarnick, Bernard Courtois, and Janusz Rajski. Generation of vector patterns through reseeding of multipe-polynominal linear feedback shift registers. In *International Test Conference*, pages 120–129, 1992.

[7] A. Jas and N. Touba. Deterministic test vector compression/decompression for systems-on-a- chip using an embedded processor. *Journal of Electronic Testing: Theory and Applications (JETTA)*, 18(4):503–514, Aug 2002.

[8] Thomas Marchok, Aiman El-Maleh, Wojciech Maly, and Janusz Rajski. A Complexity Analysis of Sequential ATPG. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(11):1409–1423, Nov 1996.

[9] M. S. Hsiao, E. M. Rudnick and J. H. Patel. Sequential Circuit Test Generation using Dynamic State Traversal. In *Proc. European. Design & Test Conf.*, pages 22–28, March 1997.

[10] Irith Pomeranz and Sudhakar M. Reddy. Vector Restoration-Based Static Compaction Using Random Initial Omission. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(11):1587–1592, November 2004.

[11] Irith Pomeranz and Sudhakar M. Reddy. A new approach to test generation and test compaction for scan circuits. In *Proc. Design Automation Test Eur.*, pages 1000–1005, 2003.

[12] Gert-Jan Tromp. Minimal Test Sets for Combinational Circuits. In *Proc. of the International Test Conference*, pages 204–209. IEEE, 1991.

[13] R. Guo, S. M. Reddy and Irith Pomeranz. Reverse-Order-Restoration-Based Static Test Compaction for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(3):293–304, March 2003.

[14] Xijiang Lin, W u Tung Cheng, Irith Pomeranz, and Sudhakar M. Reddy. SIFAR: Static Test Compaction for Synchrnous Sequential Circuits Based on Single Fault Restoration. In *Proc. of the VLSI Test Symposium*, pages 205–212. IEEE, 2000.

[15] R. Guo, Irith Pomeranz and S. M. Reddy. On Speeding-Up Vector Restoration Based Static Compaction of Test Sequences for Sequential Circuits. pages 467–471, December 1998.

[16] I. Pomeranz and S. M. Reddy. Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits. *in Proceedings of International Conference on Computer Design*, pages 360–365, October 1997.

[17] Aiman El-Maleh and Ali Al-Suwaiyan. An Efficient Test Relaxation Technique for Combinational and Full-Scan Sequential Circuits. In *Proc. of the VLSI Test Symposium*, pages 53–59, Monterey, CA, 2002. IEEE.

[18] Seiji Kajihara and Kohei Miyase. On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits. In *IEEE/ACM Int'l Conference on Computer-Aided Design*, pages 364–369, San Jose, CA, USA, Nov. 2001. IEEE.

[19] Aiman El-Maleh and Khaled Al-Utaibi. An Efficient Test Relaxation Technique for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):933–940, June 2004.

[20] Irith Pomeranz and S. M. Reddy. Vector Replacement to Improve Static-Test Compaction for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):336–342, February 2001.

[21] Irith Pomeranz and S. M. Reddy. Sequence Reordering to Improve the Levels of Compaction Achievable by Static Compaction Procedures. *Proceedings of the Conference on Design Automation and Test in Europe*, pages 214–218, March 2001.

[22] Srimat T. Chakradhar and Anand Raghunathan. Bottleneck Removal Algorithm for Dynamic Compaction in Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(10):1157–1172, October 1997.

[23] I. Pomeranz and S. M. Reddy. Dynamic Test Compaction for Synchronous Sequential Circuits using Static Compaction Techniques. *in Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 53–61, June 1996.

[24] I. Pomeranz and S. M. Reddy. Procedures for Static Compaction of Test Sequences for Synchronous Sequential Circuits. *IEEE Transactions on Computers*, 49(6):596–607, June 2000.

[25] I. Pomeranz and S. M. Reddy. On Static Compaction of Test Sequences for Synchronous Sequential Circuits. *in Proceedings of the ACM/IEEE* 33^{rd} *Design Automation Conference*, pages 215–220, June 1996.

[26] M. S. Hsiao and S. T. Chakradhar. State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits. *in Proceedings of Design Automation and Test in Europe, DATE98*, pages 577–582, February 1998.

[27] Irith Pomeranz, S. M. Reddy and Ruifeng Guo. Static Test Compaction for Synchronous Sequential Circuits Based on Vector Restoration. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(7):1040–1049, July 1999.

[28] Ruifeng Guo, Irith Pomeranz and S. M. Reddy. Procedures for Static Compaction of Test Sequences for Synchronous Sequential Circuits Based on Vector Restoration. *in Proceedings of Conference on Design Automation and Test in Europe*, pages 583–587, February 1998.

[29] H. K. Lee and D.S. Ha. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *in Proceedings of Design Automation Conference*, pages 336–340, June 1992.

[30] Yoshinobu Higami, Yuzo Takamatsu and Kazo Kinoshita. Test Sequence Compaction for Sequential Circuits with Reset States. 9^{th} *Asian Test Symposium (ATS'00)*, pages 165–170, December 04-06, 2000.

[31] Irith Pomeranz and S. M. Reddy. VERSE: A Vector Replacement Procedure for Improving Test Compaction in Synchronous Sequential Circuits. *in Proceedings of IEEE VLSI Design Conference*, pages 250–255, January 1999.

[32] S. M. Reddy, I. Pomeranz and S. Kajihara. On the Effects of Test Compaction on Defect Coverage. *in Proceedings of* 14^{th} *VLSI Test Symposium*, pages 430–435, April 1996.

[33] Irith Pomeranz and S. M. Reddy. An Approach for Improving the Levels of Compaction Achieved by Vector Omission. *IEEE/ACM International Conference on Computer-Aided Design*, pages 463–466, November 1999.

[34] Irith Pomeranz and S. M. Reddy. Enumeration of Test Sequences in Increasing Chronological Order to Improve the Levels of Compaction Achieved by Vector Omission. *IEEE Transactions on Computers*, 51(7):866–872, July 2002.

[35] S. Bommu, S.T. Chakradhar and K. Doreswamy. Static Test Sequence Compaction Based on Segment Reordering and Accelerated Vector Restoration. *in Proceedings of International Test Conference*, pages 954–961, October 1998.

[36] S. Bommu, S.T. Chakradhar and K. Doreswamy. Static Compaction using Overlapped Restoration and Segment Pruning. *in Proceedings of International Conference on Computer-Aided Design*, pages 140–146, November 1998.

[37] R. Guo, S. M. Reddy and Irith Pomeranz. PROPTEST: A Property Based Test Pattern Generator for Sequential Circuits using Test Compaction. *in Proceedings of Design Automation Conference*, pages 653–659, June 1999.

[38] R. Guo, Irith Pomeranz and S. M. Reddy. On Improving Static Test Compaction for Sequential Circuits. *in Proceedings of* 14^{th} *International Conference on VLSI Design*, pages 111–116, January 2001.

[39] T. M. Niermann, W. Cheng, and J. Patel. PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator. *IEEE Transactions on CAD*, pages 198–207, February 1992.

[40] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonaz Reorda. New Static Compaction Techniques of Test Sequences for Sequential Circuits. In *Proc. of the European Design and Test Conference*, pages 37–43. IEEE, 1997.

[41] Jaan Raik, Artur Jutman, and Raimund Ubar. Fast Static Compaction of Tests Composed of Independent Sequences: Basic Properties and Comparison of Methods. In *Proc. of the International Conference on Electronics, Circuits, and Systems*, pages 445–448. IEEE, Sept. 2002.

[42] Paulo F. Flores, Horacio C. Neto, and Joao P. Marques-Silva. On Applying Set Covering Models to Test Set Compaction. In *Proc. of the Ninth Great Lakes Symposium on VLSI*, pages 8–11, Ypsilanti, MI, USA, Mar. 1999. IEEE.

[43] Kwame Osei Boateng, Hideaki Konishi, and Tsuneo Nakata. A Method of Static Compaction of Test Stimuli. In *Proc. of the Asian Test Symposium*, pages 137–142, Kyoto, Japan, Nov. 2001. IEEE.

[44] Dorit S. Hochbaum. An Optimal Test Compression Procedure for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(10):1294–1299, Oct. 1996.

[45] M. H. Schulz, E. Trischler, and T. M. Sarfert. SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1):126–137, Jan. 1988.

[46] Irith Pomeranz and Sudhakar M. Reddy. Forward-Looking Fault Simulation for Improved Static Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1262–1265, Oct. 2001.

[47] Ilker Hamzaoglu and Janak H. Patel. Test Set Compaction Algorithms for Combinational Circuits. In *Proc. of the International Conference on Computer-Aided Design*, pages 283–289, San Jose, CA, USA, Nov. 1998. IEEE.

[48] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. M. Reddy. Cost Effective Generation of Minimal Test Sets for Stuck-At Faults in Combinational Logic Circuits. *IEEE Transactions on Computer-Aided Design*, 14(12):1496–1504, Dec. 1995.

[49] Xijaing Lin, Janusz Rajski, Irith Pomeranz, and Sudhakar M. Reddy. On Static Test Compaction and Test Pattern Ordering for Scan Designs. In *Proc. of the Int'l Test Conference*, pages 1088–1098, Baltimore, MD, USA, 2001. IEEE.

[50] Bechir Ayari and Bozena Kaminska. A New Dynamic Test Vector Compaction for Automatic Test Pattern Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(3):353–358, March 1994.

[51] Kohei Miyase, Seiji Kajihara, and Sudhakar M. Reddy. A Method of Static Test Compaction Based on Don't Care Identification. In *Proc. of the First IEEE Int'l Workshop on Electronic Design, Test, and Application*, pages 392–395, Christchurch, New Zealand, Jan. 2002. IEEE.

[52] Jau-Shien Chang and Chen-Shang Lin. Test Set Compaction for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(11):1370–1378, Nov. 1995.

[53] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freedman, San Francisco, 1979.

[54] Sheldon B. Akers and Balakrishnan Krishnamurthy. Test Counting: A Tool for VLSI Testing. *IEEE Design and Test of Computers*, 6(5):58–73, Oct. 1989.

[55] Lakshmi N. Reddy, Irith Pomeranz, and Sudhakar M. Reddy. ROTCO: A Reverse Order Test Compaction Technique. In *Proc. of the EURO-ASIC Conference*, pages 189–194, Paris , France, June 1992. IEEE.

[56] Ilker Hamzaoglu and Janak H. Patel. Test Set Compaction Algorithms for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):957–963, Aug. 2000.

[57] Seiji Kajihara, Irith Pomranz, Kozo Kinoshita, and Sudhakar M. Reddy. On Compacting Test Sets by Addition and Removal of Test Vectors. In *VLSI Test Symposium*, pages 25–28, Cherry Hill, NJ, USA, April 1994. IEEE.

[58] M. S. Hsiao, E. M. Rudnick and J. H. Patel. Alternating Strategies for Sequential Circuit ATPG. *European Design and Test Conference*, pages 368–374, 1996.

[59] T. M. Niermann and J. H. Patel. HITEC: A Test Generation Package for Sequential Circuits. In *Proc. Eur. Conf. Design Automation (EDAC)*, pages 214–218, 1991.

[60] Hyung Ki Lee and Dong Sam Ha. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1048–1058, Sept. 1996.

[61] James Mchugh. *Algorithmic Graph Theory.* Prentice Hall, NJ, 1990.

Publications

The following research papers have been published from the work in this project. In addition, three more publications are expected from the work on test compaction for combinational circuits.

Journal Publications

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Efficient Static Compaction Techniques for Sequential Circuits based on Reverse Order Restoration Based and Test Relaxation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems., Vol. 25, No. 11, pp. 2556-2564 Nov. 2006.

Conference Publications

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Reverse Order Restoration Based on Test Relaxation and Subsequence Merging for Efficient Static Compaction for Sequential Circuits*, IEEE European Test Symposium, 22nd to 26th May '05 at Tallinn, Estonia.

- Aiman H. El-Maleh, S. Saqib Khursheed, and Sadiq M. Sait. *Efficient Static Compaction Techniques for Sequential Circuits based on Reverse Order Restoration and Test Relaxation*, IEEE Asian Test Symposium, 18th to 21st Dec'05 Salt Lake City, Kolkata, India.

- Aiman H. El-Maleh, S. Saqib Khursheed. *Efficient Test Compaction for Combinational Circuits Based on Fault Detection Count-Directed Clustering*, IEEE International Design and Test Workshop, 19th to 20th Nov' 06 Dubai, UAE.