POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Corso di Laurea Magistrale in Computer Engineering, Embedded Systems

Tesi di Laurea Magistrale

NEW TECHNIQUES FOR REDUCING THE DURATION OF RECONFIGURABLE SCAN NETWORK TEST

Relatore
Prof. MATTEO SONZA REORDA

Correlatore
Dott. RICCARDO CANTORO

Candidato
LUIGI SAN PAOLO

ANNO ACCADEMICO 2017-2018

Dedicato alle persone speciali della mia vita

Acknowledgements

I would like to acknowledge all the people who made the accomplishment of this thesis possible by guiding, supporting and encouraging me. My thanks goes especially to the professor Matteo Sonza Reorda, advisor and Dr. Riccardo Cantoro, co-advisor for their availability. I also acknowledge all the other people I have worked with in the Politecnico di Torino. Finally, I would also like to thank Valentina, Simona, my family and my friends for their constant encouragement.

Luigi San Paolo

La ricerca è ciò che faccio quando
non so che cosa sto facendo.

Wernher von Braun

Abstract

The growing complexity of electronic devices has created the need for effective access to registers (called *instruments*) for non-functional purposes (e.g., test, debug, calibration). This need has driven to the development of new solutions, such as the new IEEE 1687 standard. This solution allows access to embedded instruments through a Reconfigurable Scan Network, composed of a series of reconfigurable scan chains, possibly using the Boundary Scan interface. To detect the possible permanent faults in these chains, different proposed approaches automatically generate a sequence of input stimuli capable of detecting such faults. The common approach to detect faults is to perform a sequence of test sessions on the IEEE 1687 network. Each session consists of a configuration phase and a test phase. Furthermore, all faults can be covered if the network configurations sequence is selected correctly. The cost to test the network depends on the duration of each session, i.e., of each configuration and test phase. This document discusses the problem of generating a suitable test sequence for a generic Reconfigurable Scan Network and proposes an optimization method based on evolutionary computation with related diagnostics. Experiments on standard networks are also reported.

Contents

Introduction

The modern Integrated Circuits include within them various support functions for testing, such as Built-In Self-Test modules, and sensors capable of measuring parameters such as current, temperature and delays. These features allow to monitor the operation of the integrated circuit and the registers to set up and calibrate the operation of specific modules. To simplify the access to all these resources, called *instruments*, the IEEE 1687 standard [1] was introduced. This standard is the evolution of the IEEE 1149.1 standard, in fact it is based on the scan chains that can also be divided and configured in the most appropriate way. This new approach allows a flexible choice of the best trade-off between different parameters, such as access time or area. This new standard [2] also describes the ways to design configurable networks within the circuit, so in general, we can refer to them as Reconfigurable Scan Networks (RSNs). As a result, the new generation of devices will include Reconfigurable Scan Networks accessible via the Test Access Port (TAP) interface and will support serial access to internal instruments, i.e. Test Data Registers (TDRs). For each access, the networks must be configured to select the subset of instruments to be accessed, and then it will be possible to read / write in series values from / to these selected instruments. In this way, it is possible quickly and efficiently to access the instruments. There are already CAD tools, which automate the introduction and the use of RSNs [3]. The problem of testing the scan chains introduced by the IEEE 1149.1 standard has been addressed by various studies [4] [5] [6]. With the new standard, one must face the problem of testing the hardware of the Reconfigurable Scan Networks, verifying any defects. Common Reconfigurable Scan Networks are composed of chains of flip flops interlaced with special configurable modules, such as Segment Insertion Bits (SIBs) and ScanMuxes. These special modules allow us to divide the network into segments that can be connected in series or in parallel, changing the accessible elements. An approach to check for possible permanent faults affecting a standard scan chain is to move a sequence of alternated 0s and 1s within the scan chain and to check that the same sequence occurs in the output of the chain [4] [5] [6]. The complexity of testing a Reconfigurable Scan Network is higher than the simple scan chain, because it is necessary to verify if the network can be configured correctly and if it works as expected, i.e. the expected sub-network is made accessible. Specifically, the test must check whether each special module in the network is

working properly. In a previous work, a general technique was proposed to automatically generate a test sequence for the detection of permanent faults for an IEEE 1687 network [7]. This approach provided the techniques for testing configurable components, such as SIBs and ScanMuxes. Moreover, the tests generated are independent on the specific implementation of network elements and does not require any modification in the hardware. The generation algorithms were based on heuristic calculations that are easily applicable to relatively large Reconfigurable Scan Networks. In another study, this approach has been improved to minimize the duration of the resulting test sequence by modeling the problem on graphs [8]. Furthermore, an optimal algorithm for small Reconfigurable Scan Networks and sub-optimal solutions for real cases has been described. In this thesis, a new method is proposed, capable of handling even large and complex Reconfigurable Scan Networks and that can produce a sequence of tests to detect any permanent failures on the reconfigurable modules, but whose duration is generally lower than the duration of the test sequences generated by the heuristic solutions previously proposed [7]. The proposed method is based on evolutionary computation and provides a good compromise between the required computational cost and the quality of the solution in terms of test duration. The experimental results are made on the set of benchmark networks described in an article [9] to report the effectiveness of the proposed approach. The paper is organized as follows. In Chapter I we describe the key features of the IEEE 1687 networks and a previously proposed sub-optimal algorithm. In Chapter II we propose the techniques to generate an optimized test sequence for a Reconfigurable Scan Network. The diagnostic method is presented in the Chapter III. Chapter IV reports some experimental results, and Chapter V finally draws some conclusions.

Chapter 1

Background

The Reconfigurable Scan Networks have been a research topic for many years. With the introduction of the IEEE 1687 standard, several problems have arisen concerning design, validation, testing of these structures and their use in the field. In an article, the time required to access all the instruments in the circuit was evaluated in several possible scenarios [6]. The goal of several jobs is to automate the design of the reconfigurable IEEE Std 1687 scanning networks. In a study, an approach has been proposed that can generate networks composed of multiple SIBs that optimize parameters such as area overhead, total access time or average access time [10]. Analysis of different scenarios were presented, such as the position of the configuration bit with respect to the configurable module, in addition to a formal analysis for verification [11]. A formal analysis of the reconfigurable scan networks based on modeling in satisfiability problems (SAT) has been described in a document [12] [13]. In the same document, an analysis of modeling, verification, and modeling issues has been addressed, but without considering the minimization of testing time. The same studies have presented a formal verification methodology for the safety of the reconfigurable scan networks [9], which allow to verify the access protection at the logical level by unbounded model checking. Pattern retargeting is a problem that involves finding the operations necessary to transport the bits of data requested from / to the instrument registers [14]. One way to define specific instrument operations is provided by IEEE Std 1687. The specific problem of configuring the network so that the time to access that segment is minimized has been addressed by several works [14] [15]. The test of the reconfigurable scan networks has been analyzed in some previous works, focusing on maximizing the coverage of structural fault [16]. The main objective of this document is to propose a new method able to manage large and complex Reconfigurable Scan Networks and able to produce a sequence of tests capable of detecting any permanent fault affecting reconfigurable modules, but with a test time lower than that of test sequences generated by heuristic solutions proposed in another article [7]. The following is an overview of the Scan Network and key elements in Reconfigurable Scan Networks, such as ScanMux and SIB, followed by a discussion of the

related fault model and the problem of testing Reconfigurable Network Scan.

1.1 Scan Network

A scan network as introduced by IEEE Std 1149.1 [2], is a suitable on-chip circuitry that provide controllability and observability and thus allow the testing of devices on a board, even when their pin accessibility is very low. On each chip a chain of scan cells is suitably inserted between the pins and the logic. Scan network supports two types of test: the test of the interconnections between components, the test of single components. Test of the chip can be performed by shifting in the values to be applied to the circuit, latching the values of the circuit, shifting out the circuit outputs. Test of the interconnections between circuit A and B can be performed by scanning to circuit A the values to be applied to the interconnections; latching the values received by circuit B, scanning out these values. The Scan Chain can work in two different operating modes:

- normal mode: Scan Chain allow the normal data flow.
- test mode: scan in values; apply values to the core logic or to the output signals; capture values from the core logic or from the input signals; scan out values.

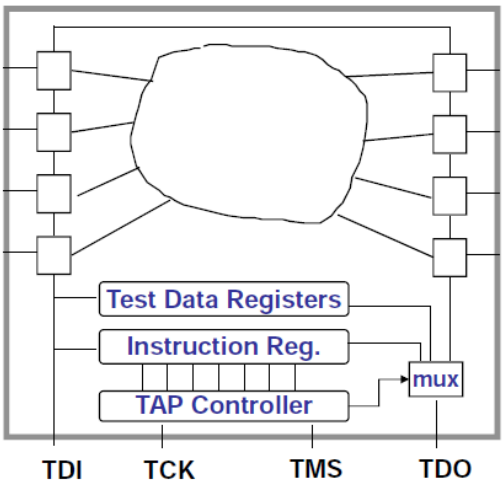

Figure 1.1: Example IEEE Std 1149.1.

The scan network can be accessed via the TAP Test Access Port, which introduces some pins like Test Clock to sample the input signals, Test Mode Select to set up test circuitry behavior, Test Data Input to serially provide data and instructions to test circuitry, Test Data Output to serially send to the output the values in the internal registers. In general, a cell of the Scan Chain is composed of two Flip Flops:

- Capture.
- Update.

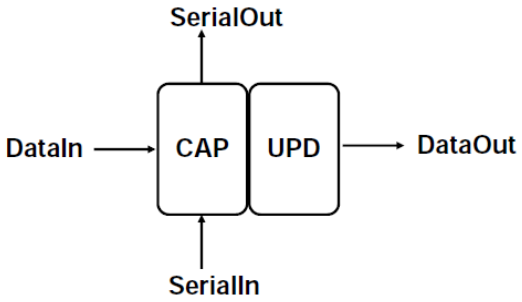

Figure 1.2: Scan cell.

A typical Scan Cell supports 4 operating modes depending on the value of Mode and Shift/Load:

- normal mode: the cell is transparent.
- capture mode: the value on DataIn is latched in the chain by the application of a clock pulse.
- scan mode: cells are connected in a chain through the SerialIn and SerialOut pins; the first cell is connected to TDI, the last to TDO;
- update mode: the value of Q_A is written in QB by applying a pulse to Update; DataOut is forced to Q_B.

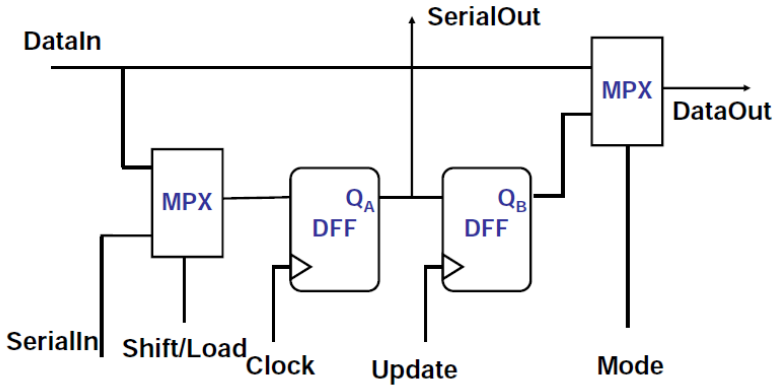

Figure 1.3: Detailed Scan cell.

1.2 Reconfigurable Scan Networks

Post-silicon validation, test, debug, and diagnosis require efficient access to on-chip instrumentation. Reconfigurable Scan Networks, as proposed by IEEE Std 1687-2014 [1] and IEEE Std 1149.1-2013 [2], are often hierarchical and may have complex structural and functional dependencies. Reconfigurable Scan Network of Std 1687, also referred to as IJTAG (Internal JTAG), have introduced the concept of reconfigurable scan chains. This kind of chains are separated in several segments, which are combined with special elements, as reconfigurable modules. Each segment can include one or more instruments. The interface with an instrument is the Test Data Register (TDR), which can include capture logic to read and update logic to write. The active path connected between the scan input and scan output pins of the reconfigurable scan chain at a given time and connect several segments together according to the configuration of reconfigurable modules. Since the complexity of these reconfigurable scan chains can be high (i.e., many possible active paths may exist), the standards refer to them as networks. Each Test Data Register can be constructed as a chain of multiple segments, some of which are always scanned while others, called excludable segments and selectable segments, are scanned only in particular configuration. One of the most important structures the standard defines is called Segment Insertion Bit (SIB), which represents the concept to an excludable segment. The excludable segment of a Test Data Register is controlled by a configuration module (Segment Insertion Bit) composed of one bit, which eventually excludes the segment from the active path of the network, and is followed by a switching element controlled by the configuration module. The selectable segments of a TDR are segments, even of different lengths, which are connected to a selection circuit implemented by means of scan multiplexer (ScanMux) modules. According to the value of the configuration module composed of one or more bits, only one segment at a time is selected as the scanout for the set. When a SIB is said to be asserted, the segment it controls is included in the active path; otherwise, it is said to be de-asserted. Each segment controlled by a SIB or a ScanMux can be a complex network itself. To access an instrument in a reconfigurable scan network, a scan-in bit sequence must be generated according to the current state and structure of the network. In order to bring a reconfigurable scan network into a certain network configuration, vectors have to be shifted through the scan input port. Then, an update operation moves the vector from the shift flip-flops (CAP cells) to the update latches (UPD cells) of the configuration module. This operation changes the active path of the network. Since a reconfigurable scan network can have a hierarchical structure, the operation of making an instrument, placed deep into the network, part of the active path may require multiple configuration phases. As an example, a simple IEEE Std 1687 scanning network shown in Fig. 1.2 is used. The sample network is accessible via an IEEE Std 1149 TAP interface and consists of a ScanMux that selects two segments: the first with a single TDR and the other with two SIBs that control two TDRs. The TDR_0, TDR_1, TDR_2 are respectively 2, 8 and 8 cells long. Each reconfigurable module, i.e., SIBs

and the ScanMux, are associated with a configuration bit (cb_1, cb_2 and cb_3) and are highlighted in gray. The table 1.1 shows all the possible configurations of the network, which depend on the configuration of the SIBs and ScanMux. The table uses 'A' to indicate that the SIB is in the asserted position, 'D' to indicate de-asserted, 0 and 1 for the two possible configurations of the ScanMux. During system reset, a known configuration is selected that is determined by the status of the reconfigurable modules. In the example network, the reset configuration is 0, D, D.

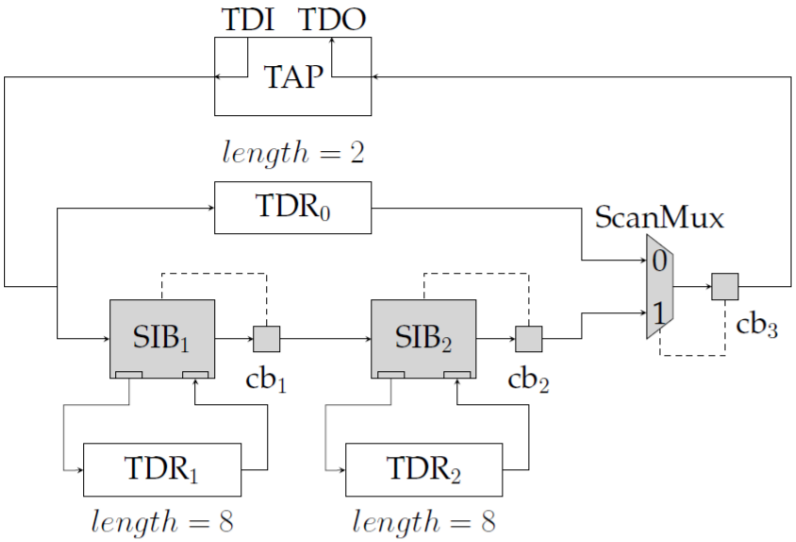

Figure 1.4: Example Reconfigurable Scan Network.

ScanMux	SIB_1	SIB_2	Path length	Active Path
0	D	D	3	$TDI \mapsto TDR_0 \mapsto cb_3 \mapsto TDO$
0	D	A	3	$TDI \mapsto TDR_0 \mapsto cb_3 \mapsto TDO$
0	A	D	3	$TDI \mapsto TDR_0 \mapsto cb_3 \mapsto TDO$
0	A	A	3	$TDI \mapsto TDR_0 \mapsto cb_3 \mapsto TDO$
1	D	D	3	$TDI \mapsto cb_1 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
1	D	A	11	$TDI \mapsto cb_1 \mapsto TDR_2 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
1	A	D	11	$TDI \mapsto TDR_1 \mapsto cb_1 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
1	A	A	19	$TDI \mapsto TDR_1 \mapsto cb_1 \mapsto TDR_2 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$

Table 1.1: Possible configurations for the network in the example.

1.3 Test of RSNs

A reconfigurable scan network consists of several reconfigurable modules that allow access to different parts of the network. The active path is decided by the value stored within each configurable module. In the example, one ScanMux and two SIBs are present in the network. The ScanMux select the path to activate between the path with TDR_0 and the path with SIB_1 and SIB_2. The SIB instead decides whether to include the respective TDRs in the main path. It is important to note that in the case where the path with the SIBs is not part of the active path, then it is not possible to configure the two SIBs with the respective cells. In this thesis, each element of the network is associate to the most specific segment possible. In the example, SIB_2 controls the TDR_2 segment and, the configuration bit of SIB_2, i.e. cb_2, is situated in the segment of SIB_2. The same applies to cb3 that configures the ScanMux, in fact the ScanMux controls the TDR_0 segment and the SIBs segment, instead cb_3 is in the upper segment, i.e. the one with the ScanMux. Each element has a depth, which indicates the hierarchical level of its position in the network. For example, the TDR_1 and TDR_2 segments have depth 3, TDR_3, SIB_1, SIB_1, cb_1, cb_2 have depth 2, instead cb_3 and ScanMux has depth 1 because it is placed at the top level. In general, the depth of a configuration bit is the depth of the configurable module decremented by one. The length of the active path is equal to the sum of the lengths of all the elements included in that path, counting TDR segments and configurable module controlling. A generic configuration of the network (i.e., the value of all configuration bits) is referred to as C_i. The reset configuration is indicated with the term C_0. Each configuration C_i can be associated to a record, which contains an identifier and the following information:

- the configuration bit values for each reconfigurable module M_i (e.g., asserted/de-asserted for SIBs, an input identifier for ScanMuxes);
- the active path length;
- the list of possible faults (each referred to as F_i) affecting the network, that can be detected by performing test operations while the network is configured with C_i.

The test operation checks whether the path inserted between the scan input and scan output pins is the same at the expected path using test vectors, i.e., whether the right instruments can be accessed during the normal operation. The test operations associated to a generic test vector tv_i correspond to:

1. a sequence of length equal to the active path is shifted in TDI, forcing it to follow the active path and to go out on the other end;
2. scan output pins (e.g., TDO) are monitored: possible faults can be detected observing sequence output that matches the expected one or not. Indeed, in the case a fault is

present, the expected output sequence will appear on scan output pins after a wrong number of clock cycles because the active path has changed due to a fault.

A change in the configuration of a network is defined as a network Transition through one or more configuration vectors. The operations associated to a generic configuration vector cv_i correspond to:

1. as many shift operations how many the active path length, to store the next configuration in the CAP flip-flops of the configuration bits of the reconfigurable modules.

2. an update operation to apply the next configuration to the network, so change the configuration of modules and the active path.

It is possible to pass from a configuration C_i to a configuration C_j directly or through others configuration vectors. When it requires a single configuration vector, then C_j is a neighbor configuration of C_i and the transition cost in terms of clock cycles is equal to the active path length of C_i increased by one for the update operation. it is important to note that if C_j is neighbor C_i then it is not true that C_i is neighbor C_j, i.e. the neighborhood relation is not reversible. For example, let us consider the network in Fig. 1.2, whose configurations are listed in Table 1.1. In this network, the configuration bits are placed in the same segment of the related reconfigurable module (i.e., right after each SIB and ScanMux), then the network can be moved from the configuration C_1 = 1AA to C_2 = 0DD by shifting a single vector. On the contrary, when the network is in C_2, two vectors are needed to reach C_1, passing through the intermediate configuration C_3 = 1DD. The neighborhood Zi of a certain configuration C_i is obtained by generating all permutations on the reconfigurable modules' configuration bits included in the active path, so they can be changed by shifting a single vector in the network with configuration C_i. In the previous example, the network is configured in C_1 and all configuration bits are part of the active path, thus all other configurations are part of the neighborhood of C_1. On the contrary, the configuration C_2 only shows the element ScanMux, while SIB_1 and SIB_2 are not present in the active path; thus, the neighborhood of C_2 is obtained by changing the configuration of ScanMux, i.e., it only includes C_3. A generic session, indicated with S_i, includes configuration and test vectors, it is composed of two phases [17]:

1. a configuration phase (Cfg), corresponding to a network transition, in which a certain number of configuration vectors are applied, until the target configuration is reached;

2. a test phase (Tst), in which test vectors are applied. There are different sequences of test vectors in the test phase and depends on the kind of defects to be tested.

Considering the generic session S_i, we denote by t_c^i the duration (in clock cycles) of the configuration phase Cfg_i and by t_t^i the duration of the test phase Tst_i. The configuration

time is the time needed to apply all the configuration vectors of the session. Each vector requires a certain time to be shifted in, plus a few clock cycles to update it into the UPD cells of the corresponding path, this time is denoted as JTAG protocol overhead [4] and it is implementation dependent. Each vector may have a different length because the active path changes after each update operation. The duration of the test phase (t_t^i) depends on the active path length l of the target configuration, i.e., the configuration after the last configuration vector. The total test duration of N sessions for a network is thus given by

$$T = T_c + T_t = \sum_{i=1}^{N} t_c^i + \sum_{i=1}^{N} t_t^i \tag{1.1}$$

where T_c is the sum of clock cycles of each Cnf_i and T_t is the sum of the clock cycles of each Tst_i.

1.4 Fault Model for Reconfigurable Modules

Since physical faults are often difficult to deal with, logical faults are often used, which are an abstract model. The way a logical fault models a physical defect is called fault model. The most commonly used fault model is the single-line stuck-at. Reconfigurable modules are used to include segments into the scan path or to exclude them from the scan path. In the event of faults in the reconfigurable modules, the network configuration may become different than expected or unknown, so the network becomes unusable. A functional fault model is used to make the faults independent of the implementation. Thus, a given fault in a reconfigurable module forces the network into a different configuration than expected one. A fault F_i selects an active path different from the expected one without fault, this path activated by the fault is called faulty path. For example, in Fig. 1.2 the SIB_1 can have a fault that does not allow access to the TDR_1 segment, so it can be blocked in the de-asserted configuration, regardless of the value of the configuration bit. The same can happen to the ScanMux which always selects the same segment regardless of the configuration. By testing these functional faults, Stuck-at faults in the shift flip-flops (CAP cells) of the configuration modules and the faults affecting the update logic of reconfigurable modules are implicitly considered as detected. Moreover, these faults also cover those errors on the reset logic with the effect of blocking the module at its reset value, but other faults such as the ineffective reset are not considered. A proper test for functional faults of a reconfigurable module is composed of the following operations:

1. Excite the fault to create a different scenario from the one without fault. Then, configure the network so that the active path includes the faulty element. In the case of reconfigurable modules, the excited faults change the length of the active path.

2. A proper sequence is shifted into the network, and the expected path length is compared against the length of the active path. In particular, the comparison is performed by counting at the number of clock cycles required by the input sequence to appear on the scan output pin.

For example, if the ScanMux of Fig 1.2 has a functional fault that always selects the segment at input 1, then it is possible to excite the fault with a configuration that forces input 0 of the ScanMux. Table 1.1 shows all these possible configurations and the length of the selected faulty paths. It is possible to notice that the first configuration of the table is not able to detect the fault, because the faulty path length is equal to the length of the active path. Therefore, you can select one of the remaining three configurations for the test. When all faulty paths have the same length as the active path, then there is no configuration that can test the fault, so the fault is untestable.

Afterwards, the basic concepts presented are applied in a test procedure for SIB and ScanMux modules. In the test procedure, configuration vectors and test vectors are used.

ScanMux	SIB_1	SIB_2	Path length Faulty	Path length Active	Faulty Path
0	D	D	3	3	$TDI \mapsto cb_1 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
0	D	A	11	3	$TDI \mapsto cb_1 \mapsto TDR_2 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
0	A	D	11	3	$TDI \mapsto TDR_1 \mapsto cb_1 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$
0	A	A	19	3	$TDI \mapsto TDR_1 \mapsto cb_1 \mapsto TDR_2 \mapsto cb_2 \mapsto cb_3 \mapsto TDO$

Table 1.2: Effect of the functional fault ScanMux always-selects-1, when selecting different active paths.

For the test phase we include an initialization vector that forces the network to a known value, because each test vector must determine if the active path is long as expected path. As initialization vector, a sequence of 0s equal to the longest path in the network can be used, because an fault involving a reconfigurable module determines an faulty path, whose length is generally different from the expected path length. Therefore, the length of the longest path in the network, or the maximum length between all the defective paths and the intended path, are suitable values to be used before any test phase. In case of the longest path, each t_t^i contribution in Eq. 1.3 includes this length. In the example of Fig. 1.2, an initialization vector composed of 19 0s can be used before each test phase.

1.4.1 SIBs

Given a SIB, the test procedure for testing both the SIB stuck at asserted (stuck-at-A) and de-asserted (stuck-at-D) faults is the following:

1. configure the network, so that the target SIB becomes part of the active path;
2. shift in an initialization vector whose length is equal to the one of the longest path in the network;
3. shift in a test vector as long as the expected path length;
4. check whether the expected sequence appears on the output of the path;
5. reconfigure the network, so that the SIB is part of the active path and at the opposite configuration;
6. shift in an initialization vector whose length is equal to the one of the longest path in the network;
7. shift in a test vector whose length is equal to the one of the expected path length;
8. check whether the expected sequence appears on the output of the path.

As an example, to test SIB_2 of network in Fig 1.2, a sequence of alternated 0s and 1s is used as test vectors, with two consecutive 1s as sequence terminator. Until the sequence terminator is shifted out, the output pin is monitored to calculate the active path length and to compare it against the expected one. Assuming as reset configuration, the de-assertion of the SIBs and the input 0 of ScanMux, the test of SIB_2 in details:

1. Reset – active path (AP): TDI-TDR_0-cb_3-TDO
2. Apply configuration vector 1:
   - Shift in 001 (length = 3)
   - Update – AP: TDI-cb_1-cb_2-cb_3-TDO
3. Apply initialization vector 1:
   - Shift in 0000000000000000000 (length = 19)
4. Apply test vector 1:
   - Shift in 010 (length = 3)
   - Shift in 11 (length = 2)

5. Check test vector 1, while applying configuration vector 2:
    - Shift in 011 (length = 3)
    - Update – AP: $TDI\text{-}cb_1\text{-}TDR_2\text{-}cb_2\text{-}cb_3\text{-}TDO$
6. Apply initialization vector 2:
    - Shift in 0000000000000000000 (length = 19)
7. Apply test vector 2:
    - Shift in 01010101010 (length = 11)
    - Shift in 11 (length = 2)
8. Check test vector 2, shifting 11 bits or at maximum 19 bits (the longest path) plus 2 bits (the length of the sequence terminator) to comes out the last sequence terminator from the output pin.

1.4.2 ScanMuxes

The test procedure to scan multiplexers is the same procedure as the SIBs. The basic idea is to configure the ScanMux in order to reach a certain configuration, thus activating a certain path. The faulty path of a SIB is always longer or shorter than the active path, instead this is not the case of faulty paths of ScanMuxes because the length of the faulty path may vary depending on the configuration of other modules in the active path. Moreover, the faulty paths can be more than one as in a 4-to-1 multiplexer. In order for the fault to be testable, the length of each faulty path has to be different than the active path. In details, the test procedure for a testable scan multiplexer fault is the following: 1apply a certain number of configuration vectors, until:

1. the multiplexer is part of the active path and set at a certain configuration, and
2. the other modules are configured such that the faulty paths have different length than the active path;
3. shift in an initialization vector as long as the longest path in the network;
4. shift in a test vector as long as the expected path length;
5. check whether the expected sequence appears on the output of the path;
6. repeat the previous steps for all the multiplexer's configurations.

From Fig 1.2, to test the fault stuck-at 0 of ScanMux, assuming as reset configuration, the de-assertion of the SIBs and the input 0 of ScanMux:

1. Reset – active path (AP): TDI-TDR_0-cb_3-TDO
2. Apply test vector 1:
   - Shift in 010 (length = 3)
   - Shift in 11 (length = 2)
3. Check test vector 1, while applying configuration vector 1:
   - Shift in 001 (length = 3)
   - Update – AP: TDI-cb_1-cb_2-cb_3-TDO
4. Apply configuration vector 2:
   - Shift in 011 (length = 3) (or 111, 101)
   - Update – AP: TDI-cb_1-TDR_2-cb_2-cb_3-TDO

5. Apply initialization vector 1:
    - Shift in 0000000000000000000 (length = 19)
6. Apply test vector 2:
    - Shift in 01010101010 (length = 11)
    - Shift in 11 (length = 2)
7. Check test vector 2, shifting 11 bits or at maximum 19 bits (the longest path) plus 2 bits (the length of the sequence terminator) to comes out the last sequence terminator from the output pin

1.5 Depth-First Solution

Testing a non-reconfigurable scan chain for permanent faults has been a widely studied subject for years. There are several techniques, for example shifting a sequence of 0s and 1s through the scan chain, such as the sequence "00110011" that applies all possible transitions in two cycles [4]. However, Reconfigurable Scan Networks are more complicated to test. In fact, it is necessary to test the TDR flip-flops, which must be tested to check if they can move the values correctly when included in the active path, and to test reconfigurable modules, such as IEEE Std 1687 SIBs and ScanMuxes, to check if they can move the network in all its possible configurations. A technique based on the depth-first search of a topological representation of a network has been studied [17]. This technique checks that the capability of a network to change its configuration is not corrupted by a fault. Therefore, this technique finds a sub-optimal solution in terms of test time that tests all the reconfigurable modules, such as SIBs and ScanMuxes, and all the configuration bits associated to these modules. The main motivation of this work is to find sub optimal solutions in terms of test time using evolutionary computation, starting also from sub optimal solutions obtained from the depth-first algorithm. The topology graph is a simplified representation of the scan network in a graph that provides a topological view. Each vertex represents an element of the scan network, such as TDRs, SIBs, ScanMuxes and configuration bits, while the edges indicates the connections between elements. The vertices also include elements associated to the input and output pins of the scan network, e.g., TDI and TDO. the hierarchical depth of each element is annotated in the corresponding vertex. The topology graph of the scan network of Fig.1.2 is shown in Fig.1.5. In the example, the reconfigurable modules' vertices are highlighted in grey. For each vertex, the depth d is also reported in Fig.1.5.

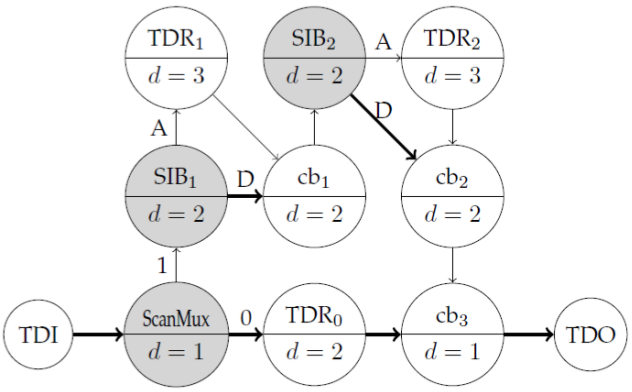

Figure 1.5: Topology graph of the example network.

To test a Reconfigurable Scan Network, it is necessary to reach a certain number of configurations, each one able to include in the active path a subset of the registers and the reconfigurable modules. After reaching the target configuration, the active path is tested, and the scan output values are monitored. Initially the network is set to its initial configuration, call reset configuration. The overall test procedure requires a certain amount of sessions. For each session, the target configuration of the previous session becomes the current configuration. During the test phase, the active path includes a certain number of reconfigurable modules to be tested. In this phase, an initialization vector composed of as many 0s as the longest path length is applied, followed by a test vector composed of an alternate sequence of 0s and 1s. The depth-first approach applies a sequence of test sessions by traversing the topology graph. At each step of the graph traversal, a subset of reconfigurable modules of the current active path is selected. In the set of selected modules, the modules, capable of forcing untested faults and located at maximum depth, are configured in the opposite way. All new configurations are applied together by means of a single configuration pattern or multiple configuration patterns. A test pattern is applied when the new configuration is reached, i.e., the configuration in which all excited faults become observable- Finally, the process is repeated until all faults are covered. As an example, the depth-first strategy on the graph in Fig. 1.5 produces the following test sequence after reset:

- Session 1
    1. Configuration 1,D,D
    2. Test: SIB_1 stuck-at-A, SIB_2 stuck-at-A
- Session 2
    1. Configuration 1,A,A
    2. Test: SIB_1 stuck-at-D, SIB_2 stuck-at-D, SMux stuck-at-0
- Session 3
    1. Configuration 0,A,A
    2. Test: SMux stuck-at-1

The work of this thesis uses the Depth-First algorithm to improve the performance of the evolutionary computation, since it allows to have a starting point of a sub-optimal solution with complete fault coverage.

1.6 Diagnostic analysis

The approach [7] described in this chapter for testing IEEE 1687 networks is based on the two-step iteration:

1. Network configuration to establish a certain path between the serial input of the network and the serial output
2. Shift a specific sequence within the network and observe the serial output.

The test session is defined as the combination of these two steps. The test sequence is defined as the sequence of test sessions to reach a given fault coverage. In an article [18] the extension of this approach to diagnosis is presented. The article describes a method to generate a sequence of stimuli that allows to identify the faulty element in a faulty network. This approach uses the fault model described above. The fundamental point is how to identify the sequence of sessions able to distinguish the largest possible number of faults pairs. The approach uses a database, called the fault dictionary [19], in which the behavior of the network is reported for every possible fault when a certain set of stimuli is applied, called the diagnostic sequence. The stimuli for the diagnosis can force the network to reach a different state for every possible fault in the ideal case. In case of a state incorrect, to identify the fault that caused the incorrect behavior it is necessary to search the state incorrect in the database. Furthermore, it is possible to identify those pairs of undistinguished faults by means of some diagnostic analysis of a test sequence. Given a configurable module in a network, and a test session in a test sequence, in which the module belongs to the session path and is configured in a given state, the faulty path is defined as the path that would be selected if the module would be in the opposite state with respect to the expected one [18]. Note that the length of each faulty path is affected by the configuration bits of the special modules, then by the network state in the previous session. The approach [18] is based on the identification of the distinguishable and not-distinguishable pairs of faults, calculating the behavior of the network for every possible fault. In this paper we only consider the faults of the reconfigurable modules. To determine which pairs of faults in SIBs and in ScanMuxes are distinguished by a given test session, it is important to note that there is at least one session for which the circuit fails, if there is a faulty reconfigurable module. If we include the faulted SIB in the active path, the configuration reached will be affected by the faulted SIB, so the active path will be different than expected. If the fault-free and faulty paths have different lengths, inserting a vector test in the network we can observe that the output will appear after a different number of clock cycles than expected. Depending on the difference in the cycles, the special faulty module can be identified. In practice, for each session it is necessary to determine the length of the faulty path, i.e. the path selected in case of a faulty reconfigurable module. For each session Si, the Session Fault Set (SFS_i) indicates the set of all the faults connected to the SIB and ScanMux of the active path and excited by the session.

The following properties [18] are described:

- given a session, all the faults of the (SFS_i) are distinguished from all the faults that do not belong to the session path;
- given a session, for each fault pair within the (SFS_i), the faults can be distinguished one from the other if the two faulty paths have different lengths.

Chapter 2

Evolutionary Algorithm

The approach proposed aimes at minimizing the test time while guaranteeing that all testable faults are covered. The approach is based on evolutionary computation. An Evolutionary algorithm is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An evolutionary algorithm uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators. Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. Techniques from evolutionary algorithms applied to the modeling of biological evolution are generally limited to explorations of microevolutionary processes and planning models based upon cellular processes. In most real applications of Evolutionary algorithms, computational complexity is a prohibiting factor. In fact, this computational complexity is due to fitness function evaluation. Fitness approximation is one of the solutions to overcome this difficulty. However, seemingly simple Evolutionary algorithm can solve often complex problems; therefore, there may be no direct link between algorithm complexity and problem complexity. In Evolutionary algorithm, a possible solution to the optimization task is called an individual and a set of individuals is called a population. Given an initial population, the individuals are recombined by the application of different operations and new individuals are generated. A subset of individuals is selected in this new set according to their fitness to determine the next population. This algorithm is divided into two parts: the *generation* of individuals and the *evaluation* of individuals.

The approach in the problem of Reconfigurable Scan Network test requires the implementation of the following features:

- A tool able to produce individuals are through mechanisms that ape both sexual and asexual reproduction. (GENERATION)
- A tool able to produce a list of configuration vectors that move the Reconfigurable Scan Network from the generic configuration source to the specific configuration destination and able to produce the list of faults that can be excited when the Reconfigurable Scan Network is moved to the generic configuration. (EVALUATION)
- A sub-optimal solution to the problem, i.e., a set of configurations able to excite all possible faults in the Reconfigurable Scan Network.

2.1 Generation tool

In this case, the individual is a sequence of test patterns composed by bit vectors. As tool for generation, the μGP3 (micro genetic programming) [20] is used. Given a task, it first fosters a set of random solutions, then iteratively refines and enhance them. Its heuristic algorithm uses the result of the evaluations, together with some internal structural information, to efficiently explore the search space, and eventually to produce the optimal solution. A population of different solutions is considered in each step of the search process, and new individuals are generated through mechanisms that ape both sexual and asexual reproduction. In case of Reconfigurable Scan Network, the individuals are sequences of test vectors, so patterns of bit. The evolutionary computation uses two main forces: Recombination and mutation create the necessary diversity and thereby facilitate novelty, while selection acts as a force increasing quality. The Generation tool use different operation, like one Point Crossover, two Point Crossover, single Parameter Alteration Mutation, insertion Mutation, removal Mutation, replacement Mutation, alteration Mutation, subgraph Removal Mutation, subgraph Replacement Mutation, random Walk Mutation.

- Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Crossover is a process of taking more than one parent solution and producing a child solution from them. In a one-point crossover operation of patterns bit, a single crossover point on both parents' organism bits is selected. All data beyond that point in either organism is swapped between the two parent organisms. The resulting organisms are the children. Two-point crossover calls for two points to be selected on the parent organism bits. Everything between the two points is swapped between the parent organisms, rendering two child organisms.

- Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. The purpose of mutation is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. The table shows examples of crossover and mutation operations on bit patterns.

The table 2.1 shows an example of one-point, two-point crossover and mutation of bit patterns.

	OnePointCrossover	**TwoPointCrossover**	**Mutation**
Parents	00011|1111000 01010|0101010	000|11111|1000 010|10010|1010	0001111|1|1000
Children	00011|0101010 01010|1111000	000|10010|1000 010|11111|1010	0001111|0|1000

Table 2.1: Example evolutionary operations.

2.2 Evaluation tool

In Evolutionary algorithm, a fitness function is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims. For configurable scan networks, the individuals are sequences of test vectors which may have adjacent configurations of network or may need multiple configurations of network between each pair of test vectors. So, fitness functions must be able to produce a list of configuration vectors that move the Reconfigurable Scan Network from the generic configuration source to the specific configuration destination and must be able to produce an evaluation for a given individual. The fitness is composed by two parameters, the first is the fault coverage and the second is the reciprocal of the total cost in terms of clock cycles of the test time and configuration time. Moreover, given a sequence of test patterns the Evaluation tool determines the configuration patterns of minimum cost in terms of clock cycles between each pair of test patterns, i.e. creates the sequence of configuration and test patterns from the initial individual, and computes the fault coverage and the total cost, i.e. the fitness.

Let's consider a simple Reconfigurable Scan Network example, which includes five instruments accessible through the TAP port, reading or writing from/to the relevant Test Data Registers (from TDR_1 to TDR_5). The network includes three SIBs and a ScanMux, as shown in Fig. 2.1; each of these modules can be configured to allow access to a given subset of TDRs.

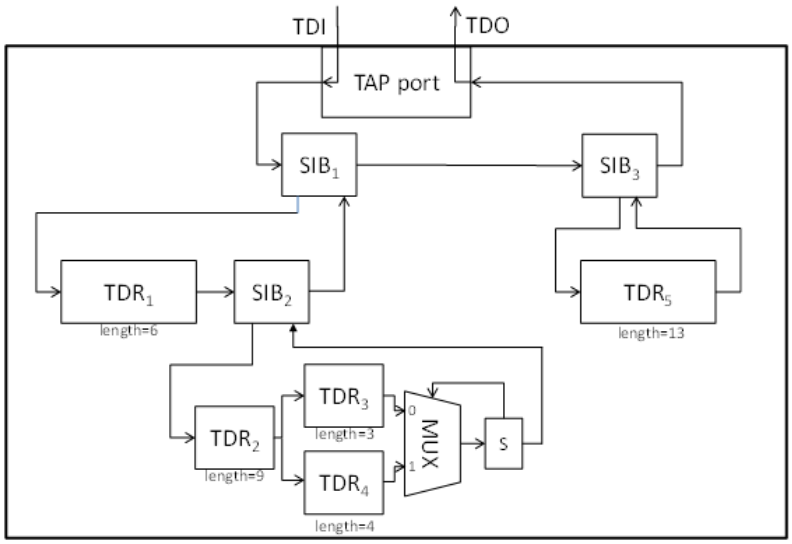

Figure 2.1: Example of IEEE 1687 Reconfigurable Scan Network.

All the possible configurations of this network are shown in the table 2.2, they depend on how the SIB and ScanMux have been configured. The sequence of bits to configure the network is called the configuration vector and is referred to as cv_i. It is possible to access a subset of TDRs after reaching a certain configuration. The subset of accessible TDRs is the active path.

Configuration	**SIB_1**	**SIB_2**	**SIB_3**	**ScanMux**	**Length**	**Active Path**
C_0	D	D	D	0	2	-
C_1	D	D	D	1	2	-
C_4	D	A	D	0	2	-
C_5	D	A	D	1	2	-
C_2	D	D	A	0	15	TDR_5
C_3	D	D	A	1	15	TDR_5
C_6	D	A	A	0	15	TDR_5
C_7	D	A	A	1	15	TDR_5
C_8	A	D	D	0	9	TDR_1
C_9	A	D	D	1	9	TDR_1
C_{10}	A	D	A	0	22	$TDR_1 \mapsto TDR_5$
C_{11}	A	D	A	1	22	$TDR_1 \mapsto TDR_5$
C_{12}	A	A	D	0	22	$TDR_1 \mapsto TDR_2 \mapsto TDR_3$
C_{13}	A	A	D	1	23	$TDR_1 \mapsto TDR_2 \mapsto TDR_4$
C_{14}	A	A	A	0	35	$TDR_1 \mapsto TDR_2 \mapsto TDR_3 \mapsto TDR_5$
C_{15}	A	A	A	1	36	$TDR_1 \mapsto TDR_2 \mapsto TDR_4 \mapsto TDR_5$

Table 2.2: Possible configurations for the network of Fig. 2.1.

Then, a list of configurations is generated by the evolutionary engine. The test session is composed by applying a transition function to generate intermediate configuration patterns for each configuration in the list, i.e. move the network from a configuration C_i in the list to C_{i+1}. Initially, the Transition function is applied between the reset configuration C_0 and the first configuration in the list if it is not equal to C_0. After each transition to a configuration in the list, a test pattern is applied, and the fault coverage is updated by calculating the faults achieved with that transition. Furthermore, the total test time is obtained from the sum of the configuration phase time to apply the generated configuration vectors plus the test phase time for all test vectors. Considering the network in the figure with reset configuration C_0, a possible solution consists in applying a test vector in each of the following configurations: C_0,C_{10},C_{12},C_{13}.

Each test vector that is shifter in the network is made as follows:

- as many 0s as the longest path length, i.e., 36 bits in the example network;
- an alternated sequence 0101..., as long as the length of the active path currently selected;
- two consecutive 1s (or two consecutive 0s) as the sequence terminator;
- only for the last test vector, a sequence as long as the length of the active currently selected (values being shifted in are not important).

Configuration	**Faults excited**
C_0	SIB_1-*A* SIB_3-*A*
C_1	SIB_1-*A* SIB_3-*A*
C_4	SIB_1-*A* SIB_3-*A*
C_5	SIB_1-*A* SIB_3-*A*
C_2	SIB_1-*A* SIB_3-*D*
C_3	SIB_1-*A* SIB_3-*D*
C_6	SIB_1-*A* SIB_3-*D*
C_7	SIB_1-*A* SIB_3-*D*
C_8	SIB_1-*D* SIB_2-*A* SIB_3-*A*
C_9	SIB_1-*D* SIB_2-*A* SIB_3-*A*
C_{10}	SIB_1-*D* SIB_2-*A* SIB_3-*D*
C_{11}	SIB_1-*D* SIB_2-*A* SIB_3-*D*
C_{12}	SIB_1-*D* SIB_2-*D* SIB_3-*A* *ScanMux*-1
C_{13}	SIB_1-*D* SIB_2-*D* SIB_3-*A* *ScanMux*-0
C_{14}	SIB_1-*D* SIB_2-*D* SIB_3-*D* *ScanMux*-1
C_{15}	SIB_1-*D* SIB_2-*D* SIB_3-*D* *ScanMux*-0

Table 2.3: List of faults excited by each configuration for the network of Fig. 2.1.

All network faults can be excited by these configurations, as indicated in the list of faults excited by each configuration (Table 2.3).

The vectors corresponding to this list of configurations is composed as follows:

1. tv_1 in C_0 (shift of 36+2+2 bits)
2. cv_1 from C_0 to C_{10} (shift of 2 bits, then update)
3. tv_2 in C_{10} (shift of 36+22+2 bits)
4. cv_2 from C_{10} to C_{12} (shift of 22 bits, then update)
5. tv_3 in C_{12} (shift of 36+22+2 bits)
6. cv_3 from C_{12} to C_{13} (shift of 22 bits, then update)
7. tv_4 in C_{13} (shift of 36+23+2+23 bits).

If to move the TAP controller moves from shift to update and vice-versa has a cost of 5 clock cycles, the test session described is performed in 325 clock cycles. Note that the order of the configurations in the list is important because the transition from one configuration to another generates different vectors. Considering the previous configurations in a different order: C_0,C_{10},C_{13},C_{12}. In this case, the list of vectors composing the test session is the following:

1. tv_1 in C_0 (shift of 36+2+2 bits)
2. cv_1 from C_0 to C_{10} (shift of 2 bits, then update)
3. tv_2 in C_{10} (shift of 36+22+2 bits)
4. cv_2 from C_{10} to C_{12} (shift of 22 bits, then update)
5. cv_3 from C_{12} to C_{13} (shift of 22 bits, then update)
6. tv_3 in C_{13} (shift of 36+23+2 bits).
7. cv_4 from C_{13} to C_{12} (shift of 23 bits, then update)
8. tv_4 in C_{12} (shift of 36+22+2+22 bits)

This test session has the fault coverage as the previous session, but with a longer duration (357 clock cycles). Furthermore, the C_{12} configuration is visited twice before applying a test vector (tv_4).

The table 2.4 shows an example of the Evaluation tool applied to a simple network, in which the first column represents an individual composed by tree test vectors and the second column represents the output of tool, i.e. the sequence of test vectors and configuration vectors with fault coverage and total cost.

Individual	**Transition**	**Evaluation**
000000000000000 [T]	000000000000000 [T] (102)	Configuration patterns: 4
111100001110000 [T]	111100001110000 [U] (165)	Test patterns: 3
111111111111111 [T]	111100001110000 [T] (165)	Configuration cost: 747
	111110001110000 [U] (179)	Test cost: 2068
	111111001111000 [U] (281)	Total cost: 2815
	111111111111111 [U] (332)	Fault coverage: 0.83870965
	111111111111111 [T] (332)	Total JTAG overhead: 35

Table 2.4: Example Evaluation function.

2.3 Optimization Techniques

An important factor that influences performance is the initial population. It is possible to use various methods for the starting population for example to generate random individuals. However, starting from individuals with complete fault coverage improves performance. A technique can be to use patterns resulting from sub-optimal algorithms such as a depth-first approach or a breadth-first approach. In this paper, we use three methods to characterize the initial population:

- individuals include random configurations;
- individuals may include one or more configurations generated by the depth-first approach;
- the initial population includes an individual with the suboptimal solution of depth-first approach.

Chapter 3

Diagnosis

The approach used to diagnose the reconfigurable modules of Reconfigurable Scan Networks is based on transforming a the test sequence into a diagnostic sequence. The test sequence has the task of exercising all possible faults, instead the diagnostic sequence has the task of identifying the fault that causes the manifested failure. In this diagnostic work only the faults of the reconfigurable modules such as SIBs and ScanMuxes are considered. The following sections describe the diagnostic analysis, to identify undistinguished faults, and the diagnostic sequence generation approach.

3.1 Diagnostic analysis

As written above, to determine which pairs of faults in SIBs and in ScanMuxes are distinguished by a given test session, it is important to note that there is at least one session for which the circuit fails, if there is a faulty reconfigurable module. If we include the faulted SIB in the active path, the configuration reached will be affected by the faulted SIB, so the active path will be different than expected. If the fault-free and faulty paths have different lengths, inserting a vector test in the network we can observe that the output will appear after a different number of clock cycles than expected. Depending on the difference in the cycles, the special faulty module can be identified. In practice, for each session it is necessary to determine the length of the faulty path, i.e. the path selected in case of a faulty reconfigurable module. The approach used to generate the diagnostic sequence is based on the propagation of the fault detected in the test sequence from a given configuration up to the test pattern for each session, so the objective of the diagnostic analysis is to calculate for each configuration pattern which faults are distinguished, and which are not. So, the idea of the diagnostic analysis applied to each session is extended to each vector configuration. Given a session composed of N configuration vector and a vector test, each applied network configuration can achieve different faulty configurations, based on possible faults.

Thus, in the approach described in the next section, diagnostic analysis is applied for each configuration pattern.

3.2 Diagnostic Sequence

The method for the generation of diagnostic sequence from a test sequence is based on the propagation of possible fault identified by a given network configuration in the test sequence until the test pattern for each session. Therefore, undistinguished faults may be present at each configuration step. The main idea to deal with undistinguished faults is to perform an exhaustive search of the configurations space in order to reach distinguishable configurations in the presence of such faults. Therefore, from a certain configuration it is possible to reach faulty configurations due to faults, if these configurations are not distinguishable, i.e. they have equal path length, then it is possible to apply a method that allows forcing the network from these faulty configurations indistinguishable from distinguishable faulty configurations, when possible. Given a sequence of 3 configurations C_1, C_2, C_3:

1. calculate all the possible faulty configurations (C_2F_i) by the transition of the network from C_1 to C_2.
2. calculate all the possible faulty configurations (C_3F_i) by the transition of the network from C_2 to C_3.
3. propagate the faults in the faulty configurations (C_2F_i) to C_3:
   (a) for each faulty configuration C_2F_i, determine the configuration reached $C_3F_i^p$ in the forced transition a C_3.
   (b) Apply the diagnostic analysis among $\{C_3F_i^p\} \cup \{C_3F_i\}$.
4. if there are undistinguishable faults U by the diagnostic analysis, then find a diagnostic pattern that allows to propagate the undistinguishable faults from C_2F_i in distinguishable faulty configurations $C_3F_i^p(D)$:
   (a) create diagnostic pattern of maximum length among the lengths in $\{C_3F_i^p\} \cup \{C_3\}$, composed of fixed bits (C_3; $C_3F_i^p(D)$) and mobile bits ($C_3F_i^p(U)$)
   (b) perform an exhaustive search of the diagnostic pattern that allows to propagate the undistinguishable faults from C_2F_i in distinguishable faulty configurations $C_3F_i^p(D)^*$

At the end of a session, it is possible that indistinguishable faults remained because the procedure described above did not find any solution. In this case, then it is possible to extend the propagation of these faults to the next session, only if the faults are distinguishable

from the fault-free path. Therefore, at the end of the session, if there are undistinguishable faults among them but distinguishable from the fault-free path then the faults are further propagated to the next session. In the next session, it is not necessary to apply a diagnostic analysis among the possible faults of the session and those propagated in the session because the latter showed an undistinguishable failure in the test of the previous session, i.e. a length different from the fault-free path and the distinguishable faulty-path. However, it is important to apply diagnostic analysis between the faults propagated in sessions. Given the sequence of configuration and test vectors C_1, C_2, C_3, T_3, C_4, C_5, T_5, such as in the Fig. 3.2 :

- apply the diagnostic pattern generation method described above to the first session (C_1-T_3).
- output distinguishable fault (green nodes).
- if there are undistinguishable faults (blue nodes) between $\{C_3F_i^p\} \cup \{C_3F_i\}$, but distinguishable from the fault-free configuration, then propagate these faulty configurations to the next session (C_4-T_5).
- apply the diagnostic pattern generation method, considering the faults propagated from the previous session as a separate set from the new faulty configurations calculated and propagated.

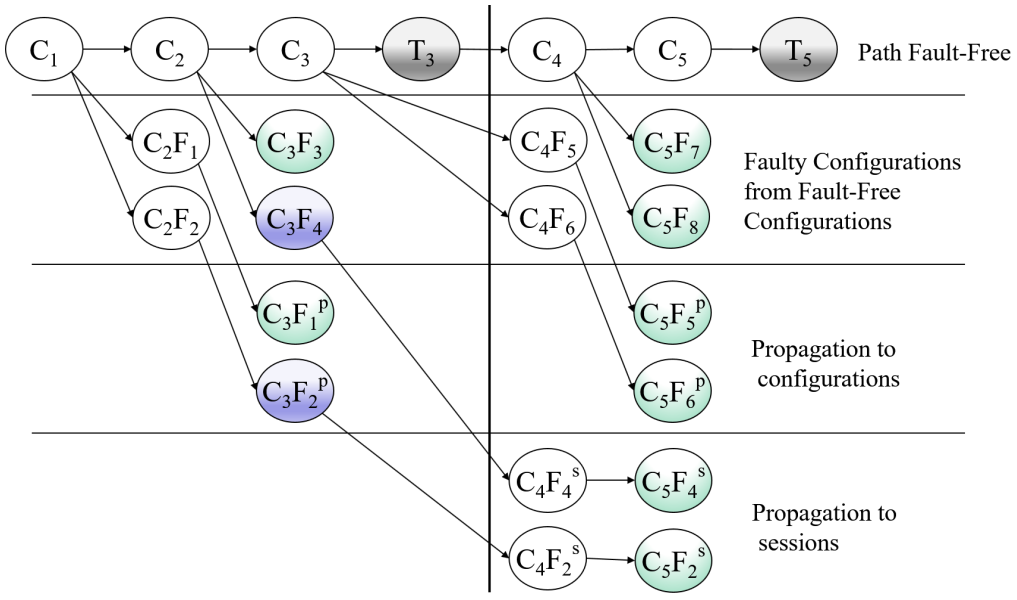

Figure 3.1: Example diagnostic method.

Chapter 4

Experimental Results

The effectiveness in terms of test duration of the proposed algorithms has been evaluated with an in-house tool on the ITC'16 benchmarks of IEEE Std 1687 Reconfigurable Scan Networks [9]. Moreover, additional networks have been used for the test. The Evolutionary algorithm has been compared against the depth-first search algorithms [7]. For experiments, a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM was used. Each benchmark network has been tested using Evolutionary Algorithm and sub-optimal approaches, depth-first. The experiments have been parallelized using up to 8 cores of the server. The number of random individual has been setup to 200, the number of genetic operation has been setup to 120, and the steady state equal to 500 generations. The activated genetic operations were: insertion, replacement, removal, alteration, swap, one-point precise/imprecise crossover, two-point precise/imprecise crossover and invert-over [21]. The test was performed with different approaches to improve performance in terms of time and results. Three approaches have been used for initial population:

- individuals of initial population generate random;
- individuals of initial population generate with chromosomes of the sub-optimal solution of the depth-first algorithm;
- individuals of initial population generate random, but with an individual equal to the sub-optimal solution of the depth-first algorithm.

The last two approaches improve performance because they allow first to reach a generation with at least one individual with complete fault coverage, as in the second case, or allow to have at least one individual with complete fault coverage in the first generation, as in the third approach, compared to the random case.

4.1 Framework

The implemented framework connects the evolutionary engine μGP [20] and a prototype evaluator. The μGP has the task to generate and to combine the individuals. The Evaluation tool, written in Java, includes the ICL Tools Software library and it is able to read the network described in ICL format and others formats such as xml. The tool represents different configurations in states to be tested or to be reached, each with its own active path length. Furthermore, it is able to perform the transition function, i.e., to calculate the configuration path from a source configuration to a destination configuration with the lowest cost. It computes the fault coverage and the total test cost in terms of clock cycles. For diagnostics, the evaluator is able to apply a diagnosis analysis to identify the faulty distinguishable configurations reached by a given configuration and to force the network, in transition, into a distinguishable configuration in the case of fault. The engine μGP and the evaluator communicate through a script in the Linux environment. Specifically, the evolutionary engine generates the initial population based on a specific model and the Evaluator reads each individual of the population, evaluates it and generates the fitness, i.e., the fault coverage and the total test cost. Finally, the evolutionary engine creates a new generation through the combinations (mutation, crossover...) of individuals with the best ratings and the process is repeated.

4.2 ITC'16 Benchmark networks

The key characteristics of the ITC'16 benchmark networks are detailed in Table 4.1. For each network, the table reports first the number of SIBs and ScanMuxes. The fourth column refers to the number of configuration bits of SIBs and ScanMuxes. The column Max depth indicates the maximum hierarchical depth of each network (for SIB-based networks this value equals to the maximum number of nested SIBs). Finally, the column Longest path reports the maximum possible number of scan cells on active path, while Total scan cells is the sum of the lengths of all scan registers in each network. In the experiments, the cost for a configuration pattern has been set to the active path length plus the JTAG protocol overhead (to move from shift to update). The cost for a test pattern has been set to the sum of the following contributions:

- the JTAG protocol overhead (to move from update to shift), which has been set to 5;
- the longest path length (initialization vector);
- the active path length plus two (a sequence of alternated 0s and 1s as long as the active path followed by two consecutive 1s).

Network	SIB	ScanMux	Config. bits	Max depth	Longest path	Total scan cells
Mingle	10	3	13	4	171	270
TreeBalanced	43	3	48	7	5,219	5,581
TreeFlatEX	57	3	62	5	5,100	5,195
TreeUnbalanced	28	-	28	11	42,630	42,630
a586710	-	32	32	4	42,381	42,410
p22810	270	-	270	2	30,356	30,356
p34392	-	96	96	4	27,899	27,990
p93791	-	596	596	4	100,709	101,291
q12710	27	-	27	2	26,185	26,185
t512505	159	-	159	2	77,005	77,005
N17D3	7	8	15	4	372	462
N32D6	13	10	23	4	84,039	96,158
N73D14	29	17	46	12	190,526	218,869
N132D4	39	40	79	5	2,555	2,991
NE600P150	207	194	401	78	23,423	28,250
NE1200P430	381	430	811	127	88,471	108,148

Table 4.1: Characteristics of the ITC'16 benchmark networks

Experimental results on ITC'16 benchmarks are shown in Table 4.2. For Evolutionary algorithm, the column 2 reports the method of generating the initial population:

A. individuals include random configurations;

B. individuals may include one or more configurations generated by the depth-first approach;

C. the initial population includes an individual with the suboptimal solution of depth-first approach.

Moreover, the table shows the total number of configuration patterns (column 3) and the total number of test patterns (column 4). The table also indicates the number of clock cycles required (column 5) by configuration patterns and test patterns. Finally, the time execution (column 6), the total number of individuals (column 7) and the total number of eras (column 8) are reported. All modeled faults have been covered in each experiment (i.e., test coverage is 100%).

Network	**M**	**Config patterns**	**Test patterns**	**Total time [cc]**	**Exe time [h]**	**#inds**	**#eras**
Mingle	B	6	7	2,078	8	24293	711
TreeBalanced	C	7	8	69,369	9	43,914	500
TreeFlatEX	C	16	6	55,776	8	51793	638
TreeUnbalanced	B	17	12	1,042,450	5	31,329	476
a586710	B	5	5	298,241	8	31,996	624
p22810	C	2	3	152,937	9	21,958	500
p34392	B	5	5	196,505	7	45,623	565
p93791	C	4	5	708,878	23	29,932	500
q12710	C	2	3	131,022	5	20,199	500
t512505	C	2	3	386,024	8	21,279	500
N17D3	A	4	5	3,851	5	164,972	1738
N32D6	A	6	5	893,017	5	79,229	965
N73D14	B	13	13	5,967,137	3	54,183	964
N132D4	B	5	6	37,257	3	61,764	869
NE600P150	C	78	79	3,726,726	12	45,842	500
NE1200P430	C	127	128	21,515,705	50	48,920	500

Table 4.2: Experimental results

The table 4.3 shows the results compared with the First-Depth approach for benchmark networks. In particular, for each algorithm the number of pattern configurations and the number of test pattern are indicated in column 3 and 4. The configuration time and test time are reported with the total time in column 5, 6 and 7. Finally, the last column shows the rate between the two algorithms. The evolutionary approach was able to reduce the test time in 9 out of 16 RSN. However, the test time of the other networks has not been improved. In the case of large networks such as NE1200P430, an improvement has not been achieved, because the search space has not been correctly explored with the evolutionary parameters used due to the size. In the remaining non-improved cases, probably the depth-first approach has reached an optimal global solution because the Reconfigurable Scan Network has a simple structure in terms of hierarchical depth as for p22810, q12710, and t512505 networks with depth 2, hence, there is no further space for improving the result. In the improved cases, for method A a reduction of about 7% of the test time was noted, for method B up to about 9% and for method C up to about 22%. These results show that by gradually adding some knowledge of the problem to the evolution, the final solutions can be improved. In the following subsections some experimental results are reported.

Network	Algorithm	Configuration patterns	Test patterns	Configuration time [cc]	Test time [cc]	Total time [cc]	Ratio
Mingle	Depth-first	6	7	362	1,920	2,281	-
	Evolutionary	6	7	362	1,716	2,078	0,911
TreeBalanced	Depth-first	7	8	8,580	60,789	69,369	-
	Evolutionary	7	8	8,580	60,789	69,369	1
TreeFlatEX	Depth-first	5	6	15,263	56,078	71,341	-
	Evolutionary	16	6	11,648	44,123	55,776	0,782
TreeUnbalanced	Depth-first	11	12	237,475	834,324	1,071,799	-
	Evolutionary	17	12	274,816	767,634	1,042,450	0,973
a586710	Depth-first	4	5	1,471	298,153	299,624	-
	Evolutionary	5	5	43,150	255,091	298,241	0,995
p228120	Depth-first	2	3	573	152,364	152,937	-
	Evolutionary	2	3	573	152,364	152,937	1
p34392	Depth-first	4	5	697	196,005	196,702	-
	Evolutionary	5	5	28,314	168,191	196,505	0.999
p93791	Depth-first	4	5	1,950	706,928	708,878	-
	Evolutionary	4	5	1,950	706,928	708,878	1
q12710	Depth-first	2	3	43	130,979	131,022	-
	Evolutionary	2	3	43	130,979	131,022	1
t512505	Depth-first	2	3	494	385,530	131,022	-
	Evolutionary	2	3	494	385,530	131,022	1
N17D3	Depth-first	4	5	802	3,341	4,143	-
	Evolutionary	4	5	656	3,195	3,851	0,930
N32D6	Depth-first	4	5	183,439	759,031	942,470	-
	Evolutionary	6	5	235,161	657,856	893,017	0,948
N73D14	Depth-first	12	13	1,577,674	4,400,373	5,978,047	-
	Evolutionary	13	13	1,676,258	4,290,879	5,967,137	0,998
N132D4	Depth-first	5	6	9,332	29,399	38,731	-
	Evolutionary	5	6	9,332	27,925	37,257	0,962
NE600P150	Depth-first	78	79	916,829	2,809,89	3,726,726	-
	Evolutionary	78	79	916,829	2,809,89	3,726,726	1
NE1200P430	Depth-first	127	128	5,014,931	16,500,774	21,515,705	-
	Evolutionary	127	128	5,014,931	16,500,774	21,515,705	1

Table 4.3: Comparison results first-depth approach and Evolutionary algorithm

4.2.1 N17D3

In this subsection, the application of the evolutionary algorithm to the N17D3 network is reported. The network has 7 SIBs, 8 ScanMuxes with maximum depth 4. In particular, the table 4.4 shows the patterns found by the first-depth approach, together with the test data using these patterns.

Individual	**Evaluation**
000000000000000 [T] (102)	Configuration patterns: 4
111100001110000 [U] (165)	Test patterns: 5
111100001110000 [T] (165)	Configuration cost: 802
111110001111001 [U] (191)	Test cost: 3341
111110001111001 [T] (191)	Total cost: 4143
111111001111111 [U] (324)	Fault coverage: 1.0
111111001111111 [T] (324)	Total JTAG overhead: 45
111111111111111 [U] (332)	
111111111111111 [T] (332)	

Table 4.4: Result N17D4 First-Depth approach.

The results of the evolutionary algorithm are shown in the table 4.5, with the relative configuration and test patterns in the first column, and the cost data in the second column. Evolution is initiated by completely random patterns, therefore without optimization methods.

Best Individual	**Evaluation**
000000000000000 [T] (102)	Configuration patterns: 4
000100000110000 [U] (150)	Test patterns: 5
000100000110000 [T] (150)	Configuration cost: 656
110110001101001 [U] (135)	Test cost: 3195
110110001101001 [T] (135)	Total cost: 3851
111111000111001 [U] (249)	Fault coverage: 1.0
111111000111001 [T] (249)	Total JTAG overhead: 45
111111111111111 [U] (332)	
111111111111111 [T] (332)	

Table 4.5: Result N17D4 Evolutionary algorithm.

In addition, network diagnostics is shown in the table 4.6. For each session, the possible fault states and related faults are shown. Note that the faults with *, are faults that occurred in the previous session, but since they are undistinguishable, they have been propagated to the next session.

Session	**Fault States**	**Faults**
000000000000000 [T] (102)	000000000010000 (153)	4Mux always-selects-1
	000000001000000 (117)	sMux6 always-selects-1
	100000000000000 (103)	sMux16 always-selects-1
	001000000000000 (95)	sMux14 always-selects-1
	000000000100000 (93)	sMux5 always-selects-1
000100000110000 [T] (150)	000100000111000 (179)	3Mux always-selects-1
	000110000110000 (164)	12Mux always-selects-1
	000100000010000 (159)	sMux5 always-selects-0
	010100000110000 (156)	sMux15 always-selects-1*
	000000000110000 (144)	13Mux always-selects-0
	001100000110000 (143)	13Mux always-selects-1*
	000100000110001 (133)	sMux22 always-selects-1
	000100000100000 (99)	4Mux always-selects-0
110110001101001 [T] (135)	110111001101001 (208)	11Mux always-selects-1
	010110001101001 (134)	sMux16 always-selects-0
	100110001101001 (129)	sMux15 always-selects-0
	110100001101001 (121)	12Mux always-selects-0
	110110000101001 (120)	sMux6 always-selects-0
111111000111001 [T] (249)	111111000111011 (305)	1Mux always-selects-1
	111111000111000 (266)	sMux22 always-selects-0
	111111100111001 (264)	sMux10 always-selects-1
	110111000111001 (256)	sMux14 always-selects-0
	111111000111101 (253)	2Mux always-selects-1
	111111010111001 (242)	sMux9 always-selects-1
	111111000110001 (220)	3Mux always-selects-0
	111110000111001 (176)	11Mux always-selects-0
111111111111111 [T] (332)	111111101111111 (339)	sMux9 always-selects-0
	111111111111011 (328)	2Mux always-selects-0
	111111011111111 (317)	sMux10 always-selects-0
	111111111111101 (276)	1Mux always-selects-0

Table 4.6: N17D4 Diagnostic.

4.2.2 N32D6

The tables 4.7 and 4.8 show the results of the experiment on the network N32D6.

Individual	**Evaluation**
00000000000000000000 [T] (2)	Configuration patterns: 4
11100000110000011111000 [U] (6)	Test patterns: 5
11100000110000011111000 [T] (6)	Configuration cost: 183439
11110000111000011111111 [U] (70)	Test cost: 759031
11110000111000011111111 [T] (70)	Total cost: 942470
11111101111100011111111 [U] (73)	Fault coverage: 1.0
11111101111100011111111 [T] (73)	Total JTAG overhead: 45
11111111111111111111111 [U] (42)	
11111111111111111111111 [T] (42)	

Table 4.7: Result N32D6 First-Depth approach.

Best Individual	**Evaluation**
00000000000000000000000 [T] (2)	Configuration patterns: 6
10000000110000010000000 [U] (6)	Test patterns: 5
10100000111000010000000 [U] (6)	Configuration cost: 235161
00110000011100000001000 [U] (70)	Test cost: 657856
00110000011100000001000 [T] (70)	Total cost: 893017
11110100111011110111000 [U] (73)	Fault coverage: 1.0
11110100111011110111000 [T] (73)	Total JTAG overhead: 55
11111111111111111111111 [U] (42)	
11111111111111111111111 [T] (42)	
10101000110001000000110 [U] (139)	
10101000110001000000110 [T] (139)	

Table 4.8: Result N32D6 Evolutionary algorithm.

4.2.3 Mingle

The tables 4.9 and 4.10 show the results of the experiment on the network Mingle.

Individual	**Evaluation**
0000000000000 [T] (2)	Configuration patterns: 6
1000001000000 [U] (6)	Test patterns: 7
1000001000000 [T] (6)	Configuration cost: 362
1010001010000 [U] (70)	Test cost: 1920
1010001010000 [T] (70)	Total cost: 2282
1011001110000 [U] (73)	Fault coverage: 1.0
1011001110000 [T] (73)	Total JTAG overhead: 65
1111001111001 [U] (42)	
1111001111001 [T] (42)	
1111101111111 [U] (139)	
1111101111111 [T] (139)	
1111111111111 [U] (171)	
1111111111111 [T] (171)	

Table 4.9: Result Mingle First-Depth approach.

Best Individual	**Evaluation**
0000000000000 [T] (2)	Configuration patterns: 6
1000001000000 [U] (6)	Test patterns: 7
1000001000000 [T] (6)	Configuration cost: 362
1010001010000 [U] (70)	Test cost: 1716
1010001010000 [T] (70)	Total cost: 2078
1011001110000 [U] (73)	Fault coverage: 1.0
1011001110000 [T] (73)	Total JTAG overhead: 65
1111001111001 [U] (42)	
1111001111001 [T] (42)	
1111101111111 [U] (139)	
1111101111111 [T] (139)	
1111110110010 [U] (69)	
1111110110010 [T] (69)	

Table 4.10: Result Mingle Evolutionary algorithm.

Conclusions

The modern Integrated Circuits include within them various support functions for testing, such as Built-In Self-Test modules, and sensors capable of measuring parameters such as current, temperature and delays. These features allow to monitor the operation of the integrated circuit and the registers to set up and calibrate the operation of specific modules. To simplify access to all these resources called instruments, the IEEE 1687 standard [1] was introduced. This last standard is the evolution of the IEEE 1149.1 standard, in fact it is based on the scan chains accessible through the Test Access Port (TAP) but, unlike the previous version, these chains can be divided and configured in the most appropriate way. This new approach allows a flexible choice of the best trade-off between different parameters, such as access time or area. This new standard [2] also describes the ways to design configurable networks within the circuit, so in general, we can refer to Reconfigurable Scan Networks. As a result, the new generation of devices will include Reconfigurable Scan Networks accessible via the TAP interface and will support serial access to internal instruments. The problem of testing the scan chains introduced by the IEEE 1149.1 standard have been addressed by various studies [4] [5] [6]. With the new standard, one must face the problem of testing the hardware of the Reconfigurable Scan Networks, verifying any defects. The complexity of testing an Reconfigurable Scan Network is higher than the simple scan network, because it is necessary to verify if the network can be configured correctly and if it works as expected, i.e. the expected sub-network is made accessible. Specifically, the test must check whether each special module in the network is working properly. The global cost for testing a device is influenced by the test time of the IEEE 1687 networks, so it is very important to minimize the test duration of these networks maintaining the same Fault Coverage. This thesis proposes an alternative method that allows to achieve this goal using evolutionary calculation. A method capable of handling even large and complex Reconfigurable Scan Networks and that can produce a sequence of tests to detect any permanent faults on the reconfigurable modules. Experimental results on the standard benchmark suite produce an optimized test sequence in 9 of 16 cases, showing in some cases a significant reduction in test time. Future developments may relate to further improvements in the method, such as reducing its computational cost.

Bibliography

[1] *"IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device"*. IEEE Std 1687-2014.

[2] *"IEEE Standard for Test Access Port and Boundary-Scan Architecture"*. IEEE Standard 1149.1-2013, 2013.

[3] F.G. Zadegan et al. *"Design Automation for IEEE P1687"*. Design, Automation, Test in Europe Conference and Exhibition, 2011.

[4] Kuen-Jong Lee and Melvin A. Breuer. *"A Universal Test Sequence for CMOS Scan Registers"*. IEEE Custom Integrated Circuits Conference (CICC), pp. 28.5/1–4, 1990.

[5] S.R. Makar and E.J. McCluskey. *"ATPG for Scan Chain Latches and Flip-Flops"*. IEEE VLSI Test Symp. (VTS), pp. 364–369, 1997.

[6] Fan Yang et al. *"On the Detectability of Scan Chain Internal Faults – An Industrial Case Study"*. IEEE VLSI Test Symp. (VTS), pp. 79–84, 2008.

[7] Cantoro Riccardo, Montazeri Mehrdad, Sonza Reorda Matteo, Ghani Zadegan Farrokh, and Larsson Erik. *"On the Testability of IEEE 1687 Networks"*. IEEE Asian Test Symposium, pp. 211-216, 2015.

[8] Cantoro Riccardo, Palena Marco, Pasini Paolo, and Sonza Reorda Matteo. *"Test Time Minimization in Reconfigurable Scan Networks"*. 2016 IEEE 25th Asian Test Symposium (ATS), 2016.

[9] Tšertov Anton, Jutman Artur, Devadze Sergei, Sonza Reorda Matteo, Larsson Erik, Ghani Zadegan Farrokh, Cantoro Riccardo, Montazeri Mehrdad, and Krenz-Baath Rene. *"A Suite of IEEE 1687 Benchmark Networks"*. IEEE International Test Conference (ITC), 2016.

[10] A.T. Dahbura et al. *"An optimal test sequence for the JTAG/IEEE P1149.1 test access port controller"*. IEEE International Test Conference (ITC), pp. 55-62, 1989.

[11] Y. Blaquière et al. *"Design and validation of a novel reconfigurable and defect tolerant JTAG scan chain"*. IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2559 – 2562, 2014.

[12] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. *"Modeling, verification and pattern generation for reconfigurable scan networks"*. IEEE International Test Conference, pp. 1–9, 2012.

[13] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. *"Reconfigurable scan networks: Modeling, verification, and optimal pattern generation"*. ACM Transactions on Design Automation of Electronic Systems, vol. 20, no. 2, pp. 30:1–30:27, 2015.

[14] R. Krenz-Baath, F. Ghani Zadegan, , and E. Larsson. *"Access time minimization in IEEE 1687 networks"*. IEEE International Test Conference, pp. 1–10, 2015.

[15] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. *"Scan pattern retargeting and merging with reduced access time"*. IEEE European Test Symposium, pp. 1–7, 2013.

[16] M. A. Kochte, R. Baranowski, M. Schaal, and H. J. Wunderlich. *"Test strategies for reconfigurable scan networks"*. IEEE Asian Test Symposium, pp. 113–118, 2016.

[17] Riccardo Cantoro, Farrokh Ghani Zadegan, Marco Palena, Paolo Pasini, Erik Larsson, and Matteo Sonza Reorda. *"Test of Reconfigurable Modules in Scan Networks"*. IEEE transactions on computers.

[18] R. Cantoro et al. *"On the Diagnostic Analysis of IEEE 1687 Networks"*. IEEE European test Symposium (ETS), 2016.

[19] M. Bushnell and V. Agrawal. *"Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits"*. Kluwer Academic Publ., 2000.

[20] MicroGP web site. Online: http://ugp3.sourceforge.net/.

[21] https://sourceforge.net/p/ugp3/wiki/Genetic20operators/.

[22] S.R. Makar and E. J. McCluskey. *"On the Testing of Multiplexers"*. IEEE International Test Conference (ITC), pp. 669-679, 1988.

[23] F.G. Zadegan et al. *"Access Time Analysis for IEEE P1687"*. IEEE Trans. on Computers, Vol. 61, No. 10, pp. 1459 – 1472.

[24] F.G. Zadegan, U. Ingelsson, G. Asani, G. Carlsson, and E. Larsson. *"Test Scheduling in an IEEE P1687 Environment with Resource and Power Constraints"*. IEEE Asian Test Symp. (ATS), pp. 525-531, 2011.

[25] E.J. Marinissen et al. *"A set of benchmarks for modular testing of SOCs"*. IEEE International Test Conference (ITC), pp.519–528, 2002.

[26] D. Ull, M. A. Kochte, and H. J. Wunderlich. *"Formal verification of secure reconfigurable scan network infrastructure"*. IEEE Asian Test Symposium, 2017.

List of Figures

List of Tables

Appendix

.1 N17D3

```
<?xml version="1.0" encoding="utf-8"?>
<Gateway Version="2" xmlns="http://..." xmlns:xsi="http://..." xsi:schemaLocation="http://...">
 <SCB ID="16" SCLengthA="10" SCLengthB="11" />
 <SCB ID="15" SCLengthA="14" SCLengthB="20" />
 <SCB ID="14" SCLengthA="28" SCLengthB="21" />
 <SIB ID="13">
  <TDR ID="13.t" SCLength="5" />
  <SIB ID="12">
   <TDR ID="12.t" SCLength="13" />
   <SIB ID="11">
    <TDR ID="11.t" SCLength="10" />
    <SCB ID="10" SCLengthA="6" SCLengthB="21" />
    <SCB ID="9" SCLengthA="24" SCLengthB="17" />
    <TDR ID="8" SCLength="31" />
   </SIB>
  </SIB>
 </SIB>
 <TDR ID="7" SCLength="26" />
 <SCB ID="6" SCLengthA="3" SCLengthB="18" />
 <SCB ID="5" SCLengthA="14" SCLengthB="5" />
 <SIB ID="4">
  <TDR ID="4.t" SCLength="18" />
  <SIB ID="3">
   <TDR ID="3.t" SCLength="27" />
   <SIB ID="2">
    <TDR ID="2.t" SCLength="4" />
   </SIB>
   <SIB ID="1">
    <TDR ID="1.t" SCLength="25" />
    <TDR ID="0" SCLength="15" />
    <TDR ID="21" SCLength="16" />
   </SIB>
  </SIB>
  <SCB ID="22" SCLengthA="31" SCLengthB="14" />
 </SIB>
</Gateway>
```

.2 N32D6

```
<?xml version="1.0" encoding="utf-8"?>
<Gateway Version="2" xmlns="http://..." xmlns:xsi="http://..." xsi:schemaLocation="http://...">
<TDR ID="31" SCLength="1667" />
<SCB ID="30" SCLengthA="3232" SCLengthB="2520" />
<SCB ID="29" SCLengthA="480" SCLengthB="1882" />
<TDR ID="28" SCLength="3307" />
<SIB ID="27">
<TDR ID="27.t" SCLength="3933" />
<SIB ID="26">
<TDR ID="26.t" SCLength="270" />
<SCB ID="25" SCLengthA="1097" SCLengthB="3454" />
<SIB ID="24">
<TDR ID="24.t" SCLength="1412" />
<TDR ID="23" SCLength="4139" />
<SCB ID="22" SCLengthA="2012" SCLengthB="2403" />
</SIB>
<SCB ID="21" SCLengthA="2035" SCLengthB="800" />
</SIB>
<TDR ID="20" SCLength="1785" />
</SIB>
<SCB ID="19" SCLengthA="3601" SCLengthB="506" />
<SIB ID="18">
<TDR ID="18.t" SCLength="2671" />
<SIB ID="17">
<TDR ID="17.t" SCLength="2227" />
<SIB ID="16">
<TDR ID="16.t" SCLength="308" />
<SCB ID="15" SCLengthA="1293" SCLengthB="4202" />
<SIB ID="14">
<TDR ID="14.t" SCLength="2442" />
</SIB>
<SCB ID="13" SCLengthA="4210" SCLengthB="2967" />
<TDR ID="12" SCLength="4125" />
</SIB>
<TDR ID="11" SCLength="1122" />
</SIB>
</SIB>
<TDR ID="10" SCLength="3473" />
<SCB ID="9" SCLengthA="406" SCLengthB="343" />
<SIB ID="8">
<TDR ID="8.t" SCLength="1153" />
<TDR ID="7" SCLength="1297" />
</SIB>
<SIB ID="6">
<TDR ID="6.t" SCLength="2195" />
</SIB>
<SIB ID="5">
<TDR ID="5.t" SCLength="2632" />
<TDR ID="4" SCLength="3314" />
</SIB>
<TDR ID="3" SCLength="2692" />
<TDR ID="2" SCLength="139" />
<SIB ID="1">
<TDR ID="1.t" SCLength="1028" />
<SCB ID="0" SCLengthA="101" SCLengthB="3773" />
<SIB ID="-1">
<TDR ID="-1.t" SCLength="3091" />
</SIB>
```

```
<SIB ID="-2">
<TDR ID="-2.t" SCLength="4396" />
</SIB>
</SIB>
</Gateway>
```

.3 Mingle

```
Module Mingle {
  Attribute lic = 'h d3a0c41d;
  Parameter regSize = 32;
  Parameter regSize1 = $regSize;
  Parameter regSize2 = $regSize;
  Parameter regSize3 = $regSize;
  Parameter regSize4 = $regSize;
  Parameter regSize5 = $regSize;
  Parameter regSize6 = $regSize;
  Parameter regSize7 = $regSize;
  Parameter regSize8 = $regSize;

  ScanInPort SI;
  CaptureEnPort CE;
  ShiftEnPort SE;
  UpdateEnPort UE;
  SelectPort SEL;
  ResetPort RST;
  TCKPort TCK;
  ScanOutPort SO {
  Source SIB2.SO;
  }

// Level 1
Instance SIB1 Of SIB_mux_pre {
InputPort SI = SI;
InputPort fromSO = SCB3.SO;
}
Instance SIB2 Of SIB_mux_pre {
InputPort SI = SIB1.SO;
InputPort fromSO = SCB2.SO;
}

// Branch A
LogicSignal sel_Void1 {
SIB2.toSEL & ~SCB1.toSEL & ~SCB2.toSEL;
}
LogicSignal sel_WI1 {
SIB2.toSEL & SCB1.toSEL & ~SCB2.toSEL;
}
LogicSignal sel_SIB3 {
SIB2.toSEL & SCB2.toSEL;
}
LogicSignal sel_SIB4 {
SIB2.toSEL & SCB2.toSEL;
}
LogicSignal sel_SCB1 {
SIB2.toSEL & SCB2.toSEL;
}
Instance WI1 Of WrappedInstr {
InputPort SI = SIB2.toSI;
InputPort SEL = sel_WI1;
Parameter Size = $regSize1;
}
Instance Void1 Of BypassReg {
InputPort SI = SIB2.toSI;
InputPort SEL = sel_Void1;
}
```

```
Instance SCB1 Of SCB {
InputPort SI = SIB2.toSI;
InputPort SEL = sel_SCB1;
}
ScanMux sMux1 SelectedBy SCB1.DO {
1'b0 : Void1.SO;
1'b1 : WI1.SO;
}
Instance SIB3 Of SIB_mux_pre {
InputPort SI = SCB1.SO;
InputPort fromSO = SIBpost2.SO;
InputPort SEL = sel_SIB3;
}
Instance SIBpost1 Of SIB_mux_post {
InputPort SI = SIB3.toSI;
InputPort fromSO = WI3.SO;
}
Instance WI3 Of WrappedInstr {
InputPort SI = SIBpost1.toSI;
Parameter Size = $regSize3;
}
Instance SIBpost2 Of SIB_mux_post {
InputPort SI = SIBpost1.SO;
InputPort fromSO = WI4.SO;
}
Instance WI4 Of WrappedInstr {
InputPort SI = SIBpost2.toSI;
Parameter Size = $regSize4;
}
Instance SIB4 Of SIB_mux_pre {
InputPort SI = SIB3.SO;
InputPort fromSO = WI2.SO;
InputPort SEL = sel_SIB4;
}
Instance WI2 Of WrappedInstr {
InputPort SI = SIB4.toSI;
Parameter Size = $regSize2;
}

ScanMux sMux2 SelectedBy SCB2.DO {
1'b1 : SIB4.SO;
1'b0 : sMux1;
}
Instance SCB2 Of SCB {
InputPort SI = sMux2;
}

// Branch B
LogicSignal sel_SIB5 {
SIB1.toSEL & ~SCB3.toSEL;
}
LogicSignal sel_SIBpost3 {
SIB1.toSEL & SCB3.toSEL;
}
Instance SIB5 Of SIB_mux_pre {
InputPort SI = SIB1.toSI;
InputPort SEL = sel_SIB5;
InputPort fromSO = SIB6.SO;
}
Instance WI5 Of WrappedInstr {
```

```
InputPort SI = SIB5.toSI;
Parameter Size = $regSize5;
}
Instance SIB6 Of SIB_mux_pre {
InputPort SI = WI5.SO;
InputPort fromSO = WI6.SO;
}
Instance WI6 Of WrappedInstr {
InputPort SI = SIB6.toSI;
Parameter Size = $regSize6;
}
Instance SIBpost3 Of SIB_mux_post {
InputPort SEL = sel_SIBpost3;
InputPort SI = SIB1.toSI;
InputPort fromSO = SIB7.SO;
}
Instance WI7 Of WrappedInstr {
InputPort SI = SIBpost3.toSI;
Parameter Size = $regSize7;
}
Instance SIB7 Of SIB_mux_pre {
InputPort SI = WI7.SO;
InputPort fromSO = WI8.SO;
}
Instance WI8 Of WrappedInstr {
InputPort SI = SIB7.toSI;
Parameter Size = $regSize8;
}

ScanMux sMux3 SelectedBy SCB3.DO {
1'b0 : SIB5.SO;
1'b1 : SIBpost3.SO;
}
Instance SCB3 Of SCB {
InputPort SI = sMux3;
}

}
```

.4 TreeBalanced

```
Module TreeBalanced {
  Attribute lic = 'h 621bd14e;
  ScanInPort SI;
  CaptureEnPort CE;
  ShiftEnPort SE;
  UpdateEnPort UE;
  SelectPort SEL;
  ResetPort RST;
  TCKPort TCK;
  ScanOutPort SO {
  Source sib0.SO;
}

// Level 1
Instance sib0 Of SIB_mux_pre {
InputPort SI = SI;
InputPort fromSO = sib2.SO;
}

// Level 2
Instance sib1 Of SIB_mux_pre {
InputPort SI = sib0.toSI;
InputPort fromSO = sib3.SO;
}
Instance sib2 Of SIB_mux_pre {
InputPort SI = sib1.SO;
InputPort fromSO = m8.SO;
}
// Level 3
Instance m6 Of H953_Basic::M6 {
InputPort SI = sib1.toSI;
}
Instance sib3 Of SIB_mux_pre {
InputPort SI = m6.SO;
InputPort fromSO = m2.SO;
}
Instance m1 Of H953_Basic::M1 {
InputPort SI = sib2.toSI;
}
Instance m4 Of H953_Basic::M4 {
InputPort SI = m1.SO;
}
Instance m8 Of H953_Basic::M8 {
InputPort SI = m4.SO;
}
// Level 4
Instance sib4 Of SIB_mux_pre {
InputPort SI = sib3.toSI;
InputPort fromSO = m5.SO;
}
Instance m2 Of H953_Basic::M2 {
InputPort SI = sib4.SO;
}
// Level 5
Instance sib5 Of SIB_mux_pre {
InputPort SI = sib4.toSI;
InputPort fromSO = sibM7.SO;
}
```

```
Instance m5 Of H953_Basic::M5 {
InputPort SI = sib5.SO;
}
// Level 6 and lower
Instance m3 Of H953_Basic::M3 {
InputPort SI = sib5.toSI;
}
Instance sibM7 Of SIB_mux_pre {
InputPort SI = m3.SO;
InputPort fromSO = m7.SO;
}
Instance m7 Of EmptyModule_NoBidirs  {
InputPort SI = sibM7.toSI;
Parameter inputs = 80;
Parameter outputs = 32;
}

}
```