Doctoral Dissertation
Doctoral Program in Control and Computer Engineering (33rd cycle)

Effective techniques for systems validation and security

Aleksa Damljanovic

* * * * * *

Supervisor
Prof. Giovanni Squillero, Supervisor

Politecnico di Torino
17th September, 2021

. . . .
Aleksa Damljanovic
Turin, 17th September, 2021

Acknowledgements

To my parents without whom I would never be the person I am today, to my other half - my brother, to my grandparents who gave me so much love.

I don't have many close and special friends in my life, but those I have I keep close to my heart. During last three years I've missed them so much, nevertheless, they were my support and strength. Thank you Aleksandra, Sofija, Adam, Petar, Ruzica and Andrijana for being the part of my life. Many thanks to a colleague and first new friend in Turin, Esteban Josie Rodriguez Condia, to my friend, roommate and professor of Italian (and Sicilian) language, Andrea Floridia, and someone with the kind heart and beautiful smile (to whom I have been the worst ski instructor) Annachiara Ruospo.

I would like to thank Matteo Sonza Reorda for giving me his trust and opportunity to do a PhD, and Giovanni Squillero for his undivided support as my tutor. I would also like to thank Ernesto Sanchez for his collaboration and Riccardo Cantoro for helping me to start my research.

Artur Jutman and Anton Tsertov from Testonica deserve thanks from my side for their industrial expertise and fruitful discussions that were essential for my research.

Special thanks go to all people involved in the RESCUE project during 3 years of joint research.

Without all of these people, this thesis would never see the light of the day and my life would never be the same. This journey was an extraordinary life experience; it was not only a school of PhD but also a school of life.

Ko sme taj može, ko ne zna za strah,
taj ide napred.

Abstract

The increasing number of embedded instruments used to perform test, monitoring, calibration and debug within a semiconductor device has called for a brand new standard—the IEEE 1687. Such a standard resorts to a *Reconfigurable Scan Network* to provide efficient, reliable and flexible access to instruments and to handle complex structures. As it has to deliver reliable service, many approaches, both formal and simulation-based, have been proposed in the literature to perform test, diagnosis, and verification of such networks.

So far, most of the test-generation approaches were either too computationally demanding to be applied in complex cases, or too approximate to yield high-quality tests. A recent idea has been exploited in this thesis in a following manner: the state of a generic reconfigurable scan chain is modelled as a finite state automaton and a low-level fault, as an incorrect transition; it then proposes a new algorithm for generating a functional test sequence able to detect all incorrect transitions far more efficiently than previous ones. Such an algorithm is based on a greedy search, and it is able to postpone costly operations and eventually minimize their number. Experimental results demonstrate that the proposed approach is broadly applicable; has limited computational requirements; and the test sequences are order of magnitudes shorter than the ones previously generated by approximate methodologies. Together with testing the system for defects that may affect the scan chains themselves, the diagnosis of such faults is also important. Therefore, a method has been proposed for generating stimuli to precisely identify permanent high-level faults in a IEEE 1687 reconfigurable scan chain. A chapter is dedicated to the problem of post-silicon validation of a network, a problem that has not been adequately addressed, yet. The mismatches between the specification and its silicon implementation were analyzed, and then a methodology was proposed to detect a subset of them by applying functional patterns and observing the length of the active scan path.

While reconfigurable scan networks are commonly used to provide fault management and embedded instrumentation access, such as safety mechanisms, in advanced safety- and mission-critical electronic systems, a failure in such infrastructure itself has a high severity. Another aspect this thesis addressed is assessment and mitigation of NBTI aging induced delays in logic paths within IEEE 1687

IJTAG Reconfigurable Scan Networks. This methodology is based on a scalable hierarchical (transistor-to-architecture) modelling of the NBTI impact on timing-critical logic paths in RSN implementations. The evaluation implies analysis of gate input signal probabilities based on the configurations and test data selected for the RSN infrastructure.

A fundamental part of the new IEEE Std 1687 is the Instrument Connectivity Language (ICL), which allows for abstract description of the scan network. The big novelty if compared to legacy solutions like BSDL is the possibility of describing new topology-enabling elements such as the ScanMuxes in a behavioural way which can be easily and efficiently exploited by Test Generation Tools to retarget instrument-level operations to top-level patterns. This means that for a given design, the Developer will have to write both the RTL and the ICL descriptions: to the author's best knowledge there is no automated tool to make the translation RTL to ICL. This methodology is error-prone due to the human factor, the difference in intent in the two descriptions and the syntactic and semantic complexity of the languages. Incoherence between ICL and RTL will result in retargeting errors, so it is fundamental to validate the equivalence between the two descriptions. In this thesis an automated methodology is presented that starting from the ICL description is able to generate a set of RTL testbenches that can be simulated against the original RTL model to detect discrepancies and incoherence, and provides quantitative metrics in terms of code and functional coverage.

Experimental results for these approaches are reported on the set of ITC2016 set of benchmark networks.

Recent trends in integrated circuits industry include decentralization of the production flow by involving different integration teams, third-party IP vendors and other untrusted entities. As a result, this is opening up a door to new types of attacks that may lead to devastating consequences, such as denial of service or data leakage. Therefore, the problem of ensuring hardware security has gained much attention in the last years, especially early in the design cycle, when an attacker may insert malicious circuitry at register transfer (RT) or gate level – a Hardware Trojan. Due to the increased complexity of modern devices, the research community is spending a lot of effort in developing more sophisticated detection methodologies and smarter attacks. However, the main problem is that they are validated on the existing benchmarks that do not reflect the real complexity. Trying to fill this gap, this thesis proposes a set of RT-Level Hardware Trojan benchmarks injected in a RISC-based pipelined microprocessor core. To prove the viability, the impacts on area, power and frequency are presented and discussed. For any proposed Hardware Trojan, the functional description, the implementation details and the effects once activated are provided.

Furthermore, despite the considerable effort that has been invested in this area, the evergrowing complexity of the modern devices always calls for sharper detection methodologies. In this regard, the last chapter of the thesis illustrates a pre-silicon, simulation-based techniques to detect Hardware Trojans. The technique exploits well-established machine learning algorithms. All of the background concepts are presented together with the methodology and the automatized flow. The validity of the approach has been demonstrated on the *AutoSoC* CPU, an industrial-grade, safety-oriented, automotive benchmark suite. Experimental results demonstrate the applicability and effectiveness of the approach: the proposed technique is highly accurate in pinpointing suspicious code sections. None of the Trojans from the set has been left undetected.

Contents

List of Tables

List of Figures

Introduction

Nanoelectronic systems are at the core of all industry sectors and are deployed in many life-critical domains such as automotive, security, healthcare, etc. [1] In the recent decades we witnessed new advances and immense progress in this field [2]. Emerging technologies, shrinking of the technology nodes, but above all the complexity of modern devices and the integration of many different components within a system pose significant challenges in terms of design and mutually dependent aspects of security, reliability and quality [3]–[5].

Given the criticality of their application and the role nanoelectronic systems have as a backbone of the large infrastructure and Cyber-Physical systems, the need of providing the intended functionality as well as being reliable and secure is of utmost importance. On the other hand, due to the complexity and miniaturization of the process nodes towards the physical limits, such requirements are getting increasingly hard to satisfy. How much will it cost? Does it satisfy initial requirements? How robust it is? Will it work and how efficient will be? How much time does it take to design and produce? These are only some of the most important questions to answer during the product's life-cycle (Fig. 7.1).

As soon as integrated circuits (ICs) emerged, the need for design tools became evident. Obviously, the design of modern ICs containing billions of devices would be infeasible without software tools that are used at every stage [6]. Related tools and methodologies that are used are called electronic design automation (EDA) [7]. It can be said that EDA, i.e. Computer-Aided Design (CAD) tools revolutionized the IC industry and continue to do so even today. They can be used to develop new designs, reuse the existing ones, and integrate entire systems [8], [9]. Such tools can have different role in the process of design, manufacturing, and test: simulation (electrical, logic gate or high-level), ATPG test vector generation, layout design, placement and route, logic synthesis, design optimization (timing, power and cost/area) and device performance prediction, verification (formal, functional, equivalence checking), etc.

A variety of complementary tools and methods were added to conventional design flows. Furthermore, library vendors started offering a whole spectrum of libraries for optimal design choices (high-performance, low-power, etc.). In that way, design was made portable across processes and foundries, thus making the whole cycle significantly more efficient. Some new design areas have matured in the meantime: gate sizing (choosing the best transistor width to obtain better performance), clock design and synthesis (reliable synchronization due to high number of sequential elements), three-dimensional ICs (continuing to follow the Moore's law and stack ICs one on top of the other).

In general, all of the listed types of tools not only automate the work of engineers, but also process large amounts of heterogeneous data. In comparison with human designers they are also definitely able to perform more accurate analyses and more efficient and advanced optimizations.

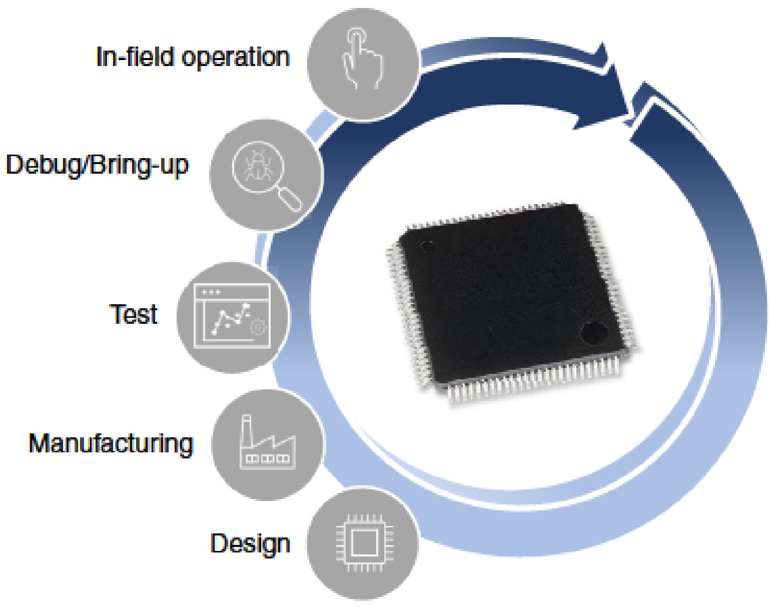


Figure 1: Life cycle of an IC

This thesis focuses on the new techniques developed to support the designer of nanoelectronic systems in the validation of their correctness. This task requires considering not only the space of all possible scenarios where the system is used, but also a further dimension represented by the possible hardware faults and external attacks the system is designed to face. Assessing the correct functionality of the system with such a huge combination of possibilities can only be done by combining different techniques coming from different communities (e.g., the one of software validation, the one of hardware validation, the one of hardware testing) and exploiting different paradigms (e.g., resorting to formal techniques, simulation, to evolutionary computation, etc.).

The growing adoption of electronic systems in systems where safety is a concern asks for strict requirements in terms of fault prevention , detection , and management , so that the probability of occurrence of any failure due to a hardware fault is kept under the acceptable threshold. More recently, another issue arose, related to the attacks that may be purposely driven against an electronic system to steal the information it stores, or to modify its behavior . Due to the increasing frequency of these attacks, it is crucial that the system is designed in such a way that the chances of their success are minimized (security) [10]–[16]. The solutions adopted by the designer to face the requirements in terms of both safety and security must be assessed in terms of correctness and effectiveness (validation).

The bridge that exists between pre-silicon validation (verification) and post-silicon validation, has to be mitigated to alleviate time-to-market pressure. With the growing design complexity, technology scaling, with improving performance and high levels of integration, design validation challenges are continuously increasing. Time required to perform design validation and verification is extremely consuming and in some complex cases may occupy most of product design cycle. As reported in [17] design verification as the most important aspect of the product development process can consume as much as 80% of the total product development time. What is even more critical is the cost needed to find and correct design errors later in the design flow [18]. Another issue is that often system design requirements in terms of security and system validation do not go along with each other.

Dependability has become a key concern, demanding the presence of numerous and various resources embedded within integrated circuits (ICs). These are used for supporting test, such as Built-In Self-Test (BIST) modules , for monitoring internal parameters such as current, temperature or delay sensors and configuration/calibration of different modules through registers . Due to the requirements for having a large number of these devices it became infeasible to include them into the single scan chain or provide a separate instruction for accessing every single one, relying on IEEE 1149.1. Additionally, instruments (or IPs) could easily originate from different vendors, making the integration a long and complicated procedure. There was not a simple and standardized way to perform the integration, but more a set of ad-hoc solutions based on the instructions from the IP provider given to the ASIC test/design engineer. It included extensive learning and different setup for each new IP, as well as integration into the latest ASIC and latest process node.

A solution for simplifying not only the access to embedded resources and reducing the overhead, but also alleviating the aforementioned issues with the integration has been described and published as the IEEE 1687 standard [19]. The standard has enabled designers to flexibly trade-off between area, access time and other parameters, since the scan chain accessible through the JTAG's Test Access Port (TAP) can now be split and configured. Furthermore, it offers a standard test hardware description, and a standard test procedure language for each IP. More details about the standard itself and so called, Reconfigurable Scan Networks (RSNs) will be given in Chapter 1.

The correct operation of IJTAG-compliant infrastructure is a product of many aspects and components including the actual hardware on the chip, the respective standard descriptions, such as ICL (Instrument Connectivity Language) and PDL

(Procedure Description Language) files as well as the software used to import the descriptions and control the hardware. Being a relatively new standard, there are still many different aspects insufficiently explored related to the infrastructure it describes, especially in the context of dependability. I believe that such research is essential for providing support and therefore wide-spread usage of this infrastructure. The first part of this thesis is oriented on the following issues:

- post-manufacturing test and efficient techniques for generating configurations used to test reconfigurable modules within RSNs.
- identifying faults within RSNs (diagnosis)
- post-silicon validation of RSNs
- in-field use, and reliability issue related to NBTI-aging effect
- equivalence checking between ICL and RTL—verification

Another argument that this thesis will advance is the Hardware Security. In the recent years, the growing complexity of modern devices and the fabrication costs led the IC industry to pursue a new global business model. The half-trillion-dollar semiconductor supply chain is one of the world's most complex. The production of a single computer chip often requires more than 1,000 steps passing through international borders 70 or more times before reaching an end customer [20]. In that regard, even more companies around the world are deeply involved in all phases of the IC supply chain. The outsourcing of part of the process to untrusted third-party entities raises increasing concerns about the hardware security of the products. The problem of ensuring hardware security has gained much attention in the last ten years, especially early in the design cycle, when an attacker may insert malicious circuitry at register-transfer (RT) or gate level. Such type of attacks that may lead to devastating consequences, such as denial of service or data leakage.

Particularly, Hardware Trojans (HTs) are an important topic not only for industry and academia, but also for government bodies [21].

Hardware Trojans are modifications that an adversary is able to make in original circuitry to gain access to sensitive information (encryption keys), downgrade the performance, prevent user access or completely disable the device or its functions. It is assumed that the attacker is able to access the design (or its part) and perform maliciously modifications before or during fabrication.

What renders difficult to detect such alterations is:

- large number and the complexity of soft, firm and hard IPs that are present in modern SoC designs;
- the cost of applying reverse engineering and inspecting physically **each** device given their complexity and technology miniaturization;

- given the characteristics of the advanced technology used nowadays, physical measurements in a circuit infected by a Trojan may still stay within the margins due to process variation and thus, remain undetected;
- Trojan are by construction difficult to activate (a set of complex low-probability conditions) and while the attacker might have information about the design, the defender might not have any information about the Trojan, i.e., its type, size, position etc.;
- manufacturing tests that are applied on every device are ineffective for the reasons listed above.

Due to evolving attacks, the research community is spending a lot of effort in developing more and more sophisticated detection methodologies. However, the problem is that they are validated on the benchmarks that do not reflect the real complexity of the devices used for industrial applications, such as automotive. First contribution related to the aforementioned issue in the release of new benchmarks targeting a pipelined open-source RISC microprocessor core.

Although most of the detection techniques work at the gate level, shifting the detection of HTs inserted at RTL to the gate level would result in increased design and verification costs. That is why a new and efficient method for detecting such Trojans has been developed. It will be presented as a second contribution in this part of the thesis. A branch of artificial intelligence (AI) and computer science that continues to receive tremendous attention and has alleviated life to many engineers is Machine Learning (ML). This is especially true in the context of ICs, given the large amounts of available data from both production and use life-cycle phases.

The proposed methodology is based on ML technique for Hardware Trojans detection, based on a deep analysis of the RT-Level model. The analysis is based on dynamic and static properties extracted from such model. An Artificial Neural Network (ANN) as well as the Support Vector Machine (SVM) algorithm are used to identify suspicious code fragments potentially hiding a Trojan.

Part I

IJTAG Reconfigurable Scan Networks Dependability

Chapter 1

Background

In many of the latest ICs designers introduced resources whose purpose is not to support the circuit functionality, but rather to support ancillary features such as test, calibration, debug and monitoring. In particular, current ICs often integrate a plethora of sensors and actuators, each associated to a register to be read and/or written from the outside, sometimes at the end of the manufacturing process, sometimes during the operational phase. Many test solutions, such as BIST, also require registers to activate/initialize the test and retrieve results. In order to effectively access all these registers (also called instruments, or TDRs), companies used to include them into a single chain, often accessed through the standard IEEE 1149.1 interface. With the significant rise in complexity and the number of devices, existing infrastructure became inefficient. One of the limitations originated from the length of a single scan chain which was constantly increasing; performing access to communicate with a single device resulted in large time overhead. Moreover, the reliability of such structure became an issue, since a problem on a single bit would render the entire scan-chain non-operational and consequently, lead to a catastrophic breakdown. Another possibility involving architecture for accessing each instrument individually, apart from limited flexibility it provides, requires infeasible number of instructions to be implemented. To tackle these issues solutions based on so called Reconfigurable Scan Networks (RSNs) were introduced. 1149.1. was updated to support such constructs, while 1687 introduced increased flow flexibility and the possibility to define more complex structures in the network during the chip integration phase in a newly developed description language. Nevertheless, both 1687 and 1149.1-2013 have a lot of common advantage in terms of reuse, efficiency, automation and improved quality that affect several stages of chip development. They can provide a means of standardization of IP verification, insertion, internal IP test interface, etc. Both standards introduce dynamically reconfigurable scan chains.For the sake of simplicity as different terminology is used, basic network constructs will be described as a part of IEEE 1687 Std in Section 1.2.

1.1 IEEE 1149.1 – JTAG

As the devices' complexity increased and with the limitations in terms of number, cost and physical access to its pins, it became both impractical and inefficient to use them for accessing the device and test the interconnections between different ICs. To deal with these issues, IEEE 1149.1 industrial standard was devised. It is commonly known as Joint Tag Access Group (JTAG) and describes a testing infrastructure. Registers described by the standard, boundary scan cells, as their name suggests are composed of individual bits, or cells that are located at the boundary of the device. They are placed between the functional core and the pins or balls by which it is connected to a board. The standard also introduced the description of a logic block called Test Access Point (TAP). TAP consists of a controller in charge of executing access and data flow, which is a sequential state machine, Instruction Register, and a number of Data Registers providing write and read functionality (Fig. 1.1). Given its flexibility for implementing additional commands and adding new logic blocks, TAP soon found its alternative application for programming and debugging devices. The JTAG interface consists of 5 signals: Test Data In (TDI), Test Data Out (TDO), Test Clock (TCK), Test Reset (TRST) and Test Mode Select (TMS).

The TAP controller as defined by the IEEE-1149.1 standard represents a 16-state finite state machine. It is controlled by TCK and TMS. Each transition is determined on the rising edge of TCK, by the state of TMS. Two analogous paths through the state machine are used to capture and/or update data by scanning through the instruction register (IR) or through a data register (DR). The JTAG state machine is depicted in Fig. 1.2.

1.2 IEEE 1687 – IJTAG

Exacerbated by the increasing number of instruments within a single scan chain, the lack of flexibility is a major issue in both IEEE 1149.1 boundary-scan (JTAG) and IEEE 1500 core test standards, along with weaknesses of the test scheduling and scalability limitations. The new IEEE 1687 standard (IJTAG) [1] was designed to be able to deal with problems caused by the long scan chains and the substantial number of instructions required to access instruments. It exploits the idea of Reconfigurable Scan Networks (RSNs), residing between device interface and instrument interface and allowing a scan chain to be partitioned into segments that can be selectively included or excluded. Through dynamic configuration and variable-length scan-chain, IJTAG enables flexible and efficient access to all instruments. Thus, designers are able to take into account various configurations and choose the best trade-off between parameters such as area or access time. Although the standard does not impose an external access mechanism, the most widely accepted one is

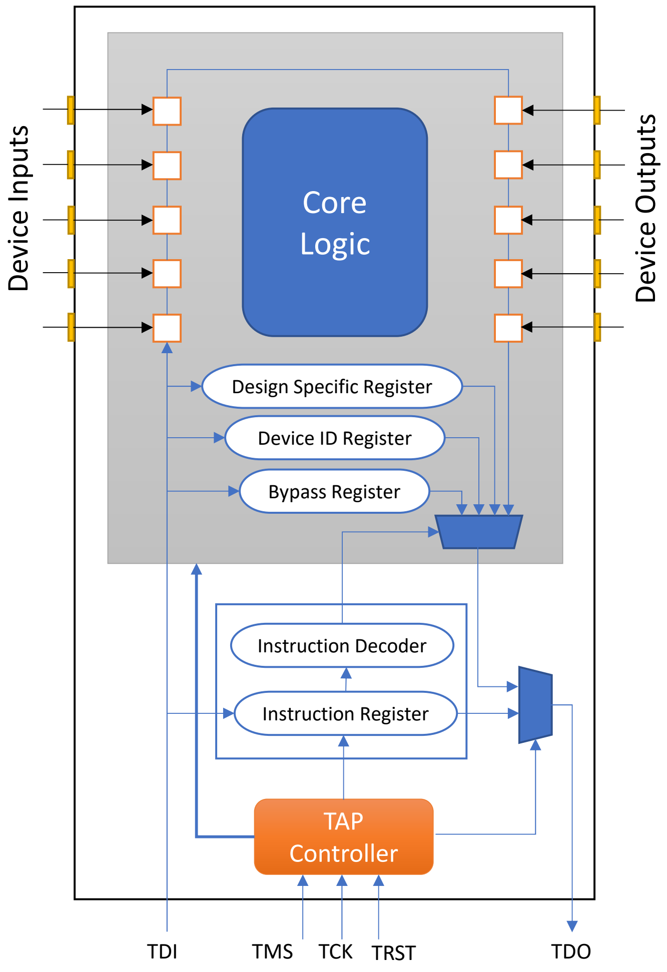


Figure 1.1: JTAG boundary scan with dedicated registers and TAP controller

TAP. The IEEE 1687 standard also introduces two languages: Instrument Connectivity Language (ICL) and Procedural Description Language (PDL) that allow describing the structure of the network and the protocol to access the different instruments.

To interface each instrument, a register of a variable length is used. This corresponds to a set of scan cells, IEEE 1149.1-compatible, referred to as a TDR. TDRs can be Read-Only, Write-Only or Read-Write. Furthermore, three operations are used to control the network and read/write data from/to TDRs: capture (C), shift (S), update (U).

Apart from TDRs, the network is composed out of two types of programmable modules. These are used to partition the set of instruments; including or excluding a set of instruments obviously has an effect on the scan chain length. A segment insertion bit (SIB) module behaves as a gateway with respect to the segment it

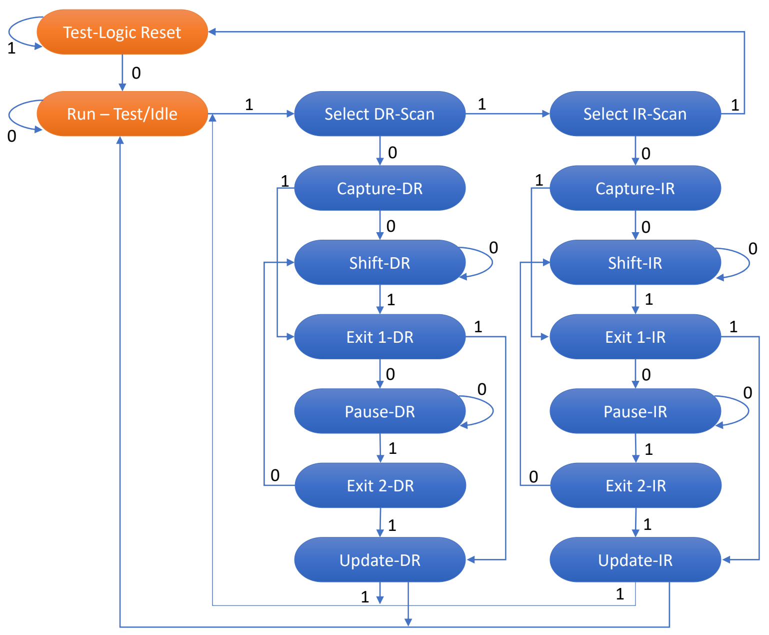


Figure 1.2: TAP controller state machine

controls: it is able either to bypass the segment or to include the segment into the active path (Fig. 1.3(a)). In the bypass state it is referred to as de-asserted, while it is said to be asserted when is configured to expand the scan chain. SIBs can be used to obtain a hierarchical structure of the network, allowing hierarchical access to the registers interfacing the instruments. The Fig. 1.3(b) shows a symbol used to represent a SIB.

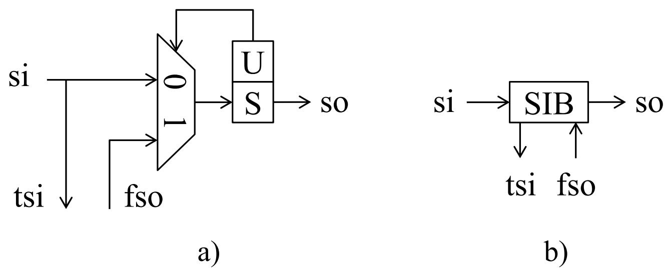


Figure 1.3: Segment Insertion Bit (SIB) module: Simplified schematic (left) and symbol (right)

Other than using a SIB to include or bypass a segment, a different module is used to support exchange of one scan chain segment for another. A scan multiplexer with shift-update cells (ScanMux, SM) can be seen as a configurable multiplexer, which is used to alter the scan chain by selecting which of its input branch segments are to be included into the active path. Thus, the existence of mutually exclusive scan chains is supported, reinforcing the trade-off of access time with access length. The example given in Fig. 1.4(a) shows a scan multiplexer with a two-bit shift-update control register which is used to choose one among four segments. The symbol shown in Fig. 1.4(b) will be used to represent a shift-update cell. Control registers of the modules consist out of two stage cells. A cell is referred to as a flip-flop with additional control logic. Shift (S) cell is a part of scan chain and it shifts values, while Update (U) cell stores the S cell value, when update operation is performed. The configuration of the module is defined by the value in the U cell. For example, a SIB module is configured by shifting in the desired value into the S scan cell, followed by an update, thus storing the value from the S cell to the U scan cell. Indicatively, as illustrated by Fig. 1.3(a), in this work a SIB is considered to be asserted if the latched bit is 1 and if so, it includes the path between tsi and fso terminals. Conversely, it is regarded as being in a de-asserted state if the latched bit is 0, bypassing the segment between tsi and fso terminals, directly connecting si and so through the S cell. The provided shortcut has the length of one bit.

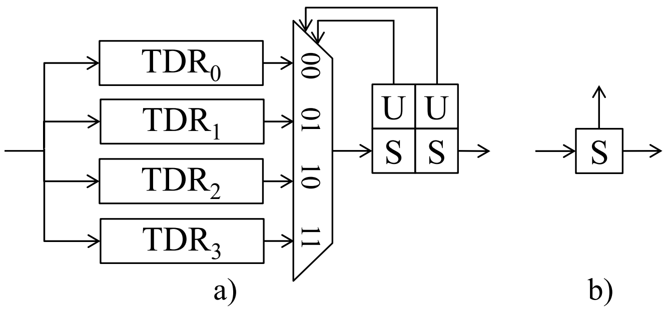


Figure 1.4: ScanMux (SM) module: Simplified schematic (left) and symbol (right)

Depending on the position of the configuration bit(s) with respect to the programmable module itself the module can be either inline or remote. A module is considered to be inline if its associated configuration cell is located in the same segment of the related module. Otherwise, it is said to be remotely controlled. Some basic architectural constructs, varying on the organization of TDRs, are provided in Figure 3. A simplest one is a flat structure where TDR is always accessible (Fig. 1.5a). MUXed TDRs enable mutually exclusive access (Fig. 1.5b), while excludable TDR is either a part of the active path or is bypassed (Fig. 1.5c). Partial configurations involve combining previous structures, thus in Fig. 1.5d (partially selectable TDRs) always two TDRs belong to the active path (one fixed, another selectable), while in Fig. 1.5e (partially excludable TDRs) one or two registers belong to the active path (path always includes one, while the other one can be inserted). The Fig. 1.5f shows the partially excludable and selectable configuration (one TDR is always accessible, while one of the remaining two can be included into the active path). These can be combined to design more complex networks.

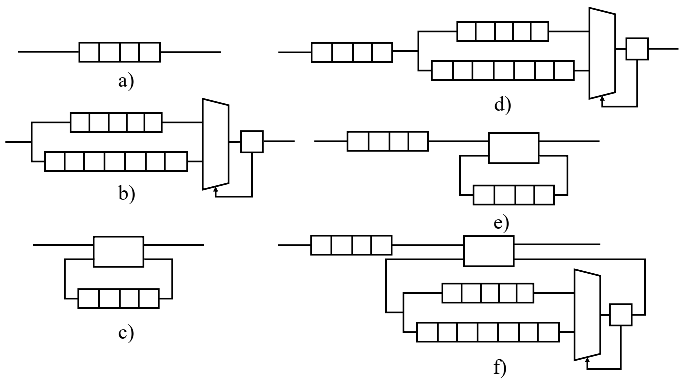


Figure 1.5

To keep the drawings simple inFig. 1.3 and Fig. 1.4, the clock, reset, control signals (namely, *shift*, *update*, and *capture*), and the *select* signal used to gate the control signals are not shown. To follow the examples in this work, it should suffice to assume that only the TDR connected to the selected port of a ScanMux receives (i.e., reacts to) the clock and control signals. It should be noted that the configuration of the network (i.e., the status of the latched bits) does not change when shifting a new vector through the shift cells, but only in the update phase where the shifted vector is latched into the U cells.

To operate an IEEE 1687 network from outside the chip, the IEEE 1149.1 TAP can be used. The TAP finite state machine provides the control signals needed to configure the IEEE 1687 network and access the instruments through it.

As an example, let us consider an RSN that includes five instruments: the user can access them through the TAP port, reading or writing from/to the associated Test Data Registers (TDR$_1$ to TDR$_5$). In order to save time when accessing to the instruments, the designer, instead of connecting all TDRs into a single chain, like in 1149.1-complaint circuits, may decide to adopt an IEEE 1687 network including three SIBs and one ScanMux (SM), as shown in Fig. 1.6; each of these four configuration modules can be configured to allow the access to a given subset of TDRs (and the associated instruments). Table 1.1 reports sixteen possible configurations supported by this network, which depend on how the SIBs and the ScanMux have been configured. In Table 1.1, “A” means asserted, “D” means de-asserted, 0 and 1 correspond to the two possible positions of the ScanMux, and “-” appears when a module belongs to an inaccessible segment.

In order to move the network to a given configuration, the user must first shift-in a suitable sequence of bits, so that the S flip-flops of SIBs and ScanMuxes hold the correct value, then activate the update signal to move these bits to the U latches. The sequence of bits to configure the network is called *configuration vector*. A generic configuration vector is referred to as cv_i. Once a configuration is reached, a given subset of the TDRs is accessible, which constitutes the so-called *active*

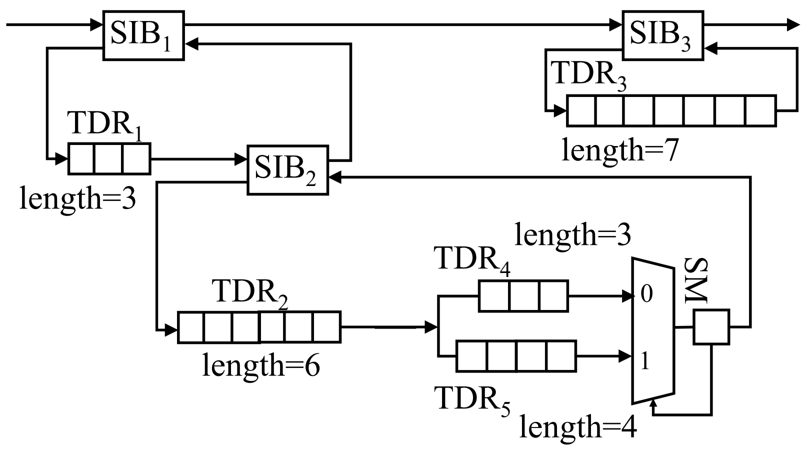


Figure 1.6: Example of IEEE 1687 RSN.

path. The rightmost column of Table 1.1 reports the length of the active path for each configuration, which corresponds to the number of TDR and S flip-flops in the path. The reader should note that moving from one configuration to another may require more than one configuration vector. For example, in the network of Fig. 1.6 moving from C_2 to C_{13} requires first turning SIB$_1$ into the asserted state (e.g., moving to C_8) and only then we will be able to change the configuration of the SM scan multiplexer to select input branch 1 and turn SIB$_2$ into the asserted state. Hence, moving from C_2 to C_{13} requires 2 configuration vectors.

1.3 Related works

In recent years the IEEE 1687 standard and RSNs have been subject of many research works, addressing test, verification, security and design. However, to our knowledge, apart from [22], this is the only work addressing the issue of permanent RSN fault diagnosis.

As already discussed, although reconfigurable scan networks introduced flexibility, minimizing access time has arisen as a potential issue. The authors in [23] analyzed various structures with different access scheduling to estimate overall access time. Additionally, the same authors developed the CAD tool to support design automation of optimized 1687 SIB networks [24]. Based on the access schedule and the set of instruments, the tool is able to design a network with optimized access time and low hardware overhead.

In [25] a general approach has been proposed to automatically generate a test sequence for an IEEE 1687 RSN with respect to permanent faults. It provides techniques for testing SIBs and ScanMuxes, and then it describes how to combine

Table 1.1: Set of possible configurations of the RSN in Fig. 1.6.

Configuration	SIB_1	SIB_2	SM	SIB_3	Active path	Length
C_0 C_1 C_4 C_5	D	D A	0 1 0 1	D	-	2
C_2 C_3 C_6 C_7	D	D A	0 1 0 1	A	TDR_3	9
C_8 C_9	A	D	0 1	D	TDR_1	6
C_{10} C_{11}	A	D	0 1	A	TDR_1, TDR_3	13
C_{12}	A	A	0	D	TDR_1, TDR_2, TDR_4	16
C_{13}	A	A	1	D	TDR_1, TDR_2, TDR_5	17
C_{14}	A	A	0	A	TDR_1, TDR_2, TDR_3, TDR_4	23
C_{15}	A	A	1	A	TDR_1, TDR_2, TDR_3, TDR_5	24

them into a single comprehensive test. This test is independent on the specific representation of the network elements and does not require any change in the hardware implementing the network. Test generation can directly start from the network's ICL description, as mandated by the IEEE 1687. The proposed test generation algorithms are based on different heuristics that could easily run even on relatively large RSNs.

In [26] that approach was refined to minimize the duration of the resulting test sequence: the faced problem was properly modelled according to the graph theory, and an optimal algorithm able to generate the minimum-duration test sequence was described. Unfortunately, such an approach works only on relatively small RSNs, and sub-optimal solutions must be accepted when dealing with real cases.

An alternative approach based on an evolutionary algorithm was developed in [27] to generate test sequence for a generic RSN with minimum duration. The usage of formal techniques to generate the minimum duration sequence able to test all reconfigurable modules in the network is explored in [28], providing a lower bound for small networks and thus assessing the effectiveness of the other approaches.

The authors of [22] analyzed the effect of permanent fault on RSN elements to determine diagnostic properties. The test sequence generated by the procedure described in [25] is further extended to enable localization of faults, i.e., to satisfy the defined properties.

A number of works also analyze the use of IEEE 1687 infrastructure to support on-line health monitoring and fault management [29]–[35].

Modelling, verification and optimal pattern generation is tackled in [36]. RSN formal model is presented considering structural and functional dependencies. The problem is transformed into Boolean Satisfability Problem. The formal method is also used for pattern retargeting, i.e., to generate scan-in data for reconfiguration and execution of commands in instrument access procedures. Moreover, the paper describes pattern generation method for efficient concurrent access. For the retargeting modelled as a sequential problem unrolled over number of time frames (CSU operations), the authors in [37] proposed a method for calculating an upper-bound on the number of required CSU operations. Knowing the upper-bound is used to deal with the model complexity for large designs and reduce the search space while preserving the optimum solution.

Defining and verifying security properties is addressed in [38]. In this work it is described how specified permissions and restrictions are transformed into predicated for a formal model unbounded checking.

The authors in [39] considered introducing some DfT modifications to enable observability of shadow registers and update logic. Moreover, different test methods are proposed for stuck-at, flip-flop transparency and bridge-faults in the RSN. Security in RSN is another important aspect considered is literature. In [40], [41], obfuscation strategies are proposed to modify the structure of the network by introducing additional logic for controlling the state of reconfigurable modules, as well as creating false paths to confuse the attacker. On the other hand, other works consider validating security properties and preventing unauthorized access by the means of external filter module both online and fixed-precomputed [42]–[44].

1.4 IEEE 1687 Benchmark RSNs

In [45] authors from both academia and industry proposed and published a set of benchmarks to enable fair and objective comparison of the developed methodologies across research groups. The benchmarks are also typical and challenging examples utilizing many different constructs and features supported by the standard. They can be classified in four different categories based on the architecture and purpose. Table 1.2 reports some basic information about the networks used to perform evaluation. In column 2 and 3, the table reports for each network the number of SIBs and SMs, respectively. The number of configuration bits of SIBs and SMs is given in the fourth column. 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, according to [45]). The column *Max path* reports the length of the longest path in the network, and the rightmost column the number of bits in all the TDRs.

Table 1.2: ITC'16 benchmark networks list

Network	SIB	SM	Tot. bits	Max depth	Max path	Scan cells
Mingle	10	3	13	4	171	270
TreeBalanced	43	3	48	7	5,219	5,581
TreeFlat_Ex	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
N132D4	39	40	79	5	2,555	2,991
N17D3	7	8	15	4	372	462
N32D6	13	10	23	4	84,039	95,158
N73D14	29	17	46	12	190,526	218,869
NE1200P430	381	430	811	127	88,471	108,148
NE600P150	207	194	401	78	23,423	28,250

Chapter 2

Test

When a circuit includes an RSN, the issue of testing the related hardware must clearly be considered, checking for possible defects affecting it. Failing to effectively solve this issue may lead to completely false results when using the RSN itself. Some works faced the issue of testing the test circuitry mandated by the IEEE 1149.1 standard [46], while other works focused on the test of possible permanent faults affecting a standard scan chain, e.g., by shifting into the chain a sequence of alternated 0s and 1s, and checking that the same sequence appears at the other extreme of the chain [47]–[49]. However, to test an RSN is a more complex task with respect to the standard scan chain test, since examining the ability of flip-flops comprising the scan chain to shift is not sufficient to guarantee correct functionality and expected performance. In addition, testing should check whether the network can be moved from one configuration to another and if it operates correctly after enforcing whichever legal configuration. Although testing an RSN clearly shares some similarities with the task of design validation [36], time required to perform test, i.e. test stimuli duration is considered to be more important with more strict limitations.

2.1 Fault model

Testing a non-reconfigurable scan chain for permanent faults can be performed by shifting a suitable sequence of 0s and 1s through the scan chain. RSNs are however far more complicated to test: in addition to flip-flops composing the TDRs, which have to be tested to check whether they can correctly shift values when included in the active path, the reconfigurable modules (i.e., SIBs and ScanMuxes) have to be also tested to check whether they are able to move the network to the corresponding configurations.

For all of the test-related techniques that will be described in this thesis, high-level fault model that was first introduced in [25] will be used. The faults affecting the reconfigurable modules, such as ScanMuxes, are modelled such that a different configuration is selected rather than the expected one. Such a fault leads to a different active path (called *faulty path*) than the expected one, and the two are likely to have a different length. For example, in Fig. 1.6 the multiplexer (ScanMux) may be affected by a permanent fault whose effect is that the segment connected to the input 0 is always selected, no matter the value in the selection cell. The same may arise for the generic SIB$_i$, which can be affected by faults named *stuck-at asserted* (SIB$_i$-s@A) and *stuck-at de-asserted* (SIB$_i$-s@D). The stuck-at faults in the scan bits of the selection cells are considered as detected by implication by testing such high-level faults, which cover also the faults affecting the update logic of the reconfigurable modules.

Moreover, such faults cover some faults affecting the reset logic, whose effect is that the module is stuck at the reset value. The other reset faults (i.e., those that make the reset ineffective) are not considered but can be targeted by employing the techniques described in [39].

2.2 Test procedure

Resorting to the high-level fault model, one can test an RSN by first configuring the RSN so that the target fault is excited, and then comparing the length of the activated path against the length of the expected path. Since the number of possible configurations of a network grows exponentially with the number of configurable modules, the problem of identifying a sequence of sessions which guarantees that 1) all the configurations modules are fully tested, and 2) the total test duration is minimized, is not trivial. Coming back to the example of Fig. 1.6, this means identifying the sequence of configurations (out of the 16 possible ones) that matches the two above goals.

As an example, the high-level fault that affects the ScanMux of Fig. 1.6, to always select the segment connected to the input 1, can be excited by a configuration which selects the input 0; configurations C_{12} and C_{14} fulfil this requirement. Once one of them is activated, one can measure the length of the active path by shifting a given sequence (called *test vector*) in TDI and checking when it will appear on TDO. Any fault modifying the length of the active path can be detected in this way. A generic test vector is referred to as tv_i in this paper.

In order to test all configurable modules in a RSN, the test sequence can be organized in *sessions*: in each session the network is first configured via one or more configuration vectors (so that each SIB and each ScanMux is switched into a given position), and then a check is performed for whether the expected path has been inserted between TDI and TDO via test vector, i.e., whether right segments can be accessed.

A proper test sequence consists of an alternating bits sequence 0101..., as long as the active path length followed by a sequence terminator, such as two consecutive 0s or 1s. For example, if the network in Fig. 1.6 is configured to C_8 (see Table 1.1), a proper test vector is 010101010**11**, that is, 9 bits of alternated 0s and 1s followed by the sequence terminator. Faults affecting the network may corrupt the network by changing the active path, which will cause the sequence terminator to be observed on the scan output in an unexpected clock cycle. For example, if a stuck-at fault affects the selection of the module SIB_1 (which is supposed to be asserted in the fault-free scenario), then the network may exclude the SIB_1's controlled segment, as in the SIB_1's de-asserted case. Thus, the active path selected in such a faulty scenario would be the same of the configuration C_0 of Table 1.1. In the faulty scenario, the path length is 2, meaning that the sequence terminator is observed earlier than expected on the scan output pin.

The complete process consists of the following steps:

- shifting in the sequence consisting of same values (all 0s or all 1s), while the length of the sequence is equal to the length of the longest path in the network; the goal of this phase is the initialization of the scan cells;
- shifting in the second, test sequence, of alternated 0s and 1s (i.e., 0101...01), with the predetermined (expected) length
- the last sequence shifts values from the currently active path. Determining fault-caused modifications of values in the scan chain and the length of the active scan path is performed by verifying previously inserted test vector; in parallel with observing the values appearing at the output, new configuration vector is shifted in.

Applying the configuration vector demands an update operation. The duration of the complete test procedure, referred to as a total cost, depends on the duration of each step and is composed of configuration step cost and test step cost, both expressed in terms of number of clock cycles. The configuration step cost is the time needed to apply configuration vectors. The time overhead of the JTAG protocol is also included, since moving the TAP controller from shift to update state and vice versa also requires a few clock cycles (Fig. 1.2). The test phase cost is the time required to shift in the test sequence. Furthermore, the duration of a session is determined by the length of the TDRs included in the path, as well as by the previous configuration.

In this thesis several testing procedures are presented. Two are semi-formal techniques to generate the set of configurations that yield high coverage and low test time based on representing the network as a Finite State Machine, while the third one uses meta-heuristic evolutionary approach combined with post-processing techniques.

1. Perform reset ($\mathrm{SIB}_1 \rightarrow \mathrm{SIB}_3$)

Table 2.1: Test procedure for the network in Fig.1.6

Input	Fault free	SIB_1		SIB_2		SM		SIB_3	
	$\bar{s}$	s@D-s^0	s@A-s^1	s@D-s^2	s@A-s^3	s@0-s^4	s@1-s^5	s@D-s^6	s@A-s^7
reset observe	DD0D (2)	DD0D (2)	AD0D (6)	DD0D (2)	DA0D (2)	DD0D (2)	DD1D (2)	DD0D (2)	DD0A (9)
1000 observe	AD0D (6)	DD0D (2)		AD0D (6)	AA0D (16)	AD0D (6)	AD1D (6)	AD0D (6)	
1100 observe	AA0D (16)			AD0D (6)		AA0D (16)	AA1D (17)	AA0D (16)	
1110 observe	AA1D (17)					AA0D (16)		AA1D (17)	
1111 observe	AA1A (24)							AA1D (17)	

2. Perform initialization
   (a) Shift in 000000000000000000000000 (length = 24)
3. Insert test sequence
   (a) Shift in 01 (length = 2)
   (b) Shift in 11 (length = 2)
4. Check test sequence while applying new configuration
   (a) Shift in 01 and observe the output; update ($\text{TDR}_1 \rightarrow \text{SIB}_2 \rightarrow \text{SIB}_1 \rightarrow \text{SIB}_3$)
5. Perform initialization
   (a) Shift in 000000000000000000000000 (length = 24)
6. Insert test sequence
   (a) Shift in 010101 (length = 6)
   (b) Shift in 11 (length = 2)
7. Check test sequence while applying new configuration
   (a) Shift in 011XXX and observe the output; update ($\text{TDR}_1 \rightarrow \text{TDR}_2 \rightarrow \text{TDR}_4 \rightarrow \text{SM} \rightarrow \text{SIB}_2 \rightarrow \text{SIB}_1 \rightarrow \text{SIB}_3$)
8. Perform initialization
   (a) Shift in 000000000000000000000000 (length = 24)
9. Insert test sequence

   (a) Shift in 0101010101010101 (length = 16)
   (b) Shift in 11 (length = 2)

10. Check test sequence while applying new configuration
    (a) Shift in 0111XXXXXXXXXXXX and observe the output; update ($TDR_1 \rightarrow TDR_2 \rightarrow TDR_5 \rightarrow SM \rightarrow SIB_2 \rightarrow SIB_1 \rightarrow SIB_3$)
11. Perform initialization
    (a) Shift in 000000000000000000000000 (length = 24)
12. Insert test sequence
    (a) Shift in 01010101010101010 (length = 17)
    (b) Shift in 11 (length = 2)
13. Check test sequence while applying new configuration
    (a) Shift in 1111XXXXXXXXXXXXX and observe the output; update ($TDR_1 \rightarrow TDR_2 \rightarrow TDR_5 \rightarrow SM \rightarrow SIB_2 \rightarrow SIB_1 \rightarrow TDR_3 \rightarrow SIB_3$)
14. Perform initialization
    (a) Shift in 000000000000000000000000 (length = 24)
15. Insert test sequence
    (a) Shift in 010101010101010101010101 (length = 24)
    (b) Shift in 11 (length = 2)
16. Last check of the test sequence
    (a) Shift in a sequence of longest path length (24)

2.3 A Semi-Formal Test Generation Technique for Reconfigurable Scan Networks

In the proposed approach, the RSN of IEEE 1687 is modelled as a finite state automaton (FSA). Each state corresponds to a configuration, that is, a determinate state of SIBs and SMs in the network; the input alphabet corresponds to reconfiguration operations; the output symbols are the lengths of the network, as this is an easily observable characteristic [9]. The high-level model is deliberately not complete, that is, the FSA's states encode only a subset of the possible configurations. As not all transitions are possible in all states, either due to the physical configuration of the RSN or to missing states in the FSA, whether an input does not correspond to a transition, the FSA is brought to a special sink state with no output transitions and a null output symbol.

Faults taken into consideration are high-level stuck-at faults affecting SIBs and SMs. Such faults are mapped to multiple transition fault on the high-level automaton, as the same configuration operations may result in different network statuses on faulty circuits, and the goal of the automatic test program generation is to devise a sequence of inputs able to discriminate between the faulty automata and the good one.

The proposed algorithm is based on a greedy search. While the simulation of the automaton is exact, the method is approximate because it does not consider all possible states nor all possible input symbols, and, consequently, not all possible transitions. Nevertheless, the approximation is conservative with respect to testability, as any missing state or transition will cause the automaton to reach the *sink state*, that by construction cannot be further distinguished from any other state.

The complexity of the proposed approach is linear on the number of states n_s times the size of the input alphabet A_{in}, that is $\mathcal{O}(n_s \cdot \|A_{in}\|)$. As both terms depend linearly on the number of configuration bits n_{cb}, the complexity is definitely smaller than the A* algorithm presented in [26], where the search space was $\mathcal{O}(2^{n_{cb}})$.

2.3.1 Network representation: FSA

The FSA is built incrementally. The FSA is initially composed of only of a state with no output transition and a *null* output symbol. Such *sink state* can not be distinguished from any other state, and, once entered, the FSA is not able to leave it. It is used to denote a pathological condition, where the algorithm is not able to provide reliable results due to the approximation of the model. Next, the *reset state*, when all configuration bits are set to zero, is added to the automaton. Then, for each SIB$_i$, two states are created: one with the SIB asserted and one with the SIB de-asserted. For each SM, one state is created for each possible output configuration. Such a straightforward approach, however, is not always sufficient. Scan chains may be nested, and a resource accessible only when its parent SIB is asserted. The procedure for building the FSA detects such situations, and creates the necessary states to handle them. The transitions from the *reset state* to all these states are eventually added.

Then, for each transition in the good automaton, the possible faulty transition are added, and whether the faulty transition would bring the automaton in a configuration not already encoded as a state, that specific state is added to the FSA. All missing transitions between existing states are also added to the automaton. Eventually, all possible faulty transition from all existing states are also added, but if one would bring the automaton in a configuration not encoded as a state, its destination is set to the sink state, meaning that the FSA is unable to model such situation. Such situations are related to ScanMux modules with two or more configuration bits and the number of inputs smaller than $2^{n_{cb}}$. Non-existent inputs may be grounded or bypassed to some other input. However, such details are related to hardware implementation and are not available in the ICL descriptions.

Some heuristics are considered in order to match configurations which may reduce the cost. For example, states representing configurations in which accessible SIBs that provide access to the deepest hierarchical level are not asserted may increase the number of required sessions and therefore the cost. Additionally, configurations in which a SIB is still asserted while already being fully tested together with its sub-hierarchical modules increase the cost. Not setting minimal path length configuration of a ScanMuxes that is fully tested together with its sub-hierarchical modules increases the cost.

As almost only the states with a hamming distance of 1 from the reset state are added to the FSA, the size of the automaton is linear in the number of configuration bits. It is possible to define an automaton with more states: for instance, at some point of the creation, all complementary states may be added as well; or all states at a hamming distance of 2 from the reset state can be considered. It is important to remember that the size of the automaton influences both the quality of the results and the performance of the algorithm. Experimental evaluations indicate that such extensions are not quite beneficial, but the designers may explicitly add relevant states to this state or provide an additional heuristic.

Table 2.2: Test procedure for the network in Fig.1.6

Input	Fault free	SIB_1		SIB_2		SM		SIB_3	
	$\bar{s}$	s@D-s^0	s@A-s^1	s@D-s^2	s@A-s^3	s@0-s^4	s@1-s^5	s@D-s^6	s@A-s^7
reset **observe**	DD0D (2)	DD0D (2)	AD0D (6)	DD0D (2)	DA0D (2)	DD0D (2)	DD1D (2)	DD0D (2)	DD0A (9)
1001 **observe**	AD0A (13)	DD0A (9)		AD0A (13)	AA0A (23)	AD0A (13)	AD1A (13)	AD0D (6)	
1100 **observe**	AA0D (16)			AD0D (6)		AA0D (16)	AA0D (17)		
1110 **observe**	AA1D (17)					AA0D (16)			

2.3.2 Greedy search algorithm

The search algorithm builds a test sequence as a sequence of *transition* and *observation* steps. During a *transition*, a sequence of bits is fed into the scan chain, bringing the RSN in a given configuration; such operation corresponds to one or more input symbols in the FSA. During an *observation*, the length of the scan chain is measured; the operation does not affect the FSA.

In more practical terms, the goal of the test generation is to find a short and effective sequence that brings the good circuit and the faulty ones in states where the scan chain is of different lengths; then, to observe the length and detect the faults. Indeed, not all transitions and not all observations require the same number of clock cycles to be performed. The search algorithm aims at minimizing the length of the test sequence with respect to the number of actual clock cycles required to execute all transitions and observations.

Let x be an input symbol for the FSA. The reset operation is denoted with **reset**, and it requires a single clock cycle to be performed. A sequence $\mathbf{t}$ of inputs $\mathbf{t} = (\mathbf{reset}, x_0, x_1, ..., x_i)$ unequivocally defines the state of the FSA. Let $\bar{s}_{\mathbf{t}}$ be the state of the FSA representing the fault-free circuit after the application of the input sequence $\mathbf{t}$, and let $\mathbf{S_t} = \{s^0_{\mathbf{t}}, s^1_{\mathbf{t}}, ..., s^n_{\mathbf{t}}\}$ be the set of the states of the FSA representing the n faulty circuits. $\mathbf{S_t}$ depends on the full sequence $\mathbf{t}$, and some faulty circuits may be in the correct state, thus $\bar{s}_{\mathbf{t}} \in \mathbf{S_t}$. It is possible that a fault also effects the reset state, while such a possibility is easily tractable by the proposed methodology, it was not considered in this work.

Let $\mathrm{DF}(\bar{s}, \mathbf{F})$ be the set of potentially detectable faults when the good machine is in state $\bar{s}$ and the faulty ones in $\mathbf{F} = (s^0, s^1, ..., s^f)$, that is, the set of all faults that caused the faulty machine to be in a state s^i with an output symbol different from $\bar{s}$. If an observation is performed, measuring the actual length of the RSN, any difference would be observed and all such faults, detected.

Given a sequence of inputs $\mathbf{t}$, the function GREEDY extends it with the most promising input symbol (Algorithm 1). That is, it appends the input symbol that brings the FSA where the highest number of faults could be detected. If no new fault can be detected by adding a single transition, the function returns an empty input sequence.

Algorithm 1 Greedy step

```
function GREEDY($\mathbf{t}$)
    $\mathbf{m} \leftarrow (\ )$                                    ▷ Empty sequence of inputs
    for $x \in$ {valid input symbols in $\bar{s}_{\mathbf{t}}$} do
        $\mathbf{u} \leftarrow \mathbf{t}$
        Append $x$ to $\mathbf{u}$
        if $|\mathrm{DF}(\bar{s}_{\mathbf{u}}, \mathbf{S_u})| > |\mathrm{DF}(\bar{s}_{\mathbf{m}}, \mathbf{S_m})|$ then
            $\mathbf{m} \leftarrow \mathbf{u}$
    return $\mathbf{m}$                                         ▷ Most promising sequence
```

The search algorithm incrementally builds the test sequence $\mathbf{t}$ calling the function GREEDY iteratively (Algorithm 2). In every step, the most useful symbol is appended to the test sequence; however, if it is not possible to detect new faults by adding a single symbol, the FSA is rolled back to a previous state where a useful input symbol may be found.

Algorithm 2 Test Sequence Generation

```
procedure TPG
    $\mathbf{t} \leftarrow (\mathbf{reset})$                          ▷ Initial test sequence
    $\mathbf{H} \leftarrow \{\mathbf{t}\}$                              ▷ History
    $\mathbf{F} \leftarrow \{$all detectable faults$\}$                 ▷ Active faults
    while $|\mathbf{F}| \neq 0$ do
        $\mathbf{g} \leftarrow$ Greedy($\mathbf{t}$)
        if empty($\mathbf{g}$) then                                     ▷ The greedy failed
            Append reset to $\mathbf{t}$                                ▷ Start over
            for $\mathbf{t}' \in \mathbf{H}$ do
                $\mathbf{g}' \leftarrow$ Greedy($\mathbf{t}'$)
                if $|\mathrm{DF}(\bar{s}_{\mathbf{g}'}, \mathbf{S}_{\mathbf{g}'})| > |\mathrm{DF}(\bar{s}_{\mathbf{g}}, \mathbf{S}_{\mathbf{g}})|$ then
                    $\mathbf{g} \leftarrow \mathbf{g}'$                 ▷ Alternative sequence
        Append $\mathbf{g}$ to $\mathbf{t}$
        Append observe to $\mathbf{t}$
        $\mathbf{H} \leftarrow \mathbf{H} \cup \{\mathbf{g}\}$          ▷ Save sequence
        Remove $\mathrm{DF}(\bar{s}_{\mathbf{g}}, \mathbf{S}_{\mathbf{g}})$ from $\mathbf{F}$
```

The symbol **observe** is used to denote an observation operation in the test sequence, it has no effect on the FSA, but its cost in term of clock cycles needs to be considered.

To provide an example, RSN from Fig. 1.6 is used. In the initial, *reset state*, original fault-free network has active path length 2. This is also the case with some other faults, except stuck-at-asserted faults on SIB_1 and SIB_3 that increase the path length to 6, i.e., 9. By inserting the test shift sequence these two faults can be detected. From state DD0D that is seen as "DXXD", since the two modules' configuration bits are not reachable (SIB_2 and SM), possible transitions that are considered and generated by the algorithm are AD0D, AD0A, and DD0A (by applying 1000, 1001, and 0001). Since the second one brings us to states where more faults can be detected, this one is applied. In this iteration s@D on SIB_1, s@A on SIB_2 and s@D on SIB_3 are detected. Algorithm continues to explore possible transitions: from state AD0A that is now seen as "ADXA", since 3 configuration bits are modifiable, possible transitions are AD0D, AA0D, AA0A (1000, 1100, 1101). The one detecting most faults (and having the lowest cost) is AA0D; it detects s@D on SIB_2 and s@1 on SM. Fianlly, AA1D (1110) detects s@0 on SM.

2.4 Enhanced version

In Section 2.3, an RSN was modelled as a Finite State Automaton (FSA), and a semi-formal method was described. Such approach is able to deal with larger and more complex circuits producing a test sequence able to detect any permanent fault affecting the reconfigurable modules, but whose duration is lower than the one of the test sequences previously generated by the heuristic solutions.

In this section, an extension of that approach is proposed. The test generator has been rewritten. The new algorithm minimizes the number of costly operations, postponing and compacting them, while it still guarantees to reach complete fault coverage. At the same time, the new algorithm is almost always faster than its predecessor. Experimental results on a set of benchmarks [50] demonstrate that the approach is able to generate test sequences orders of magnitude shorter than those reported in [51], [25] and [27], while always keeping the computational cost under control.

2.4.1 Search algorithm

Let x be an input symbol for the FSA. The reset operation is denoted with **reset**, and it requires a single clock cycle to be performed; the measurement of the length of the scan chain is denoted with **observe**, it requires several clock cycles and does not affect the state of the FSA. Both appending a symbol to an input sequence and concatenating two sequences are expressed as additions, as no ambiguities are possible. The symbol $\varnothing$ denotes an empty input and has no effect on a sequence, e.g., $\mathbf{t} = \mathbf{t} + \varnothing$. A sequence $\mathbf{t}$ of inputs starting with a **reset**, i.e., $\mathbf{t} = (\mathbf{reset}, i_0, i_1, ..., i_i)$, unequivocally defines the state of the FSA.

Two states that have indistinguishable output symbols are equivalent and are denoted with $s' \cong s''$. Conversely, non equivalent states have distinguishable output symbols and are denoted with $s' \ncong s''$. By definition, the sink state is equivalent to any other states $\forall s : s \cong \Omega$.

Let $\bar{s}_\mathbf{t}$ be the state of the FSA representing the fault-free circuit after the application of the input sequence $\mathbf{t}$, while $s^i_\mathbf{t}$ be the state of the FSA representing the circuit when fault i is present after the application of the same input sequence. If their output symbols are distinguishable, that is, $\bar{s}_\mathbf{t} \ncong s^i_\mathbf{t}$, then an additional **observe** input symbol would allow to mark the fault i as detected, and the fault is said to be *active*. The number of active faults may increase as well as decrease at each step of the input sequence.

Let $\mathcal{D}(\mathbf{t})$ be the set of all faults detected by the sequence $\mathbf{t}$. Indeed, $\mathcal{D}(\mathbf{t}) = \varnothing$ if $\mathbf{t}$ contains no **observe** symbols; and all input symbols after the last **observe** do not alter the results. Let $\mathcal{D}^*(\mathbf{t})$ be the set of all faults *potentially detected* by the sequence $\mathbf{t}$, that is, all faults either already detected or active after the application of the sequence $\mathbf{t}$, that is, the set of all faults that would be detected by appending an **observe** to the input sequence: $\mathcal{D}^*(\mathbf{t}) = \mathcal{D}(\mathbf{t} + \mathbf{observe})$.

The search algorithm incrementally builds a test sequence through a greedy search. Explicit observations, that is **observe** symbols, are not included in the test sequence unless required. The function Greedy returns the most useful input symbol to be added (Algorithm 3), neglecting observations: given an input sequence $\mathbf{t}$, it identifies the symbol s that maximizes $|\mathcal{D}^*(\mathbf{t}+s)|$. If adding a single symbol cannot activate any new fault, the function returns an empty symbol.

Algorithm 3 Identify most useful input symbol, neglecting observations.

```
function GREEDY($\mathbf{t}$)
    $best \leftarrow \varnothing$                                          ▷ Empty symbol
    for $x \in \{$valid input symbols in $\bar{s}_\mathbf{t}\}$ do
        if $|\mathcal{D}^*(\mathbf{t}+x)| > |\mathcal{D}^*(\mathbf{t}+best)|$ then
            $best \leftarrow x$
    return $best$                                                ▷ Most useful symbol
```

The search algorithm incrementally builds the test sequence $\mathbf{t}$ calling the function Greedy iteratively (Algorithm 4). In every step, the most useful symbol is appended to the test sequence, trying to increase the number of active faults. Only when a new symbol s would cause the loss of a previously activated fault, an **observe** symbol is inserted before s.

When it is not possible to activate new faults by adding a single symbol, an **observe** symbol is appended and the FSA is rolled back to a previous state where useful input symbols may still be found and the search restarted. Such a state is chosen among the previously traversed ones, and it is the closest one in term of configuration clock cycles. The procedure terminates when all detectable faults have been detected.

Algorithm 4 Test Sequence Generation

```
procedure TPG
    $\mathbf{t} \leftarrow (\mathbf{reset})$                                   ▷ Initial test sequence
    $\mathbf{H} \leftarrow \{\mathbf{t}\}$                                         ▷ History
    while $\mathcal{D}^*(\mathbf{t}) \neq \{$all detectable faults$\}$ do
        $s \leftarrow$ Greedy($\mathbf{t}$)
        if $s \neq \varnothing$ then                                  ▷ The greedy succeeded
            if $\mathcal{D}^*(\mathbf{t}) \nsubseteq \mathcal{D}^*(\mathbf{t}+s)$ then
                $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{observe}$                          ▷ Required
            $\mathbf{t} \leftarrow \mathbf{t} + s$                                        ▷ Add symbol
            $\mathbf{H} \leftarrow \mathbf{H} \cup \{\mathbf{t}\}$                                 ▷ Save sequence
        else                                                  ▷ The greedy failed
            $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{observe}$
            $\mathbf{r} \leftarrow$ shortest($\{\mathbf{a} \in \mathbf{H} : $ Greedy$(\mathbf{t}+\mathbf{a}) \neq \varnothing\}$)
            $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{r}$                                         ▷ Start over
    $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{observe}$                                ▷ Final observation
```

To demonstrate the difference in the Test Sequence Generation procedure between this approach and [51], we can use the network shown in Fig. 1.6. Table 2.2 and Table 2.3 show the phases of test sequence generation using procedure described in Section 2.3 and the present one, respectively. The first column (*input*) shows input symbols that were chosen and applied. The second column refers to the fault-free network and its states. Columns 3-10 show the state of the faulty circuits, each one for one particular fault. As it can be seen from Table 2.2, in the previous approach each configuration phase is followed by an observation phase (marked in bold). In this way a set of active faults is added to the set of detected faults, which is increasing after each session. However, in Table 2.3 one can see that not every configuration phase is necessarily followed by an observation phase. Although the number of configuration steps is higher, the number of observation steps, which can be extremely costly, is lower. In total, the number of clock cycles needed to apply the test sequence given in Table 2.2 is 235 clock cycles, while on the other hand, applying the test sequence from Table 2.3 requires 189 clock cycles, only.

In comparison with [51], the length of the configuration vector may not be equal to the value of the output symbol of the fault-free circuit's current state, $\bar{s}_{\mathbf{t}}$. The length of the configuration vector is equal to $\max(\bar{s}_{\mathbf{t}}, s^i_{\mathbf{t}})$, $(\forall i)(i \in \{0,1,\dots,n-1\} \wedge$ (fault i not detected)), where n represents the total number of faults. The length of the configuration vector is included in the configuration cost. Positions of certain configuration bits in the chain that is defined by the state $s^i_{\mathbf{t}}$ or $s^j_{\mathbf{t}}$, $i \neq j$, may not correspond to any position of configuration bits in the chain determined by the state $\bar{s}_{\mathbf{t}}$. In this case, 0 bits are placed on these positions (Fig. 2.1). Additionally, on the same position (in chains defined by different states), one may find configuration bits belonging to different modules. This is all taken into account when assembling and then applying configuration vector corresponding to the chosen input symbol.

2.5 Evolutionary approach to test reconfigurable modules in RSNs

In this section the issue of generating effective sequences for testing the reconfigurable elements within RSNs is addressed using evolutionary computation. Test configurations are extracted with automatic test pattern generation (ATPG) and used to guide the evolution. Post-processing techniques are proposed to improve the evolutionary fittest solution. Results on a standard set of benchmark networks show up to 27% reduced test time with respect to test generation based on RSN exploration.

The approach proposed in this paper aims at generating an effective test sequence able to detect all testable faults while requiring a reduced test application

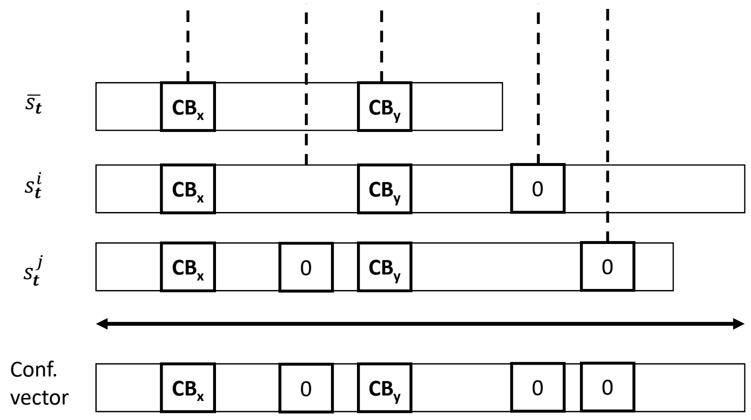


Figure 2.1: Configuration vector and scan chain bit positions for fault-free and faulty circuits

Table 2.3: Enhanced test procedure for the network in Fig.1.6

Input	Fault free	SIB$_1$		SIB$_2$		SM		SIB$_3$	
	$\bar{s}$	s@D-s^0	s@A-s^1	s@D-s^2	s@A-s^3	s@0-s^4	s@1-s^5	s@D-s^6	s@A-s^7
reset	DD0D (2)	DD0D (2)	AD0D (6)	DD0D (2)	DA0D (2)	DD0D (2)	DD1D (2)	DD0D (2)	DD0A (9)
1001	AD0A (13)	DD0A (9)	AD0D (6)	AD0A (13)	AA0A (23)	AD0A (13)	AD1A (13)	AD0D (6)	AD0A (13)
1101	AA0A (23)	DD0D (2)	AD0D (6)	AD0A (13)	DA1D (2)	AA0A (23)	AA1D (17)	AA0D (16)	AA0A (23)
1111 **observe**	AA1A (24)	DD0D (2)	AD0D (6)	DD0A (9)	DA1D (2)	AA0A (23)	AA1D (17)	AA1D (17)	AA1A (24)
0000 **observe**	DD0D (2)								DD0A (9)

time. In a first phase, an evolutionary algorithm [52] is used to cultivate a population of individuals representing a set of RSN configurations in which test vectors are applied. Then, the best individual produced in terms of test application time is further optimized by a post-processing algorithm, which tries to anticipate some test vectors and to remove redundant configurations.

In the following, the basic concepts needed to understand the proposed approach are briefly introduced (Section 2.5.1). Details are then given concerning: the algorithm used to perform a transition from a given configuration to a target one (Section 2.5.2), the evolutionary algorithm (Section 2.5.3), the encoding used for defining individuals (Section 2.5.4), and the post-processing algorithm (Section 2.5.5).

2.5.1 Methodology Basics

The proposed approach requires the following features:

1. A function (referred to as $Transition$) able to produce a sequence of configuration vectors cv_1, cv_2, ..., cv_n that moves the RSN from the generic configuration C_{src} to the configuration C_{dst}. The configuration vectors cv_1, cv_2, ..., cv_n are applied (i.e., shifted in the network through scan input pins for as many clock cycles as the active path length, and followed by an update operation), the first (i.e., cv_1) starting from C_{src} and passing through several intermediate configurations (i.e., C_1, C_2, ..., C_{n-1}) up to C_{dst}. This function can be associated to a cost in terms of clock cycles required to apply all the configuration vectors generated.

2. A function $Evaluation$ able to produce the list of faults that can be excited when the RSN is moved to the generic configuration C_i. Such faults would be covered by means of a test vector applied after reaching C_i. This function can be applied to a set of configurations; in such a case, $Evaluation(C_1, C_2, ..., C_n)$ produces the list of faults covered by all the configurations in the set: if a test vector tv_i is applied in each of the evaluated C_i, then all faults are covered.

By using an evolutionary engine which calls the $Transition$ and $Evaluation$ functions we aim at identifying a sequence of configurations detecting all faults and having minimum cost. Each configuration is associated to a test session. Each test session is composed by applying the $Transition$ function to generate intermediate configuration vectors which move the network from the configuration C_i in the list to C_{i+1}. The first time, the function is applied between the reset configuration C_{rst} and the first configuration in the list (if not equal to C_{rst}). A test vector is applied to the RSN after each transition to a configuration in the list. Thus, the $Evaluation$ function is applied to the list of configurations and the faults obtained are used to compute the fault coverage. Moreover, the total test time is obtained as the cost to apply the configuration vectors generated by the $Transition$ functions plus the time required to shift all test vectors.

As an example, let us consider the RSN in Fig. 1.6. For this network, let us suppose the reset configuration is the one indicated with C_0 in Table 1.1. A possible solution to the problem of testing the network faults consists in the following sequence of configurations: $[C_0, C_8, C_{13}, C_{14}]$. For each configuration C_i a test vector tv_i is applied, which is made as follows:

1. as many 0s as the longest path length, i.e., 24 bits in the example network;
2. an alternated sequence 0101..., as long as the length of the active path currently selected;
3. two consecutive 1s (or two consecutive 0s) as the sequence terminator;
4. only for the last test vector, a sequence as long as the length of the active path currently selected (values being shifted in are not important).

As highlighted in the list of faults excited by each configuration (see Table 2.4), the configurations $[C_0, C_8, C_{13}, C_{14}]$ allow detecting all faults in the network. The list of vectors corresponding to such list of configurations is composed as follows:

1. tv_1 in C_0 (shift of 24+2+2 bits)
2. cv_1 from C_0 to C_8 (shift of 2 bits, then update)
3. tv_2 in C_8 (shift of 24+6+2 bits)
4. cv_2 from C_8 to C_{13} (shift of 6 bits, then update)
5. tv_3 in C_{13}(shift of 24+17+2 bits)
6. cv_3 from C_{13} to C_{14} (shift of 17 bits, then update)
7. tv_4 in C_{14}(shift of 24+23+2+23 bits)

If a cost of 5 clock cycles is considered to move the TAP controller from shift to update and vice-versa (also including the first shift after the network reset), the above test sequence is executed in 235 clock cycles.

The order in which configurations appear in the list is important and results in different vectors generated by the $Transition$ function. For example, let us consider the same set of configurations as in the previous example but listed in a different order: $[C_0, C_{13}, C_{14}, C_8]$. In this case, the list of vectors composing the test sequence is the following:

1. tv_1 in C_0 (shift of 24+2+2 bits)
2. cv_1 from C_0 to C_8 (shift of 2 bits, then update)
3. cv_2 from C_8 to C_{12} (shift of 6 bits, then update)
4. cv_3 from C_{12} to C_{13} (shift of 16 bits, then update)
5. tv_3 in C_{13}(shift of 24+17+2 bits)

Table 2.4: List of faults excited by the RSN in Fig. 1.6.

Configuration	Set of covered faults
C_0 C_1 C_4 C_5	SIB_1-s@A, SIB_3-s@A
C_2 C_3 C_6 C_7	SIB_1-s@A, SIB_3-s@D
C_8 C_9	SIB_1-s@D, SIB_3-s@A, SIB_2-s@A
C_{10} C_{11}	SIB_1-s@D, SIB_3-s@D, SIB_2-s@A
C_{12}	SIB_1-s@D, SIB_2-s@D, SM_1-s@1, SIB_3-s@A
C_{13}	SIB_1-s@D, SIB_2-s@D, SM_1-s@0, SIB_3-s@A
C_{14}	SIB_1-s@D, SIB_2-s@D, SM_1-s@1, SIB_3-s@D
C_{15}	SIB_1-s@D, SIB_2-s@D, SM_1-s@0, SIB_3-s@D

6. cv_4 from C_{13} to C_{14} (shift of 17 bits, then update)

7. tv_4 in C_{14} (shift of 24+23+2 bits).

8. cv_5 from C_{14} to C_8 (shift of 23 bits, then update)

9. tv_5 in C_8 (shift of 23+6+2+6 bits).

The above test sequence has the same fault coverage of the previous example but is longer to execute (266 clock cycles). Moreover, it can be noticed that the configuration C_8 is visited twice before applying a test vector (tv_5).

2.5.2 Transition function

The TRANSITION function computes the sequence of configuration vectors able to move the network state from the starting configuration to a target one with minimal configuration cost (Algorithm 5).

Algorithm 5 Transition function

```
function TRANSITION($C_{src}, C_{dst}$)
    $\mathbf{p} \leftarrow (\,)$                                        ▷ Empty sequence of inputs
    $C_{next} \leftarrow C_{src}$
    $hasNext \leftarrow$ true
    while $next$ do
        $hasNext \leftarrow$ CONFIGUREBRANCH($C_{src}, C_{next}, C_{dst}, 0, confBits$)
        if $hasNext$ then
            Append $C_{next}$ to $\mathbf{p}$
    return $\mathbf{p}$
```

The CONFIGUREBRANCH function composes the portion of the next state that is required to configure each multiplexer in the current branch towards the target state (Algorithm 6). First, the total number of steps required to configure a multiplexer is calculated taking into account the target state of sub-hierarchical multiplexers it controls. Then, the maximum number of steps for all multiplexers in a given branch is set as a number of steps required to configure that branch. Subsequently, all multiplexers that require the highest number of configuration steps are immediately configured to match the target state. Conversely, the ones that do not, are configured to match the minimal possible length configuration starting from the higher hierarchical levels so that the previously calculated number of required configurations is not affected.

Algorithm 6 Configuring the branch for next configuration

```
function CONFIGUREBRANCH($C_{src}, C_{next}, C_{dst}, start, end$)
    $branchSteps \leftarrow$ BRANCHCONFIGSTEPS          ▷ num. of steps for conf. branch
    $i \leftarrow start$                                          ▷ scanning the branch
    while $i < end$ do
        configureBranch
        $Mux \leftarrow$ {multiplexer controlled by $i$ configuration bit}
        if $Mux$ not accessible then
            continue
        $muxSteps \leftarrow$ MUXCONFIGSTEPS                ▷ num. of steps for conf. mux
        if $branchSteps > 1$ and $muxSteps < branchSteps$ then
            $hasNext | =$ MINIMIZEMUX($C_{src}, C_{next}, C_{dst}, Mux$)      ▷ some state var.
on mux branches need to be conf.
        else if $muxSteps = branchSteps$ then
            $hasNext | =$ CONFIGUREMUX($C_{src}, C_{next}, C_{dst}, Mux$)
        $i \leftarrow$ next top level multiplexer
    return $hasNext$
```

The function CONFIGUREMUX composes the portion of next state needed to configure the given multiplexer toward the target state. First, it recursively composes the next state configuration for the selected branch of the multiplexer. Then it composes the next state that the multiplexer itself must assume, selecting the shortest branch that still needs to be configured. The function returns true if some state variables in the multiplexer branches still need to be configured, false otherwise. When multiple branches of the multiplexer have to be configured, these are configured in the order of their current scan path length in order to minimize the cost of switching between these branches.

The function MINIMIZEMUX composes the portion of the next state needed to configure the given multiplexer toward its minimal length configuration. However, often imposing the minimum length configuration on the multiplexer may result in changing (increasing) the maximum number of steps required to reach the target state. Therefore, first it calculates which branch should be configured to minimize the multiplexer length. Then it configures the multiplexer and its branches to match the target configuration. If the new selection of the branch corresponds to the minimum length branch, the length of that branch is minimized. Otherwise, the branch with the minimum length is selected. The function returns true if some of the state variables on the multiplexer branches still need be configured, false otherwise.

2.5.3 Evolutionary algorithm

The proposed approach exploits an evolutionary meta-heuristic to identify a test sequence which minimizes the test cost while guarantying the full test coverage. A population of *individuals* is cultivated by the *evolutionary engine*. An individual, $\{C_{t0}, C_{t1}, C_{t2}, ..., C_{tk-1}\}$, is represented as a variable-length sequence of valid configurations.

Individuals are evaluated by a separated *evaluation engine* that provides the evolutionary engine with the *fitness values* of each individual. In more details, the evaluation engine:

1. applies the $Evaluation$ function to the list of configurations and computes the fault coverage;

2. generates the test sequence, composed of configuration vectors obtained by applying the $Transition$ function between consecutive configurations in the list, and test vectors $\{C_{t0}, \{C_{0i}\}, C_{t1}, \{C_{1i}\}, C_{t2}, \{C_{2i}\}, ..., C_{tk-1}\}$; then, it computes the cost in terms of time (number of clock cycles) needed to execute the latter sequence.

The evolutionary framework is given in Fig. 2.2. In the proposed flow, the fitness of an individual is composed of two components: the fault coverage and the inverse of the test cost. These components are considered lexicographically: if the fault coverage is higher, the fitness is higher, independently from the test costs.

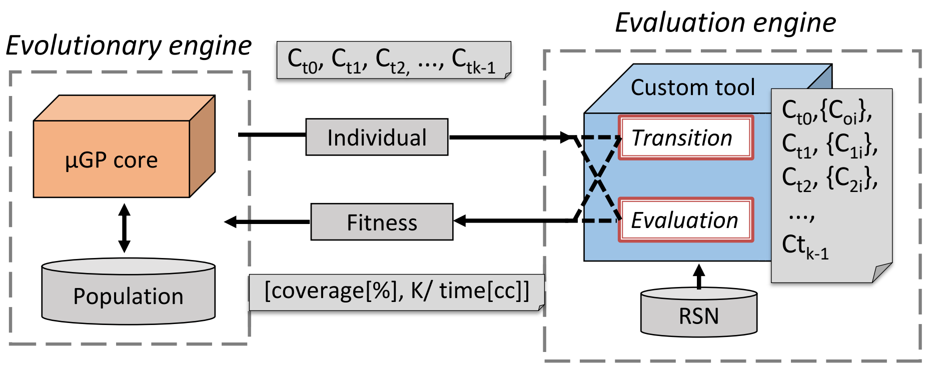

Evolutionary engine
µGP core
Population
$C_{t0}, C_{t1}, C_{t2}, \ldots, C_{tk-1}$
Individual
Fitness
[coverage[%], K/ time[cc]]
Evaluation engine
Custom tool
Transition
Evaluation
RSN
$C_{t0}, \{C_{0i}\}, C_{t1}, \{C_{1i}\}, C_{t2}, \{C_{2i}\}, \ldots, Ct_{k-1}$

Figure 2.2: Evolutionary framework.

At the beginning of the evolution, a population of n_p random individuals is generated. Then, in each step, called *generation*, the population is first expanded, then shrunk back to its original size.

During the expansion, n_o genetic operators are activated and the generated offspring is added to the population, in a *steady-state* approach. Genetic operators include the standard mutation operators, that generate a new candidate solution by slightly modifying an existing one, and crossover operators, that generate a new candidate solution by recombining two existing solutions. Then the evaluation engine is used to assess the fitness of all the new individuals. Finally, the least fit individuals are discarded, shrinking the size of the population down to the original n_p.

The process is iterated until a steady state is detected. That is, the fittest individual in the population does not change for a given number of generations. Such a condition intuitively indicates that a local optimum has been reached.

Alternatively, some individuals able to cover all faults can be directly inserted in the initial population. This technique, called *seeding*, is likely to speed up the evolutionary process: the optimizer is only asked to reduce the cost and not to saturate the fault coverage first. However, the offspring of these few initial individuals could take over the entire population quickly, bringing the algorithm into a local optimum. The experimental analyses suggest using seeding only when it is particularly hard to reach the full test coverage of the considered RSN.

2.5.4 Individual encoding

Each individual created by the evolutionary engine consists in a sequence of configurations. Since a configuration is determined by values in the selection bits of each reconfigurable element in the RSN, it can be represented by a bit-string. Individuals are thus files composed of multiple bit-strings.

The evolutionary engine creates individuals which are structured as described in a constraint library. The constraint library is also saved in a file and contains one or more *macros*, each one defining a possible mapping of a bit-string in the individual. In other words, in order for an individual to be considered as valid by the evolutionary engine, each of its lines must match one of the macros in the constraint library.

In the problem in hand, a macro describes which parts of a bit-sting are fixed to predefined values and others which can be freely modified by the evolutionary engine. As an example, if the RSN in Fig. 1.6 is considered, a possible macro in the constraint library can be "D−−D", which is satisfied by all configurations in Table 1.1 having SIB_1 and SIB_3 de-asserted ("−" means don't care). If a macro composed of all don't care bits is included in the constraint library, then the evolutionary engine is allowed to define completely random configurations. Such macro will be referred to as the *random macro*.

In the proposed methodology, other than the random macro, constrained configurations are extracted using automatic test patterns generation (ATPG) on a combinational circuit that represents the problem and converted to macros. The circuit is graphically described in Fig. 2.3 and receives as input the following values:

1. as many bits as the number of configuration bits in the RSN (*conf* in the figure);

2. as many bits as the number of functional faults in the RSN (*faults* in the figure).

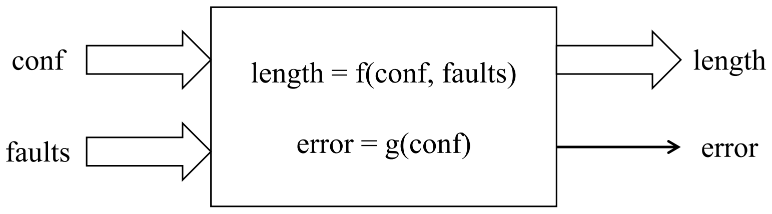


Figure 2.3: Combinational circuit used for ATPG.

If one of the input signals of *faults* is set to 1, then the corresponding fault (e.g., SIB_1-stuck-at-asserted) is activated.

As output, the circuit produces the following values:

1. the active path length (*length* in the figure) in the configuration *conf*, when one or more faults are active (i.e., one or more bits of *faults* are set to 1);

2. a bit (*error* in the figure) that alerts when an illegal configuration is used as the *conf* value.

The combinational circuit can be written in behavioural VHDL or Verilog by encoding the truth-table of the active path length function (e.g., as in Table 1.1). However, such an approach becomes easily unfeasible due to a high number of configuration bits or when the RSN is designed using certain patterns (e.g., several sibling SIBs). The approach we suggest is to build the circuit incrementally while traversing the RSN hierarchy. The final length can be expressed as a sum of different contributes associated to TDRs, SIBs, and ScanMuxes. As an example, the final length of the RSN in Fig. 1.6 is the sum of the lengths associated to the sub-networks controlled by SIB_1 and SIB_3, respectively. The pseudo-code of the functions LENGTH and ERROR for the example RSN is reported in Algorithm 7.

In order for the behavioral circuit to be ATPG ready, it is then translated in structural Verilog by means of logic synthesis. The ATPG process consists in the following steps:

1. in order to activate faults internally, the *faults* input signals are constrained to the value 0;
2. in order to generate only valid configurations, the *error* output signal is constrained to the value 0;
3. the ATPG fault list includes stuck-at-1 faults on the *faults* input signals, only;
4. X values are used as don't care bits in the patterns list.

After performing the ATPG, patterns are saved into a text file and translated into macros and included in the constraint library, such that the evolutionary engine can freely modify don't care bits while fixing the other bits to the values reported in the corresponding pattern.

Alternative encoding

A suitable test vector is shifted-in after reaching each configuration. The $Transition$ function interconnects the configurations in the list, eventually adding intermediate configurations where tests are not performed. Therefore, configuration patterns to reach the configuration C_j from C_i are decided by $Transition(C_i, C_j)$, hence also intermediate configurations. Since the purpose of the proposed approach is the minimization of the test time, the $Transition$ function should be able to compute the minimum cost path from C_i to C_j. Alternatively, if a sub-optimal $Transition$ function is available, we propose to slightly modify the structure of the individuals generated by the evolutionary engine.

Algorithm 7 Combinational circuit functions for the RSN in Fig. 1.6

```
function LENGTH($conf, faults$)
    if SIB$_2$ is de-asserted  or SIB$_2$-s@D then
        $lengthSIB_2 \leftarrow 1$
    else if SIB$_2$ is asserted  or SIB$_2$-s@A then
        $lengthSIB_2 \leftarrow 1 + 7$
    else
        $lengthSIB_2 \leftarrow 0$                                ▷ unexpected case
    if SM selects 0  or SM-s@0 then
        $lengthSM \leftarrow 3$
    else if SM selects 1  or SM-s@1 then
        $lengthSM \leftarrow 6$
    else
        $lengthSM \leftarrow 0$                                   ▷ unexpected case
    if SIB$_1$ is de-asserted  or SIB$_1$-s@D then
        $lengthSIB_1 \leftarrow 1$
    else if SIB$_1$ is asserted  or SIB$_1$-s@A then
        $lengthSIB_1 \leftarrow 1 + 2 + lengthSIB_2 + lengthSM + 1$
    else
        $lengthSIB_1 \leftarrow 0$                                ▷ unexpected case
    if SIB$_3$ is de-asserted  or SIB$_3$-s@D then
        $lengthSIB_3 \leftarrow 1$
    else if SIB$_3$ is asserted  or SIB$_3$-s@A then
        $lengthSIB_3 \leftarrow 1 + 4$
    else
        $lengthSIB_3 \leftarrow 0$                                ▷ unexpected case
     return $lengthSIB_1 + lengthSIB_3$
function ERROR(conf) return 0                            ▷ No illegal configurations
```

The alternative encoding consists in adding a flag to each configuration in the list to indicate whether a test vector should be applied in that configuration or not. An example of individual for the RSN of Fig. 1.6 is $[C_0t, C_8f, C_{12}t, C_{13}t]$, where t indicates that a test vector is applied after reaching that configuration, and f the opposite case. The example can be interpreted as the intention to force the network to pass through the configuration C_8, which becomes an intermediate configuration for the transition between C_0 and C_{12}. In details, it is like splitting $Transition(C_0, C_{12})$ into $Transition(C_0, C_8)$ and $Transition(C_8, C_{12})$. Clearly, the fault coverage is computed by applying the *Evaluation* function to the configurations that are marked with t, only. This is because faults excited by intermediate configurations are potentially excited but not explicitly observed.

Using the proposed modification, the problem of finding the best path to a configuration that requires a test vector is partially delegated to the evolutionary engine. Clearly, the problem becomes more complex compared to when an optimal *Transition* function is used; thus, the progression of the evolution becomes slower.

2.5.5 Post-processing techniques

Two post-processing techniques are proposed in order to reduce the test cost of the sequence generated resorting to the evolutionary algorithm described in Section 2.5.3. They can be applied on the provided test sequence independently, if necessary.

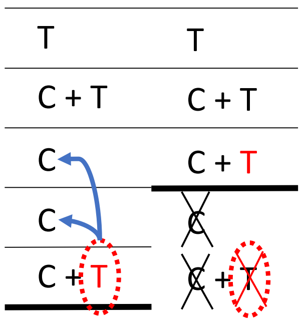


Figure 2.4: Post-processing I

The first one is used to process the full list of configurations in which test is performed and configurations that are exclusively used to interconnect the latter ones. The function reads the list in the reverse order (from end to beginning) and tries to advance the last test vector by appending it next to one of the preceding intermediate transition configurations (Algorithm 8); by doing so, all the configuration vectors required previously to reach the last test state from the penultimate one can be discarded including the last test vector (Fig. 2.4). Consequently, removing them, the number of clock cycles required to apply the generated test sequence is directly reduced. The condition for advancing such test vector is that the fault coverage has to remain unchanged while the test cost of the modified sequence should be reduced. If the last test vector is successfully anticipated, the algorithm continues checking the updated test sequence. This operation is performed until it becomes impossible to satisfy the condition and move forward currently last test vector in the modified test sequence.

The second technique is used to perform modifications on the test vector set (Algorithm 9). For each test vector in the list, a new (reduced by one) list is generated excluding that particular vector. The new list is generated by applying the

Algorithm 8 Bottom-up approach for moving the test states

```
function POSTPROC1($S$)
    $nextState \leftarrow$ true
    $U \leftarrow S$
    while $nextState$ do
        EVALUATE($U, faultC, costT$)          ▷ calculate fault coverage and test cost
        $minCost \leftarrow costT$
        $bestSeq \leftarrow U$
        $nextState \leftarrow$ false
        $U \leftarrow U\{$ remove last Test vector$\}$
        for $s_i \in \{U\}$ do
            if $s_i$ is Configuration then
                $H \leftarrow U\{$insert Test vector at $i$ position$\}$
                $H\{$remove excessive Configuration vectors$\}$
                EVALUATE($H, nfaultC, ncostT$)
                if $nfaultC = 100\%$ then                    ▷ check coverage
                    if $ncostT < minCost$ then                 ▷ check cost
                        $minCost \leftarrow ncostT$      ▷ update cost, save new sequence
                        $bestSeq \leftarrow H$
                        $nextState \leftarrow$ true
        $U \leftarrow bestSeq$
    return $U$
```

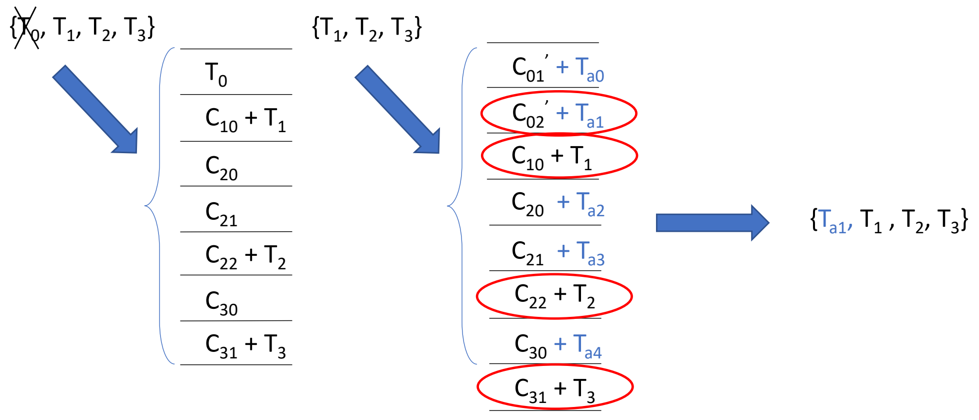


Figure 2.5: Post-processing II, first test vector removed

Transition function on all pairs of consecutive test vectors to insert interconnecting configuration vectors. All the configuration vectors are considered as potential candidates to be followed by a test vector, based on the set of faults they cover.

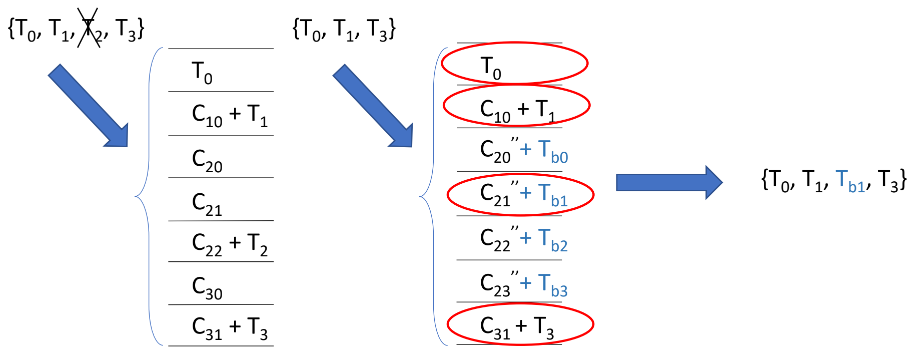


Figure 2.6: Post-processing II, third test vector removed

After traversing the whole list to find potential points of test vector insertion, the newly generated list is evaluated and recorded only if the fault coverage is unchanged (100%) while the test cost is reduced with respect to the one previously recorded. The process is repeated until no further improvement is possible for a given sequence of test vectors. The Fig. 2.5 and Fig. 2.6 show the algorithm flow and exemplify how removing different test vectors from the initial list results in having different interconnected lists and consequently different test sequences. The choice between the two is driven by the test cost, since the potential solutions with lower fault coverage are not even considered.

2.6 Experimental Results

2.6.1 Experiments for FSA approaches from Section 2.3 and Section 2.4

The effectiveness of the proposed algorithm has been evaluated on a sub-set of the ITC16 suite of benchmark reconfigurable scan networks. Some networks included in the benchmarks have not been considered since they include some constructs that are not currently supported by our environment. The algorithm proposed in this paper has been compared against three alternative approaches. The first approach, to which we refer to as FSA, has been proposed in [51]. The second approach is derived from [25] and is referred to as *depth-first* in this paper. The approach is based on the exploration of the network topology graph performing a depth-first traversal of this graph. The third approach has been proposed in [27] and is referred to as *evolutionary* in this paper. The approach makes use of an evolutionary framework to generate a test sequence possibly able to minimize the test time.

Experiments were run using a tool written in Java. The tool supports network

Algorithm 9 Removing test states and trying to insert new ones with reduced cost

```
function POSTPROC2($T$)
    $CT \leftarrow T$\{apply TRANSITION()\}    ▷ interconnect Test vect. with Conf. vect.
    EVALUATE($CT, faultC, costT$)    ▷ calculate fault coverage and test cost
    $minCost \leftarrow costT$
    $bestSeq \leftarrow T$
    $hasNext \leftarrow$ true
    while $hasNext$ do
        $hasNext \leftarrow$ false
        for $t_i \in \{bestSeq\}$ do
            $U \leftarrow bestSeq$\{remove $t_i$ test vector \}    ▷ remove one Test vector
            $FSet \leftarrow U$\{set of faults covered by set of Test vectors\}
            $TList \leftarrow U$\{apply TRANSITION()\}    ▷ add interconnecting Conf.
            $NTList \leftarrow$ ( )
            for $s_i \in TList$ do
                if $s_i$ is Configuration then    ▷ among Conf. vect. try to
                    if FAULTS($s_i$) $\setminus FSet \neq \emptyset$ then    ▷ insert Test vect. to increase
                        Append $s_i$ to $NTList$    ▷ fault coverage
                else
                    Append $s_i$ to $NTList$
            EVALUATE($NTList, nfaultC, ncostT$)    ▷ evaluate new Test vect. list
            if $nfaultC = 100\%$ then
                if $ncostT < minCost$ then    ▷ save the better solution
                    $minCost \leftarrow ncostT$
                    $bestSeq \leftarrow NTList$
                    $hasNext \leftarrow$ true
    return $bestSeq$
```

structure extraction from files in different formats including ICL. Moreover, the tool is able to distinguish all faults that are undetectable, due to the inability to produce any difference in the path length. For example, faults affecting SIB modules that do not have any register or any other module on their branch are considered to be undetectable. Additionally, faults affecting ScanMux modules that have registers of equal lengths on their branches are also considered as undetectable, again taking into account the faut model that was used in this approach. However, there is only a small number of undetectable faults in the set of benchmark networks that were used to evaluate the algorithm,for which we provide details in Table Table 2.5.

A laptop equipped with an Intel i5-480M processor was used to run experiments. Table 2.6 summarizes the experimental results. The table shows the number of configuration vectors *cv* (column 2) and test vectors *tv* (column 3) generated by

Table 2.5: List of undetectable faults

Network	Number	Comment
q12710	4	2 SIBs with the register length equal to 0
N132D4	4	2 ScanMuxes eq. branch registers (11, 18)
NE600P150	4	2 ScanMuxes eq. branch registers (45, 11)
NE1200P430	4	2 ScanMuxes eq. branch registers (115, 29)

the tool. Furthermore, the number of clock cycles required to configure the network is given in column 4, while the number of clock cycles needed to apply test vectors is given in column 5.

Table 2.6: IEEE 1687 test algorithm experimental results

Network	*cv*	*tv*	Conf. time [cc]	Test time [cc]
Mingle	6	7	628	811
TreeBal.	7	1	8,569	12,646
TreeFlat_Ex	6	3	7,750	16,267
TreeUnbal.	11	1	105,197	77,121
a586710	4	5	46,575	170,257
p22810	2	1	2,698	90,537
p34392	6	3	29,357	111,911
p93791	6	3	103,525	403,532
q12710	2	1	8,311	78,562
t512505	2	1	8,891	230,438
N132D4	7	2	9,387	7,682
N17D3	5	2	1,159	1,151
N32D6	5	2	230,390	282,236
N73D14	13	2	1,073,954	537,833
NE1200P430	128	2	1,638,849	200,258
NE600P150	79	2	347,629	55,098

Table 2.7: Experimental comparison of the enhanced algorithm (FSA_2) – Section 2.4 against the previous version [51] – Section 2.3, a Depth-first algorithm [25], and an Evolutionary approach [27]. Columns ending with "vs." show the comparison against the current result; percentages quantify how much the results delivered by the previous approaches are worse.

Network	Total test time [clock cycles]							Runtime (wall clock)			
	FSA2	[51]	[51] vs.	[25]	[25] vs.	[27]	[27] vs.	FSA2	[51]	[25]	[27]
Mingle	1,439	2,014	39.96%	2,282	58.58%	2,078	44.41%	49s	26s	1s	8h
TreeBalanced	21,215	63,843	200.93%	69,369	226.98%	69,369	226.98%	1m	48s	1s	19h
TreeFlat Ex	24,017	41,883	74.39%	71,341	197.04%	55,776	132.24%	1m	71s	1s	8h
TreeUnbalanced	182,318	719,375	294.57%	1,071,799	487.87%	1,042,450	471.78%	1m	34s	1s	5h
a586710	216,832	296,796	36.88%	299,624	38.18%	298,241	37.54%	30s	39s	1s	8h
p22810	93,235	152,399	63.46%	152,937	64.03%	152,937	64.03%	44s	39s	1s	9h
p34392	141,268	195,554	38.43%	196,702	39.24%	196,505	39.10%	36s	1m	1s	7h
p93791	507,057	706,242	39.28%	708,878	39.80%	708,878	39.80%	10m	2m	1s	27h
q12710	86,873	131,022	50.82%	131,022	50.82%	131,022	50.82%	39s	46s	1s	5h
t512505	239,329	385,440	61.05%	386,024	61.29%	386,024	61.29%	45s	50s	1s	8h
N132D4	17,069	31,995	87.45%	38,731	126.91%	37,257	118.27%	4m	2s	1s	3h
N17D3	2,310	3,765	62.99%	4,143	79.35%	3,851	66.71%	1s	1s	1s	5h
N32D6	512,626	816,634	59.30%	942,470	83.85%	893,017	74.20%	1s	6s	1s	5h
N73D14	1,611,787	4,377,449	171.59%	5,978,047	270.90%	5,967,137	270.22%	16s	97s	3s	3h
NE1200P430	1,839,107	14,794,857	704.46%	21,515,705	1,069.90%	21,515,705	1,069.90%	19m	1h	3s	50h
NE600P150	402,727	2,694,672	569.11%	3,726,726	825.37%	3,726,726	825.37%	3m	4m	3s	12h

The cost of every configuration phase expressed in clock cycles has been increased by five (JTAG overhead)[27]. In addition, the same overhead has been taken into account for calculating the cost of a test phase. This cost consists of the length of the longest path and the length of the currently active path increased by two (test pattern termination symbols).

All modelled, detectable faults were detected in each of the experiments, thus reaching full coverage.

A comparison of the enhanced approach from Section 2.4 against the approach described in Section 2.3, depth-first and evolutionary approaches is shown in Section 2.6.1. Reported data related to the evolutionary approach are taken from [27]. For the depth-first approach, data have been newly generated on the ITC16 benchmarks by running the tool implementing the same algorithm as in [25]. For each algorithm, Section 2.6.1 reports the duration in clock cycles of the generated test sequence (referred to as *Test Application Time*) and the CPU time required to apply the algorithm (referred to as *Generation Time*).

Remarkably, results in Section 2.6.1 show a clear improvement regarding the total test time, since the results delivered by the previous approaches were worse up to 705% for depth-first, up to 1,070% for depth-first and evolutionary method. Moreover, the test sequence generated by the proposed approach is shorter than the sequences obtained by the other algorithms in all networks.

Concerning the runtime, as Java's non-determinism prevents an accurate timing, only the total time is reported for all programs (wall-clock). The proposed algorithm completes in the order of seconds, while 19 minutes was required only for one network (NE1200P430). The depth-first algorithm is very fast to execute, even for large networks. The evolutionary approach, on the other hand, requires hours. Moreover, the results reported in [27] for the evolutionary approach were gathered on a multi-core server, exploiting parallelism, while a simple laptop has been used to run the proposed approach.

2.6.2 Experiments for Evolutionary approach from Section 2.5

This subsection reports experimental results obtained using the technique from Section 2.5 on a sub-set of the ITC'16 benchmark networks. The effectiveness is shown by comparing it to the previous approach from which it evolved [27] and the sub-optimal approach based on the depth-first algorithm [25].

The main reason why not all networks from the benchmark set have been considered is that they contain some constructs that are currently not supported by the tool. The networks from the evaluation set differ in the number and type of reconfigurable modules and therefore in the number of configuration bits, hierarchical depth etc.

The whole framework setup consists of three modules. First, the *evolutionary engine* μGP [53] which generates new individuals by applying genetic operators. A next generation of individuals is created based on the fitness values of the newly created offsprings. The second module, the *evaluator*, is written in Java and works independently. Its role is to provide the complete set of transitions for each of the individuals calling the *Transition* function. Additionally, for each of the individuals generated by the μGP the fitness scores are formed based on the values returned by the *Evaluation* function that calculates the fault coverage and the number of clock cycles required to apply the generated test sequence. The tool is able to read a file containing the network description in various formats, including the ICL. Finally, a separate tool – *individual optimizer* is developed in Java and can be optionally used to further reduce the test cost by manipulating the best individual created by the μGP.

The experiments were run on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM (evolutionary phase) and on a laptop with dual Intel i5-7200U CPU and 8GB of RAM (post-processing phase). The server was used to run the evolutionary engine and perform evaluations for each of the individuals, while the laptop was used to perform the post-processing. To emphasize, the reason behind running the evolutionary and post-processing algorithms on two different platforms is not necessity, but commodity, since the algorithms have been developed in different environments. Additionally, the post-processing phase does not require large amount of RAM so there was no obvious advantage of running this task on the server as well. However, if this phase is executed on the server, wall-clock time would be conservatively reduced up to five times.

For each benchmark RSN, the sub-optimal approach based on the depth-first algorithm that traverses the RSN isomorphic graph structure has been executed (it requires a single run). The depth-first approach is very efficient in terms of time and requires few seconds to complete. The evolutionary approach has also been run on each benchmark and compared with the depth-first approach. The experiments executed on the server have been parallelized using up to 8 cores.

The μGP parameters were configured as follows: n_p set to 200, n_o set to 120, while a steady state of 500 generations was chosen. Concerning genetic operators, the following mutation operators have been enabled: insertion, removal, replacement, alteration, swap; and for crossover: one-point precise/imprecise, two-point precise/imprecise, inver-over [54].

The initial population is composed of:

- individuals that may contain random configurations (due to *random* macro);
- apart from random configurations, individuals may contain partially predefined configurations, i.e., in this case configurations generated by the ATPG approach

- an individual with a sub-optimal solution (depth-first) that has been directly inserted into the population - seeding.

Table 2.8 provides experimental results and is organized as follows: the *evolutionary* segment including columns 2-8 reports results regarding the evolutionary stage, while the second, *post-processing* segment (columns 9-12) provides results obtained by employing described post-processing techniques. For each of the benchmark networks, the wall-clock time (in hours) required by the evolutionary algorithm to reach the steady state is given in column 2 (*Wall-clock time*). The column 3 (*#macros*) reports the total number of macros defined in a constraint file for each of the networks. The number of evaluated individuals and the number of generations used by the evolutionary algorithm are given in columns 4 (*Eval.ind.*) and 5 (*Gen.*), respectively. Then, the number of configuration (*#conf*) and test (*#test*) vectors as well as the total time in clock cycles (*Test time*) required to apply the test sequence delivered by the evolutionary algorithm are reported in columns 6-8. The wall-clock time for post-processing to be applied is reported in column 9 of the same table. After running post-processing algorithm on the test sequence generated by the evolutionary engine, a potentially modified sequence is obtained for which the number of configuration (*#conf*) and test (*#test*) vectors are given in columns 10 and 11, respectively. The total time (number of clock cycles) needed to apply the aforementioned sequence is contained in column 12 (*Test time*).

A comparison between the presented approach and the two previously described approaches (the evolutionary approach [27] and the depth-first approach [25]) is given in Table 2.9. For all three approaches the table reports the number of configuration vectors (*#cv*) and the number of test vectors (*#tv*), as well as the total time in clock cycles required to apply the generated sequence (*Test time*). In addition, the results obtained resorting to the proposed approach have been confronted with the results from [25] and [27]. The numbers are given in percentages in the last two columns, respectively; they are calculated based on how much is the new *Test time* reduced with respect to the previous results.

Applying the generated test sequences results in achieving full test coverage, i.e., 100%, given the adopted fault model. Furthermore, by only rewriting the *Transition* function which is used to generate the configuration vectors between two test steps we were able to achieve up to 27% decrease in total test cost for 6 out of 16 benchmark networks when compared to the depth-first approach. In some cases, due to the size and complexity of the networks, seeding the population with the sub-optimal solution individual has led the evolutionary algorithm to saturate the population, thus not improving the inserted sub-optimal solution. However, introducing the described post-processing methods led to a further decrease of the total test cost for the remaining circuits, i.e., in total in 14 out of 16 cases. The post-processing has shown to be highly effective even for the two large networks (NE1200P430 and NE600P150). The results for the networks with low hierarchical depth have not been particularly influenced (small improvement or none) by the new technique, probably due to their low hierarchical depth and small number of test vectors. In these cases, the depth-first approach has most likely produced the solution close or equal to the global optimum. Additionally, here we report only basic statistical qualifiers such as minimum, maximum and median values of time reduction due to the limited and insufficient number of benchmark networks. When the proposed approach is confronted to [25], the latter values are 0%, 27% and 10.1%, respectively; when compared to [27], 0%, 26.6% and 7.1% values are derived.

Table 2.8: Experimental results on the ITC'16 benchmark networks

	evolutionary							post-processing			
Network	Wall-clock time [h]	#macros	Eval. ind.	Gen.	#conf	#test	Test time [cc]	Wall-clock time [min]	#conf	#test	Test time [cc]
Mingle	8	13	49,169	576	6	7	2,135	<1	6	7	2,014
Tree Balanced	6	47	43,914	500	7	8	69,369	<1	7	8	63,843
TreeFlat_Ex	13	38	34,931	1,178	22	6	52,086	<1	22	6	52,086
TreeUnbalanced	5	31	31,329	818	17	12	1,026,333	<1	12	12	1,021,023
a586710	15	14	49,129	500	5	5	299,624	<1	5	5	298,210
p22810	32	78	21,001	500	2	3	152,937	1	2	3	152,399
p34392	68	32	26,292	1,069	5	5	196,223	<1	5	5	196,128
p93791	27	48	29,932	500	4	5	708,878	1	4	5	706,242
q12710	24	16	19,900	500	2	3	131,022	<1	2	3	131,022
t512505	8	40	21,279	500	2	3	386,024	<1	2	3	385,440
N132D4	3	46	47,177	552	5	6	38,731	<1	5	6	31,645
N17D3	7	15	59,384	509	4	5	3,841	<1	4	5	3,797
N32D6	3	15	36,786	419	4	5	904,974	<1	4	5	856,406
N73D14	2	36	34,075	774	14	13	6,078,868	<1	13	13	4,762,150
NE1200P430	78	317	48,902	500	127	128	21,515,705	4k	127	128	16,131,171
NE600P150	19	286	45,857	500	78	79	3,726,726	180	78	79	2,735,016

Table 2.9: Comparison of the experimental results with the approaches from [25] and [27]
.

	Depth-first [25]			Evolutionary [27]			Proposed approach			Comparison	
Network	#*cv*	#*tv*	Test time [cc]	#*cv*	#*tv*	Test time [cc]	#*cv*	#*tv*	Test time [cc]	Test time reduction vs. [25]	Test time reduction vs. [27]
Mingle	6	7	2,282	6	7	2,078	6	7	2,014	11.7%	3.1%
Tree Balanced	7	10	69,369	7	8	69,369	7	8	63,843	8.0%	8.0%
TreeFlat_Ex	5	6	71,341	22	6	55,776	16	6	52,086	27.0%	6.6%
TreeUnbalanced	11	12	1,071,799	12	12	1,042,450	17	12	1,021,023	4.7%	2.1%
a586710	4	5	299,624	5	5	298,241	5	5	298,210	0.5%	0.0%
p22810	2	3	152,937	2	3	152,937	2	3	152,399	0.4%	0.4%
p34392	4	5	196,702	5	5	196,505	5	5	196,128	0.3%	0.2%
p93791	4	5	708,878	4	5	708,878	4	5	706,242	0.4%	0.4%
q12710	2	3	131,022	2	3	131,022	2	3	131,022	0.0%	0.0%
t512505	2	3	386,024	2	3	386,024	2	3	385,440	0.2%	0.2%
N132D4	5	6	38,731	5	6	37,257	5	6	31,645	18.3%	15.1%
N17D3	4	5	4,143	4	5	3,851	4	5	3,797	8.4%	1.4%
N32D6	4	5	942,470	4	5	893,017	6	5	856,406	9.1%	4.1%
N73D14	12	13	5,978,047	13	13	5,967,137	13	13	4,762,150	20.3%	20.2%
NE1200P430	127	128	21,515,705	127	128	21,515,705	128	128	16,131,171	25.0%	25.0%
NE600P150	78	79	3,726,726	78	79	3,726,726	78	79	2,735,016	26.6%	26.6%

2.7 Chapter Summary

This chapter introduced several new approaches to minimize the test time of reconfigurable modules in an RSN.

One of the methodologies is primarily based on evolutionary computation. Additionally, the problem of finding suitable test configurations has been converted into a circuit suitable for applying the automatic test pattern generation procedure. An optimized transition function and some techniques for post-processing the solution delivered by the evolutionary engine have also been presented. Experimental results on the standard set of benchmark networks show the effectiveness of the proposed approach, since the test time has been reduced up to 27% in 14 out of 16 cases, particularly impacting the test time for large networks.

Remaining two methodologies can be defined as semi-formal because the FSA that models the circuit is exact, but incomplete, and the search procedure is based on a greedy algorithm. Experimental results on the ITC'16 benchmark suite clearly demonstrate the effectiveness of the approaches: the proposed techniques are able to achieve better results with less computation effort than previous methods.

Chapter 3

Diagnosis

Although the IEEE 1687 standard alleviated many problems and resolved many issues, it has also introduced some additional ones. Testing traditional scan-chains for permanent faults is relatively simple, since shifting so called, flush sequence (a sequence of alternated 1s and 0s) through the scan chain is sufficient to detect and even understand the type of defect affecting it, if any [47]–[49]. On the other hand, to test an RSN, apart from testing the capability of FFs (comprising TDRs) to shift, modules such as SIBs and ScanMuxes also have to be tested. Without considering the test of reconfigurable elements, no guarantee can be given that applying any valid configuration will result in network changing its state or being in a state corresponding to the wanted one. The previous chapter addressed exactly those issues, by introducing newly developed techniques to test RSN reconfigurable modules.

However, the problem becomes even more complex when discriminating between the faults affecting SIBs and ScanMuxes is required. Even though a number of works is focused on identifying faults affecting scan-chain [55]–[60], little attention has been given to resolving the issue of fault diagnosis within RSNs. The main motivation of this work is to determine which RSN modules (TDRs, SIBs, ScanMuxes) are potentially affected by a permanent fault. Identifying the faulty reconfigurable module is a challenging task given the complexity of the current RSNs and reducing the duration of the diagnosis procedure is crucial. Once the faulty reconfigurable module is identified, the diagnostic procedure may be completed resorting to techniques already available.

Although stimulating instruments and collecting responses could eventually resolve the issue of ambiguity between the modules, the functional access to the instruments is not being considered here. Therefore, identifying classes of undistinguishable faults is necessary. A fault affecting any of the elements within the same class of equivalence has the same effect on the output, no matter which input stimuli is applied to the network.

Since the effect of a fault observed at the output differs depending on the module it affects, the main goal can be divided into two categories:

- discriminating between fault-affected TDRs and detecting the pairs of TDRs defined as *undistinguishable*;
- discovering a fault-affected reconfigurable module (SIB, ScanMux).

To recapitulate, this chapter is going to introduce a technique to identify and localize permanent faults affecting reconfigurable modules in RSN. In more details, a technique is presented to produce a diagnostic sequence of stimuli able to localize the faulty element. The proposed method relies on a semi-formal approach: a Finite State Automaton is dynamically built and then used by a heuristic algorithm to generate a quite effective sequence able to diagnose all permanent faults in the target RSN. Such a diagnosis has to be complemented with the diagnosis of remaining components of the RSN.

3.1 Fault model and Diagnostic Mechanism

Introducing reconfigurability has made diagnosis more challenging. Apart from localizing the faults affecting the ability of flip-flops forming TDRs to correctly shift values, distinguishing between faults affecting reconfigurable elements is also required. The high-level fault model introduced in [25] and then adopted in several works (e.g., [27], [51] and [28]) is used as an abstract representation of defects on network modules. Although it was originally designed after analyzing the effects of possible stuck-at faults, it has certain limitations regarding certain faults (e.g., faults affecting reset and enable logic).

Faults affecting TDRs are considered at the level of a single FF composing the TDR. In this case, a pair of stuck-at faults (stuck-at-0 and stuck-at-1) may affect the output of the FF. Consequently, when a faulty TDR is accessed, repeated subsequent values of 0s or 1s are observed at the output of the scan chain. Faults affecting two different FFs within the same TDR can not be distinguished between themselves, at least not using only structural information. However, performing access at the instrument level to control and collect responses is not being considered in this work. All stuck-at-0 and stuck-at-1 TDR's FF faults are grouped into two TDR faults, respectively. While the consideration that the scan cells may be affected exclusively by stuck-at faults is an important simplification, additional fault models may be easily taken into account thanks to the high-level nature of the approach, e.g., timing faults including *slow faults* (slow-to-rise, slow-to-fall and slow) and *fast faults* (fast-to-rise, fast-to-fall and fast), resulting from setup/hold-time violations.

Although flush patterns are both sufficient and efficient to detect defects and determine their type, they cannot identify the faulty scan cell(s). As different flush patterns exist, the effect of permanent faults of different type on the inserted pattern (00110011) is shown in Table 3.1. Additional type to be considered is intermittent fault. Increasing diagnostic resolution which is currently at the level of the scan segment may benefit from the approaches presented in survey [57]. Different flush sequences may be used and failures from multiple failing scan patterns may be analysed to trace back failures to the origin.

Table 3.1: Set of possible scan-chain fault models and their effect on the inserted pattern 00110011.

Fault models	Output effect (permanent faults)
Fault-free	00110011
Slow-to-rise	$001\underline{0}001\underline{\mathrm{X}}$
Slow-to-fall	$0\underline{1}110\underline{1}1\underline{\mathrm{X}}$
Slow	$0\underline{1}1\underline{0}0\underline{1}1\underline{\mathrm{X}}$
Fast-to-rise	$\underline{\mathrm{X}}011\underline{1}011$
Fast-to-fall	$\underline{\mathrm{X}}0\underline{0}100\underline{0}1$
Fast	$\underline{\mathrm{X}}0\underline{0}1\underline{1}0\underline{0}1$
Stuck-at-0	$00\underline{00}00\underline{00}$
Stuck-at-1	$\underline{11}11\underline{11}11$

Faults affecting SIB and ScanMux reconfigurable modules are modelled as high-level stuck-at faults. Accordingly, a SIB can be *stuck-at asserted* or *stuck-at de-asserted*. Regardless of the configuration, a fault-affected SIB may be permanently bypassing or including the associated segment. The effect of such a fault after configuring the network, is that the path between TDI and TDO differs from the expected one (*faulty path*). By shifting in a sequence of alternated 0s and 1s the same sequence will appear at the output after a number of clock cycles different than the expected one. Similarly, a ScanMux may be stuck-at one of its configurations (e.g., for a 2-to-1 ScanMux, faults are stuck-at-0 and stuck-at-1, while for a 4-to-1 ScanMux, faults are stuck-at-00, stuck-at-01, stuck-at-10, and stuck-at-11). A corresponding input branch is always selected no matter the configuration. The same effect on the scan path length can be observed in this case. By shifting in a sequence of alternated 0s and 1s the same sequence is going to appear at the output after a different number of clock cycles with respect to the expected one.

According to this high-level fault model, one can perform diagnosis on an RSN by configuring the RSN so that the target fault is excited, shift in the sequence of alternated 0s and 1s, and observe when it will appear at the output. By comparing the length of the activated path against the lengths of all other path lengths including faulty ones and the expected one the existing fault can be identified. Especially

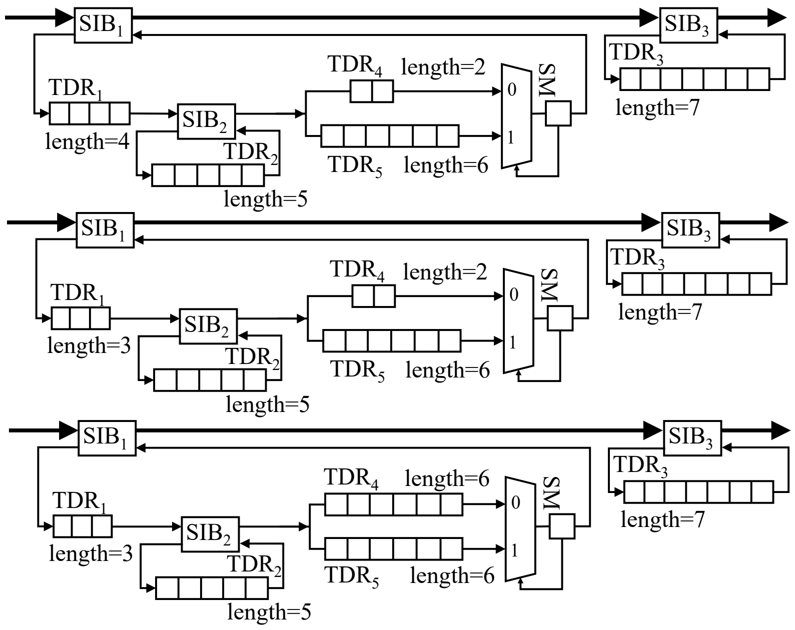


Figure 3.1: Examples of IEEE 1687 RSNs: the top image as a reference one; the middle one with TDR_1 of different length; the bottom one with equal length TDRs (TDR_4 and TDR_5) on SM input segments.

when all applicable configurations, starting from the initial one, result in having different path length, the above approach is easily applicable. As an example, the high-level fault affecting the SM in the network shown in Fig. 3.1 (top), which always selects the segment connected to the input 1, is considered. The faulty module can be excited by a configuration that selects its input 0; an additional requirement is for the module itself to be included into the active path, otherwise the fault is masked. Configurations C_8, C_{10}, C_{12} and C_{14} (given in Table 3.2) fulfil these conditions. Once one of them is activated, one can measure the length of the active path by shifting a given sequence (called diagnostic vector) through TDI and checking when it will appear on TDO. It has to be noted that the total number of clock cycles required to apply the generated diagnostic sequence, namely, cost, is not influenced in the same manner by the choice of different configurations from the aforementioned set.

However, it is not always trivial to localize the fault, since different faults may result in having the same active path length. For example, in the network from Fig. 3.1 (middle), stuck-at-asserted faults on SIB_1 and SIB_2 result in having the same path length ($9 = SIB_1(1) + TDR_3(7) + SIB_3(1) = TDR_1(3) + SIB_2(1) + TDR_4(2) + SM(1) + SIB_1(1) + SIB_3(1)$). This issue can be resolved by continuing to stimulate the network, until a unique sequence of path lengths is observed taking into account the previous active path lengths. The principle is described in more details in section Section 3.2. Moreover, some of the faults may remain undistinguishable.

Table 3.2: Set of possible configurations of the RSN in Fig. 3.1 (top).

Config.	SIB_1	SIB_2	SM	SIB_3	Active path	Len.
C_0	D	D	0	D	-	2
C_1			1			
C_4		A	0			
C_5			1			
C_2	D	D	0	A	TDR_3	9
C_3			1			
C_6		A	0			
C_7			1			
C_8	A	D	0	D	TDR_1, TDR_4	10
C_9	A	D	1	D	TDR_1, TDR_5	14
C_{10}	A	D	0	A	TDR_1, TDR_3, TDR_4	17
C_{11}	A	D	1	A	TDR_1, TDR_3, TDR_5	21
C_{12}	A	A	0	D	TDR_1, TDR_2, TDR_4	15
C_{13}	A	A	1	D	TDR_1, TDR_2, TDR_5	19
C_{14}	A	A	0	A	TDR_1, TDR_2, TDR_3, TDR_4	22
C_{15}	A	A	1	A	TDR_1, TDR_2, TDR_3, TDR_5	27

Pairs of faults affecting a ScanMux module with the same length of TDRs on its input branches belong to this group. By using this approach, it remains impossible to differentiate between stuck-at-0 or stuck-at-1 faults on the SM module ($6 = TDR_4(6) = TDR_5(6)$) from Fig. 3.1 (bottom).

The same procedure is used to the single permanent fault diagnosis on the RSN elements (TDRs, SIBs and ScanMuxes). The complete generated procedure is organized as a set of *sessions*, each composed of a diagnostic step and a configuration step. Configuration step corresponds to shifting configuration bits in the scan chain and performing update. Each diagnostic step consists of the following phases:

1. shifting in the first sequence consisting of same values (all 0s or all 1s), while the length of the sequence is equal to the length of the longest path in the network; the goal of this phase is the initialization of the scan cells;

2. shifting in the second sequence of alternated 0s and 1s (i.e., 0101...01), with the predetermined length of the sequence (equal to the maximum length of the

expected path and all faulty paths). As a sequence terminator two identical bits (either 00 or 11) are added;

3. the last, diagnostic sequence shifts values from the currently active path.

In a standard TAP controller, reaching the ShiftDR state from the UpdateDR state requires visiting CaptureDR state. Therefore, Step 1) is required when capture values are either not defined or not considered. Each configuration and diagnostic step can be translated into a JTAG vector-*Scan Data Register* (SDR). SDR is a state command to perform an IEEE 1149.1 Data Register scan and is defined within Serial Vector Format (SVF). It is used with 4 arguments: TDI, TDO, MASK and SMASK, i.e., the value to be scanned into the target, the values to be compared against the actual values scanned out of the target, the mask to be used when comparing TDO values against the actual values scanned out of the target, and mask for specifying TDI data that is "don't care", respectively. A configuration step corresponds to a JTAG vector (SDR) of a predetermined length taking into account active path, while an SDR vector corresponding to an observation step followed by a configuration step contains increased number of shift operations. The latter situation is known as "overscanning" when scanning longer than the length of the longest path.

Determining fault-caused modifications of values in the scan chain and the length of the active scan path is performed by verifying inserted diagnostic vector; in parallel with observing the values appearing at the output, new configuration vector is shifted in. The path length is deduced from the position of sequence termination symbol. Finally, applying the configuration vector demands an update operation. The duration of the complete diagnostic procedure, referred to as a total cost, depends on the duration of each step and is composed of configuration step cost and diagnostic step cost, both expressed in terms of number of clock cycles. The configuration step cost is the time needed to apply configuration vectors. The time overhead of the JTAG protocol is also included, since moving the TAP controller from shift to update state and vice versa also requires a few clock cycles. The diagnostic phase cost is the time required to shift in the diagnostic sequence. Furthermore, the duration of a session is determined by the length of the TDRs included in the path, as well as by the previous configuration.

Consideration that the TAP controller is used to access and control the network imposes certain restrictions. The TAP Finite State Machine with its defined transitions is able to traverse 3 main states (capture, shift and update) in the following order: either capture, shift and then update or capture and then update, avoiding the shift state. Therefore, without acquiring and applying certain design techniques to improve the observability of such registers [39], it is not possible to check if one shift operation destroys/overwrites previously shifted-in data. If such defect is present, after shifting data into some other scan element in parallel or rather than into the desired one, some update operation must be performed, followed by a capture. Consequently, any previously stored data is overwritten, thus removing any trace of unwanted/undesired access caused by the fault.

3.2 Proposed Diagnostic Methodology

As in some of the previously presented approaches to test the RSN modules [51][61], an IEEE 1687 RSN is modelled as a finite state automaton (FSA). Each state of SIBs and ScanMuxes in the network, referred to as *configuration*, represents an automaton state. The input alphabet corresponds to the possible network's reconfiguration operations. Apart from being in relation to the fault model, the length of the active path is an easy obtainable property [25]. Therefore, output symbols are mapped to the lengths of the active paths. Although the high-level model is exact in modelling the circuit, it is deliberately incomplete, since the FSA's states encode only a subset of the possible configurations. Due to the particular structural properties of the RSN, not all transitions are possible in all states. When an input does not correspond to a transition, the FSA is brought to a special *sink state* (Ω), that is a state with no output transitions and a *null* output symbol. For instance, in some networks it is possible to use a Scan Multiplexer whose configuration is based on the values of multiple configuration bits (n). However, such multiplexers do not necessarily have defined inputs for all possible configurations (2^n) in the ICL description of the network. Even though at the implementation level, either at the gate- or RTL-level, these pins might be tied to some other input or to logical 0/1, to prevent any ambiguities, transitions to such unspecified configurations lead to a special state.

Given the stuck-at faults affecting SIBs and ScanMuxes, the same configuration operations may result in different network statuses on faulty circuits. Therefore, such faults are mapped to multiple transition faults on the high-level automaton. Taking into account the faulty automata and the good one, the goal of the diagnostic procedure is to produce a sequence of inputs able to make a unique discrimination between each one of them.

A greedy search strategy represents the basis of the proposed algorithm. Not all possible states nor all possible input symbols are considered, and, consequently, not all possible transitions. Nevertheless, the simulation of the automaton is exact, while any missing state or transition will cause the automaton to reach the *sink state*, that by construction cannot be further distinguished from any other state.

3.2.1 Finite State Automaton to model an RSN

Initially, the FSA is composed of only a state with no output transition and a *null* output symbol. Such *sink state* is used to denote a pathological condition, where the algorithm is not able to provide reliable results due to the approximation of the model. This state is characterized by its ambiguity with respect to any other state. Once entered, the FSA permanently remains in this state.

Next, the state when all configuration bits are set to the initial value, denoted as *reset state*, is added to the automaton. Then, for each SIB$_i$, two states are created: one with the SIB asserted and one with the SIB de-asserted. Similarly, for each SM, one state is created for each possible configuration.

Such a straightforward approach, however, is not always sufficient. Scan segments may be nested, and a resource accessible only when its parent SIB is asserted. The procedure for building the FSA detects such situations, and creates the necessary states to handle them. The transitions from the *reset state* to all these states are eventually added.

Since the FSA is built in an incremental mode the following modifications are performed:

- for each transition in the good automaton, the possible faulty transitions are added;
- if the faulty transition would bring the automaton in a configuration not already encoded as a state, that specific state is added to the FSA;
- all nonexistent transitions between existing states are added to the automaton;
- eventually, all possible faulty transitions from all existing states are also added, but if one would bring the automaton in a configuration not encoded as a state, its destination is set to the *sink state*, meaning that the FSA is unable to model such situation.

As almost only the states with a hamming distance of 1 from the reset state are added to the FSA, the size of the automaton is linear in the number of configuration bits. It is possible to define an automaton with more states: for instance, at some point of the creation, all complementary states may be added as well. It is important to remember that the size of the automaton influences both the quality of the results and the performance of the algorithm.

While considering the possible transitions, some additional states corresponding to the configurations which may reduce the cost are also examined. Of course, it is important to remember that the highest priority is to continuously increase the number of diagnosed faults, while reducing the cost is a secondary goal. For example, states representing configurations in which accessible SIBs that provide access to the deepest hierarchical level are not asserted may increase the number of required sessions and therefore the cost. Additionally, configurations in which a SIB is still asserted while already all faults associated with it and its sub-hierarchical modules are diagnosed may increase the cost. If all faults affecting ScanMux and its sub-hierarchical modules are diagnosed, the configurations in which a ScanMux branch with not minimal length is included into the path might also increase the cost. It is worth noting that, although designers may explicitly define and include some additional states to this state or provide additional heuristic, experimental validations suggest that such extensions are unlikely to be beneficial.

For the network given in Fig. 3.3, FSA with its states and transitions is given in Fig. 3.2. The states are represented by circular shapes with the label and output symbol, while the lines with arrows denote the transitions between the states. Initially, only states filled with white color (*sDDDD, sDDDA*, *sDDAD*, *sDADD*, *sADDD*) are created. States in light grey are created consequently, dynamically, after applying the chosen input symbols and performing transitions on fault-free FSA.

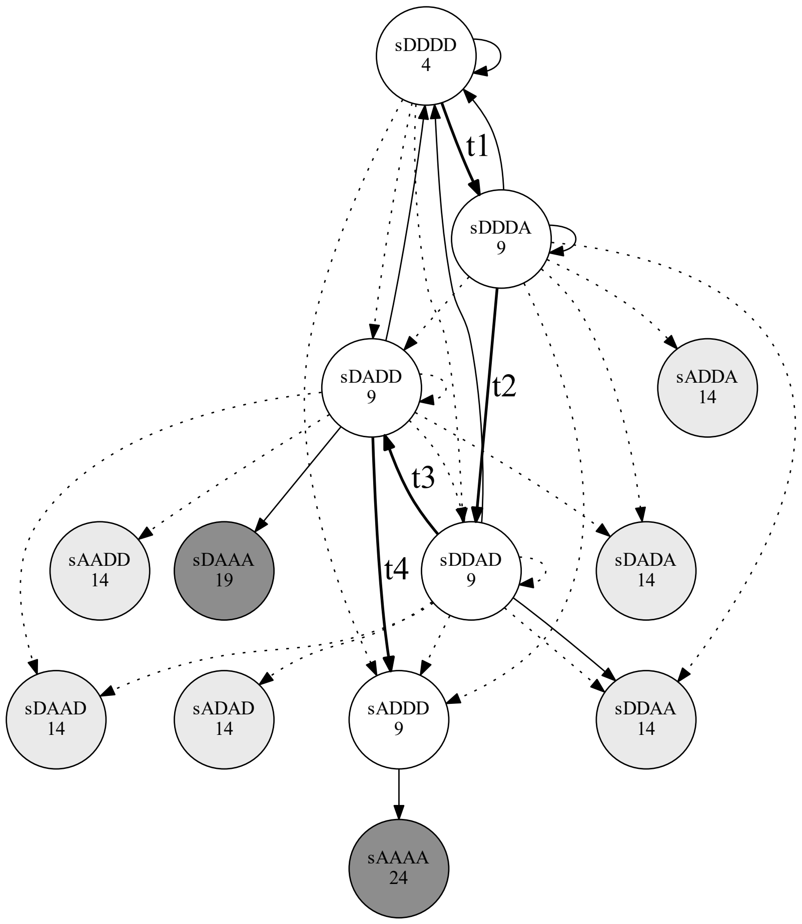


Figure 3.2: Example of generating FSA for the network from Fig. 3.3
.

Remaining states, in dark grey are created as a result of transition on faulty FSAs. As for the transitions, some of them are created for examining the end states (dash lines). The transitions drawn in bold lines correspond to transitions of the fault-free FSA (t_1–$\{sDDDD \rightarrow sDDDA\}$, t_2–$\{sDDDA \rightarrow sDDAD\}$, t_3–$\{sDDAD \rightarrow DADD\}$, t_4–$\{sDADD \rightarrow sADDD\}$). The remaining transitions are executed on the faulty FSAs while applying the sequence of chosen input symbols (in solid style).

3.2.2 Search Algorithm

A sequence of *transition* and *observation* steps is constructed by the search algorithm. A *transition* corresponds to a change in the configuration of the RSN and it is performed by shifting an array of bits into the scan chain. On the other hand, an *observation* step does not change the configuration of the RSN and does not affect the FSA, since it is comprised only out of shift operations.

The goal of the diagnosis sequence generation procedure is to make the good circuit and the faulty ones pass through different states to be able to distinguish between them by comparing sequences of output symbols of the traversed states. For every circuit a sequence of scan chain lengths is observed; if it is unique then a fault is considered to be diagnosed.

Let x be an input symbol for the FSA. The reset operation is denoted with **reset**, and it requires a single clock cycle to be performed; measuring the length of the scan chain is characterized with **observe** symbol. Moreover, it requires a certain number of clock cycles and does not affect the state of the FSA. Both concatenation of the two sequences and appending a symbol to an input sequence are expressed as additions, as no ambiguities are possible. The symbol $\varnothing$ denotes an empty symbol and has no effect on an input sequence, e.g., $\mathbf{t} = \mathbf{t} + \varnothing$. The state of the FSA is unambiguously defined with a sequence $\mathbf{t}$ of inputs starting with a **reset**, i.e., $\mathbf{t} = (\mathbf{reset}, +t_0, t_1, ..., t_k)$.

Two states that have undistinguishable output symbols are equivalent and are denoted with $s' \cong s''$. Conversely, non equivalent states have distinguishable output symbols and are denoted with $s' \ncong s''$. By definition, the sink state is equivalent to any other state $\forall s : s \cong \Omega$.

Let $\bar{\Lambda}_{\mathbf{t}}$ be the sequence of states $[\bar{s}_{\mathbf{reset}}, \bar{s}_{t_0}, \bar{s}_{t_1}, ..., \bar{s}_{t_k}]$, the FSA representing the fault-free circuit goes through after applying the input sequence $\mathbf{t}$, while $\Lambda^i_{\mathbf{t}}$ be the sequence of states $[s^i_{\mathbf{reset}}, s^i_{t_0}, s^i_{t_1}, ..., s^i_{t_k}]$ the FSA representing the circuit when fault i is present goes through when the same input sequence is applied. The two Λ sequences Λ^i_t and Λ^j_t are considered to be different if there are at least two different output symbols, corresponding to the same state position in both of the sequences (e.g., $s^i_{t_{k-1}}$ and $s^j_{t_{k-1}}$). Moreover, the sequence $\Lambda^i_{\mathbf{t}}$ is unique, if $\Lambda^i_{\mathbf{t}} \ncong \bar{\Lambda}_{\mathbf{t}}$ and $\forall j, j \neq i$, $\Lambda^i_{\mathbf{t}} \ncong \Lambda^j_{\mathbf{t}}$. Having a unique sequence Λ^i_t allows to mark the fault i as diagnosed, by appending an **observe** input symbol to $\mathbf{t}$. The fault i is said to be “diagnosable”.

Let $\mathrm{DF}(\bar{\Lambda}_{\mathbf{t}}, \mathbf{S_t})$ be the set of potentially diagnosable faults when the good circuit traversed the states in $\bar{\Lambda}_{\mathbf{t}}$ and the faulty ones $\mathbf{S}_{\mathbf{t}} = (\Lambda^0_{\mathbf{t}}, \Lambda^1_{\mathbf{t}}, ..., \Lambda^{f-1}_{\mathbf{t}})$, i.e., the set of all faults that caused the faulty circuit to traverse the states in a unique Λ^i sequence. If an observation is performed, measuring the actual length of the RSN path, any unique difference would be observed and all such faults, diagnosed. Initially, all faults are annotated with an identical score (equal to -1). During the execution of the sequence generation procedure, these values are updated with the index number of the session in which a fault was diagnosed. If by running the Diagnostic Sequence Generation procedure, not all faults from the **F** list are diagnosed, the same procedure is run from the beginning, this time with the updated **score** priority list.

A sequence of input symbols is generated iteratively since in each run the function Greedy(Algorithm 10) searches for the most optimistic input symbol for the sequence of inputs **t** to be extended with (Algorithm 11). In other words, the appended input symbol brings circuits in states where the highest number of faults with the lowest score can be diagnosed. A fault with a low score assigned means that it was either not diagnosed in the previous run, or it was diagnosed before some others (with a higher score). If no new fault can be diagnosed by adding a single transition, the function Greedy returns an empty input sequence. In every iteration, the most useful symbol is appended to the sequence, trying to increase the number of diagnosed faults. When no symbol can result in a fault being diagnosed, *reset* is appended to the sequence and history of taken transitions is considered as an alternative. An *observe* symbol is always added after finding a new useful transition (symbol). Additionally, in each run a set of diagnosed faults is updated.

Algorithm 10 Greedy score step

```
hbt!
function GREEDY($\mathbf{t}$, score)
    $\mathbf{m} \leftarrow (\ )$                                   ▷ Empty sequence of inputs
    for $x \in \{$valid input symbols in $\bar{s}_{\mathbf{t}}\}$ do
        $\mathbf{u} \leftarrow \mathbf{t}$
        Append $x$ to $\mathbf{u}$
        $FS_u \leftarrow (\bar{\Lambda}_{\mathbf{u}}, \mathbf{F}_{\mathbf{u}}, \mathbf{score})$
        $FS_m \leftarrow (\bar{\Lambda}_{\mathbf{m}}, \mathbf{F}_{\mathbf{m}}, \mathbf{score})$
        if $FS_u > FS_m$ then
            $\mathbf{m} \leftarrow \mathbf{u}$
    return m                                      ▷ Most promising sequence
```

3.2.3 Diagnostic analysis

According to the fault model it is obvious that the faults affecting different types of RSN modules have a different effect and can therefore, be distinguished one

Algorithm 11 Diagnostic Sequence Generation

```
procedure DPG(score)
    $\mathbf{t} \leftarrow (\mathbf{reset})$                                          ▷ Initial diagnostic sequence
    $\mathbf{H} \leftarrow \{\mathbf{t}\}$                                             ▷ History
    $\mathbf{F} \leftarrow \{\text{all diagnosable faults}\}$                          ▷ Active faults
    $\mathbf{tscore} \leftarrow \{-1\}$                                                ▷ Initialize fault score
    while $|\mathbf{F}| \neq 0$ do
        $\mathbf{g} \leftarrow \text{Greedy}(\mathbf{t}, \mathbf{score})$
        if empty($\mathbf{g}$) then                                                    ▷ The greedy failed
            Append reset to t                                                          ▷ Start over
            for $\mathbf{t}' \in \mathbf{H}$ do
                $\mathbf{g}' \leftarrow \text{Greedy}(\mathbf{t}')$
                if $|\text{DF}(\bar{\Lambda}_{\mathbf{g}'}, \mathbf{S}_{\mathbf{g}'})| > |\text{DF}(\bar{\Lambda}_{\mathbf{g}}, \mathbf{S}_{\mathbf{g}})|$ then
                    $\mathbf{g} \leftarrow \mathbf{g}'$                                ▷ Alternative sequence
        Append g to t
        Append observe to t
        $\mathbf{H} \leftarrow \mathbf{H} \cup \{\mathbf{g}\}$                         ▷ Save sequence
        Remove $\text{DF}(\bar{\Lambda}_{\mathbf{t}}, \mathbf{S}_{\mathbf{t}})$ from F
        Set tscore for $\text{DF}(\bar{\Lambda}_{\mathbf{t}}, \mathbf{S}_{\mathbf{t}})$
    $\mathbf{score} \leftarrow \mathbf{tscore}$
```

from another. The faults affecting TDRs corrupt the inserted diagnostic sequence, changing the values of particular bits. For example, a stuck-at-1 on a scan cell within a TDR will result in observing only fixed values of 1 on all vector positions. On the other hand, SIB or ScanMux module affected by a fault may result in a path length different than the expected one. Consequently, the diagnostic sequence is not corrupted, but is observed before or after the expected number of clock cycles.

Faults affecting scan cells of a single TDR are all made equivalent. Accordingly, it is said for a fault to affect a TDR and the fault is distinguished at the TDR instance level. The presented approach is able to generate input symbols, i.e., configuration vectors that modify the state of a network in such a way that in each session, some of the SIB modules may become de-asserted and consequently some TDRs may be excluded from the active path, but only one module can be reconfigured to include a new segment to the active path. As shown in Fig. 3.3, all inputs applied to the network not affected by the reconfigurable module faults result in only one TDR being included in the active path in each session ($0001 - \text{TDR}_1, 0010 - \text{TDR}_2, 0100 - \text{TDR}_3, 1000 - \text{TDR}_4$). Therefore, when a diagnostic sequence is applied, a session that fails will specify the faulty TDR.

If the network is designed in such a way, that two or more registers are located in the same segment, they can not be distinguished using only structural information and are referred to as *undistinguishable*. Diagnosing faults affecting this sub-set of

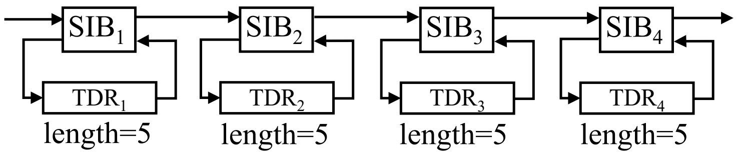


Figure 3.3: 4 serially connected SIBs with TDRs of equal length of 5

TDRs may in some cases be possible using default capture values of the registers, since the TAP controller has to pass through the capture state in order to reach the shift state. Additionally, reading circuit's default capture values without inserting initialization sequence may be used to distinguish faults at the level of a single bit. However, this issue was not considered in this work. As stated in the standard, capture values may be fixed and predefined for some scan elements, i.e., TDRs. However, if the scan elements are defined, not as Write-Only (here, capture functionality is not necessary) but as Read-Only or Read-Write, capture source may be defined as an external signal or value from the shadow, i.e., update stage of that register. Additionally, an approach such as the one we propose may rely exclusively on structural information (given by ICL), before any detailed information on which types of instruments are to be integrated and how they are operated is available.

Faults affecting ScanMux modules may also not always be diagnosable. If a ScanMux module has at least two branches containing equal fixed length segments, then all faults forcing the ScanMux configuration to always select those branches are considered as *undistinguishable*.

One of the advantages of the proposed approach is that is able to handle types of networks constructed out of equally long TDRs placed behind serially connected SIBs. An example is given in Figure 3.3, with four SIB modules and four TDRs with the register length equal to 5. Since it is enough to observe a difference in length, SIB stuck-at-asserted faults in this case are easily detected. However, distinguishing between them is a more challenging task.

Table 3.3 shows the configurations the fault-free and the fault affected networks go through, when input sequence generated by the proposed approach is applied.

As illustrated by Fig. 3.4, after applying reset, depending on the location of the fault affecting a SIB module, the resulting scan chain can have not only a different length, but also a different position of particular scan cells. It can be observed that initially, the length of the scan chain in case of a fault free network and all stuck-at de-asserted faults ($\bar{s}, s^0, s^2, s^4, s^6$) is 4. In this case, the position of configuration bits within the scan chain is the same ($\mathrm{CB}_1, \mathrm{CB}_2, \mathrm{CB}_3, \mathrm{CB}_4$). On the other hand, for all stuck-at asserted faults (s^1, s^3, s^5, s^7), the scan chain is of length 9, with a different configuration bits arrangement, due to the TDRs which are now part of the active path. In case of s^1, register TDR_1 is in the active path, while e.g., in case of s^5, TDR_3 is a part of the active path. In the first case, all four SIB configuration bits (supposed that post-SIBs are used) are located after scan cells comprising TDR$_1$. On the other hand, in the second case two SIB configuration bits (CB$_1$ and CB$_2$) are preceding TDR$_3$, while two are following it (CB$_3$ and CB$_4$). As a consequence of having the same path length for pairs of different faults, at this point no fault can be diagnosed. Faults s^0, s^2, s^4, s^6 can not be distinguished between themselves and between correctly operating circuit (length 4), while faults s^1, s^3, s^5, s^7 are equivalent between themselves (length 9). Regardless the fault, a configuration sequence *cv* can be shifted in the scan chain. In cases where the actual length of the scan chain is lower than the one of the configuration vector, not all values in the configuration vector will be stored in the chain cells; some of them will be "cut-off", i.e., shifted out. By applying the update, new states will correspond to the ones shown in the second row of Table 3.3, thus allowing five faults to be diagnosed (s^1, s^3, s^5, s^6, s^7). Even though the lengths of the active path after applying the *cv* configuration vector in case of the fault-free network (s) and the network in which SIB$_4$ is affected by stuck-at asserted fault (s^7) are equal, the fault is diagnosed. When array of lengths of the active path in presence of the fault $[9, 9]$ is compared against others $\{[4, 9], [4, 9], [9, 24], [4, 9], [9, 19], [4, 9], [9, 14], [4, 4]\}$, it is determined to be unique.

Table 3.3: Diagnostic procedure for the network in Fig.3.3

Input	Fault free	SIB$_1$		SIB$_2$		SIB$_3$		SIB$_4$	
	$\bar{s}$	s@D-s^0	s@A-s^1	s@D-s^2	s@A-s^3	s@D-s^4	s@A-s^5	s@D-s^6	s@A-s^7
reset observe	DDDD (4)	DDDD (4)	ADDD (9)	DDDD (4)	DADD (9)	DDDD (4)	DDAD (9)	DDDD (4)	DDDA (9)
0001 observe	DDDA (9)	DDDA (9)	AAAA (24)	DDDA (9)	DAAA (19)	DDDA (9)	DDAA (14)	DDDD (4)	DDDA (9)
0010 observe	DDAD (9)	DDAD (9)		DDAD (9)		DDDD (4)			
0100 observe	DADD (9)	DADD (9)		DDDD (4)					
1000 observe	ADDD (9)	DDDD (4)							

3.3 Experimental Results

For the purpose of validating the proposed algorithm, a sub-set of the ITC'16 suite of benchmark reconfigurable scan networks [45] was chosen. Not all benchmarks have been used since some of them include constructs which are currently not supported by our environment. However, some additional benchmarks, considered in [22], were included into the validation set to provide effectiveness comparison

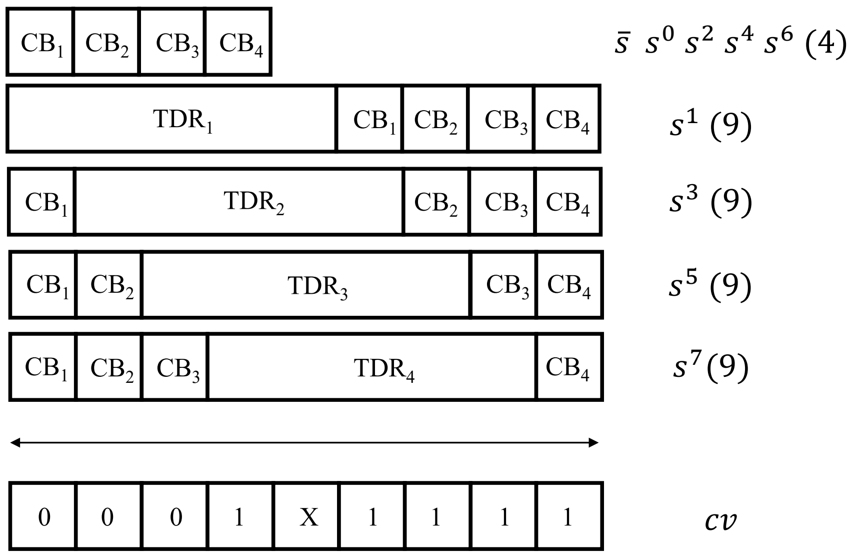


Figure 3.4: Initial scan chain structure with conf. vector for network from Fig. 3.3

with the present approach. It is worth noting that some of the latter benchmarks are part of the ITC'16 set.

An in-house tool implementing the proposed algorithm was developed in Java. The tool is first of all able to find all reconfigurable modules, for which the faults affecting them are not distinguishable. As already discussed, this is due to their inability to produce a different path length. Moreover, the tool can generate a list of sets of TDRs; faults affecting TDRs in the same set are considered to be undistinguishable between themselves, e.g., because they belong to the same segment.

The basic information on the benchmarks' evaluation set is reported in Table 3.4. Columns 2 and 3 give the number of SIBs and SMs for each network. The total number of SIB and SM configuration bits is reported in the fourth column. The depth of a module is equal to the number of nested modules controlling its segment. The hierarchical depth of the network corresponds to the highest module depth in the network and is provided in column 5. The sixth and seventh columns represent the maximum path length and the total number of scan cells in a given network, respectively. The upper part of the table contains the list of networks used to evaluate the approach proposed in [22], while the list of considered ITC'16 benchmarks is contained in the lower part of the same table.

A computer with an Intel i5-7200U processor and 8 GB of RAM was used to perform the experiments. Table 3.5 reports the experimental results obtained from running the proposed algorithm on the set of benchmark networks. In columns 2 and 3, the number of configuration vectors, i.e., the number of diagnosis vectors is given. In this table, the cost of performing the diagnostic procedure on a given network by applying the generated sequence is given in terms of number of clock

Table 3.4: Benchmark networks list

Network	SIB	SM	Tot bits	Max depth	Max path	Scan cells
A586710	6	0	6	2	41,972	41,972
N17D3	7	8	15	4	372	462
N32D6	13	10	23	4	84,039	96,158
N49D0	16	18	34	1	949	1,114
N61D2	11	22	33	2	1,162	1,422
N73D14	29	17	46	12	190,526	218,869
N88D8	32	32	64	4	1,637	2,013
N100D2	31	37	68	3	1,833	2,293
N132D4	39	40	79	5	2,555	2,991
P22810	30	0	30	2	30,915	30,915
P34392	22	0	22	2	23,478	23,478
Mingle	10	3	13	4	171	270
TreeBalanced	43	3	48	7	5,219	5,581
TreeFlat_Ex	57	3	62	5	5,100	5,195
TreeUnbalanced	28	0	28	11	42,630	42,630
a586710	0	32	32	4	42,381	42,410
p22810	270	0	270	2	30,356	30,356
p34392	0	96	96	4	27,899	27,990
q12710	27	0	27	2	26,185	26,185
t512505	159	0	159	2	77,005	77,005
NE600P150	207	194	401	78	23,423	28,250
NE1200P430	381	430	811	127	88,471	108,148

cycles required to apply all configuration steps (*cv*, column 4) and all diagnostic steps (*dv*, column 5).

A number of experiments has been run to evaluate the effectiveness of the high-level fault model used in this work. Selected benchmarks have been synthesised using NanGate 45 nm Open Cell Library. Synopsys tool TetraMAX[1] was used to perform fault simulation on the designs at the gate-level by applying test sequences generated by the method in [28]. The fault simulation results showed that in general a high or complete stuck-at fault coverage is achieved. Certain corner cases appeared as a result of faults affecting modules positioned deep in the hierarchy and with the ScanMuxes having more than 2 inputs. Finally, the stuck-at faults affecting the update logic and the flipflops are either covered or they propagate long sequences of Xs in the circuit. Since the proposed test sessions demand for a precise sequence of 0 and 1 s to be observed on scan output ports, faults that propagate sequences of Xs can be safely marked as covered.

By using the proposed approach and according to the fault model which was

[1]TetraMAX ATPG User Guide, Version M-2016.12, Synopsys, www.synopsys.com.

Table 3.5: IEEE 1687 algorithm experimental results

Network	*cv*	*dv*	Conf. cost [cc]	Test cost [cc]
A586710	6	7	44,168	377,435
N17D3	15	16	3,287	9,492
N32D6	23	24	989,654	3,041,145
N49D0	34	35	20,429	55,029
N61D2	33	34	25,882	67,055
N73D14	46	47	4,883,376	13,945,235
N88D8	63	64	54,488	160,405
N100D2	67	68	73,903	200,260
N132D4	77	78	115,928	317,291
P22810	30	31	62,191	1,023,787
P34392	22	23	65,008	606,897
Mingle	14	15	921	3,591
Tree Balanced	47	48	132,172	393,223
TreeFlat_Ex	62	63	184,536	515,997
TreeUnbalanced	28	29	149,021	1,386,204
a586710	61	62	75,525	2,703,562
p22810	301	302	6,472,380	15,697,407
p34392	187	188	50,712	5,296,399
q12710	25	26	34,082	716,281
t512505	159	160	190,736	12,512,221
NE600P150	399	400	4,186,745	13,563,116
NE1200P430	809	810	39,615,088	111,319, 225

described previously, all distinguishable faults were diagnosed, thus reaching 100% diagnostic coverage. Considering the discussion in Section 3.1, faults affecting modules that are not considered as undistinguishable are considered to be distinguishable. The comparison of the proposed approach with the approach from [22] is given in the upper part of Table 3.6. The lower part of the same table refers to the results obtained on the sub-set of ITC'16 benchmarks. The second column of the table gives the total number of clock cycles needed to apply the input sequence generated by the proposed approach. Comparison data on diagnosis duration in clock cycles are taken from [22] and are provided in column 3. The fourth column shows the comparison against the current result. The ratio between the duration of the diagnostic sequence generated by the previous and new approach is reported, thus showing how faster the latter one is. Due to the Java's non-determinism at run-time no accurate timing in terms of CPU time for generating the sequence is possible. Therefore, only the program's total execution time is reported in column 5. The number of pairs of undistinguishable faults affecting TDRs and SMs is reported in columns 6 and 7.

As it can be observed from the Table 3.6, in all the cases where comparison data exists, the time required to perform the diagnosis is significantly reduced, up to 43 times. Although no data regarding the execution time of the previous algorithm is provided, it is evident from the Table 3.6 that the presented algorithm is efficient and fast to execute, since in most of the cases, the required time is measured in the order of seconds. Exceptionally, more than one hour was needed for three networks.

Table 3.6: Experimental comparison of the proposed algorithm vs. [22]

Network	Diagnostic sequence duration [clock cycles]	[22]	[22] vs. proposed	Runtime (wall clock)	Und. pairs of faults TDR	CM
A586710	421,603	3,879,326	9.20x	0s	0	0
N17D3	12,779	48,099	3.76x	1s	4	0
N32D6	4,030,799	25,038,071	6.21x	0s	16	0
N49D0	75,458	263,866	3.50x	1s	105	0
N61D2	92,937	333,280	3.59x	1s	260	0
N73D14	18,828,611	320,437,112	17.02x	2s	78	0
N88D8	214,893	2,065,800	9.61x	4s	68	0
N100D2	274,163	2,219,397	8.10x	5s	206	0
N132D4	433,219	3,900,952	9.00x	17s	435	4
P22810	1,085,978	46,601,832	42.91x	1s	0	0
P34392	671,905	20,665,284	30.76x	0s	0	0
Mingle	4,512			19s	0	0
Tree Balanced	525,395			48s	9	3
TreeFlat_Ex	700,533			40s	14	3
TreeUnbalanced	1,535,225			23s	14	0
a586710	2,779,087			22s	5	0
p22810	22,169,787			1.3h	6	0
p34392	5,347,111			3m	14	0
q12710	750,363			17s	0	4
t512505	12,702,957			4m	0	0
NE600P150	17,749,861			5h	427	4
NE1200P430	150,934,088			15h	643	4

3.4 Chapter Summary

In summary, this chapter describes a new sequence generation technique to diagnose permanent faults in RSNs resorting to an FSA model of the circuit and a greedy search algorithm. By resorting to overscanning, once a fault is present, applying the set of generated configurations will make the network pass through states. The fault will be identified by observing lengths of the active path.

Experimental results demonstrate that the presented approach outperforms the previous ones in terms of number of clock cycles required to run the generated diagnostic sequence. Furthermore, this technique can be applied to a wide range of network types of different complexity since for all the test cases and benchmark networks full diagnostic coverage has been reached while keeping the computation effort under control.

Chapter 4

NBTI-induced aging analysis in IEEE 1687 RSNs

The expansion of contexts where electronic systems serve people has also led to a significant growth of the number and variety of safety- and mission-critical embedded systems. This comes along with the trend of implementation technology miniaturization that allows boosting the nanoelectronic systems' functionality, but brings the lifetime reliability concern to the front. Autonomous and unmanned vehicles, robotic systems, fly-by-wire aircrafts and complex industry automation machines are often empowered by advanced computing systems and require extreme levels of safety and dependability for years of the operational lifetime.

The phenomenon of nanoelectronics aging was addressed by numerous related works focusing e.g., on the degradation issues caused by the Negative Bias Temperature Instability (NBTI) in memories [62], [63] and in functional logic [64], [65]. To mitigate such effects in functional logic, approaches exist at different levels, e.g., some are based on redesign or transistor sizing techniques [66], while others rely on modifying voltage and frequency of the circuit [67] or resort to NBTI-aware synthesis [68]. Mitigation on a circuit level has been exploited in [69] where authors propose using idle-time of the processor and unused bits in source operands of the instruction. To the best of author's knowledge, no work has addressed the reliability issues caused by NBTI-induced aging in the IJTAG RSNs, so far. This chapter will present the analysis of the effect of NBTI on logic paths in IJTAG RSNs by estimating delay resorting to the model introduced in [65]. Additionally, it contains details on a novel approach to mitigate the degradation with a case-study demonstration. The effectiveness of the approach is evaluated on a sub-set of ITC2016 benchmark RSN designs. It should be however, noted, that this work does not aim at developing new or extending already existing technology-level models for NBTI.

4.1 Hierarchical Modelling of the NBTI-Induced Delays

Bias Temperature Instability (BTI) phenomenon causes threshold voltage V_{TH} shift on MOS transistors. Two types of BTI are defined depending on the type of stressed transistor. Negative BTI (NBTI) caused by the negative gate stress occurs on pMOS transistors, while the Positive BTI (PBTI) is related to nMOS transistors due to the positive gate stress. This paper focuses on NBTI [70] as it is considered to be a dominant aging mechanism for the current implementation technologies.

Two phases can be identified in a pMOS transistor due to NBTI, *stress* and *recovery*. When $V_{GS} = -V_{DD}$, *stress* phase occurs. The transistor is in a *recovery* phase when bias voltage is removed ($V_{GS} = 0$). When the transistor is switching, such phases alternate and the NBTI effect is reversed to some extent. The change of V_{TH_p} of the device under constant stress (static NBTI) is significantly higher when compared to dynamic NBTI, i.e. alternation of stress and recovery phases. Furthermore, the same tendency can be identified within the logic path delay degradation though on a smaller scale. In [71] authors examined different factors and their effect on NBTI degradation. Their results show strong correlation of the NBTI degradation on the duty factor, i.e., input signal probability (stress/recovery).

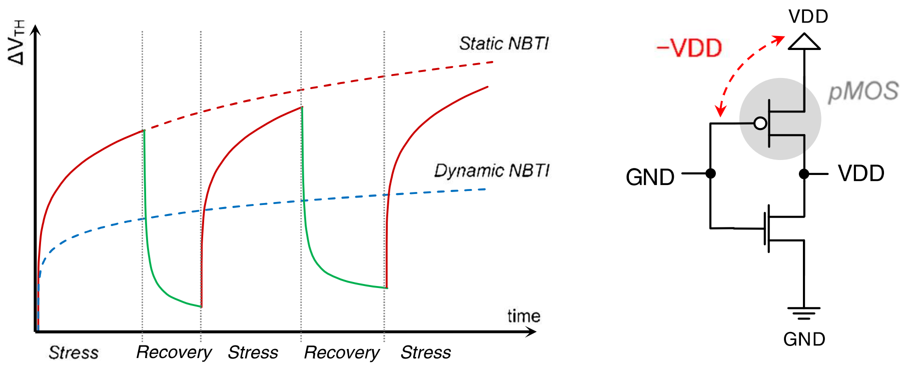


Figure 4.1: Dynamic and static NBTI effect on threshold voltage in a pMOS transistor

In that regard, authors of [65] proposed fast yet accurate modeling of NBTI-induced delays at the gate level.

First, technology and environment dependent curve of the threshold voltage shift as a function of the transistor's gate input signal probability $\Delta V_{THp}(P_z)$ has to be obtained at the transistor level. Then, technology and environment dependent curves of the gate delay degradation as a function of the threshold voltage shift $\Delta t(\Delta V_{THp})$ for each gate type in the netlist (e.g. INV, 2NAND, 2NOR) assumed for gate-level implementation.

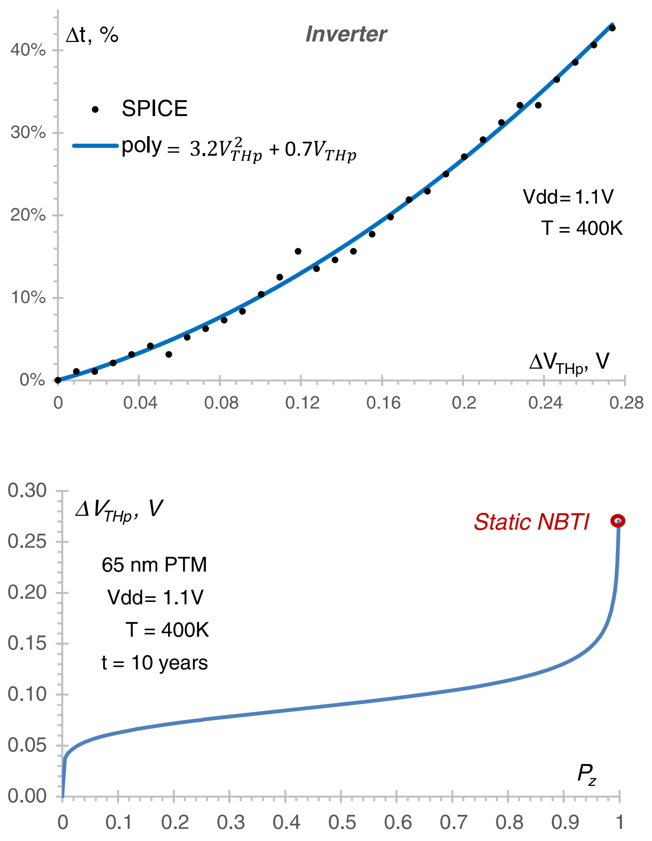


Figure 4.2: Threshold voltage shift V_{TH_p} as a function of signal probability P_z (up). Gate delay increase Δt dependency on voltage threshold shift ΔV_{TH_p} in an inverter gate (down).

In the NBTI effect analysis, a reaction-diffusion (R-D) predictive model for dynamic NBTI is used [70], [72]. This model predicts the long term threshold voltage V_{THp} degradation due to NBTI at a time $t > 1{,}000s$ at high frequencies [70]. It captures the dependence of NBTI on a gate input signal probability P_z (probability that the related pMOS transistor is under stress) in addition to its dependence on other key process and design parameters as presented in [70]. The values of the involved technology and environmental parameters can be summarized by a parameter γ in the following form:

$$|\Delta V_{THp}| = \gamma(\frac{P_z}{1 - P_z})^n \tag{4.1}$$

Note that Equation 4.1 is valid only for dynamic stress, as ΔV_{THp} becomes infinite when P_z reaches the value 1. Therefore, the upper limit of ΔV_{THp} is defined by static

NBTI models [70]. Equation 4.1 represents a convenient mathematical function of the threshold voltage V_{THp} degradation dependence on the signal probability for the gate input signal $P_z(x_i)$ of a pMOS transistor. In the equation, $n = 1/6$ represents the variety of the dominant diffusion species (H or H_2) expressed by the time exponent parameter and $\gamma = 0.0904$ represents a parameter that incorporates the selected technology and environmental variables. In Fig. 4.2 (up), the corresponding dependence is illustrated for PTM 65 nm technology [72] after 10 years of NBTI-induced degradation at constant temperature $T = 400K$ with supply voltage $V_{DD} = 1.1V$. The calculated value of ΔV_{THp} for static NBTI is $0.27V$. The model allows fast estimation of NBTI-induced V_{THp} shifts.

A set of SPICE simulations for each logic cell is used to create a polynomial curve to model the gate delay degradation (see Fig. 4.2) (down):

$$\Delta t_{gate} = \lambda * \Delta V_{THp}(X_i) + \mu * (\Delta V_{THp}(X_i))^2 \tag{4.2}$$

Here, Δt_{gate} is the gate output delay increase (in percentage) compared to the nominal gate delay, $\Delta V_{THp}(x_i)$ is the change of V_{THp} for the stressed pMOS transistor at the gate input x_i, while λ and μ are technology dependent constants. For example, in the current experimental setup λ and μ parameters are set to 0.7 and 3.2 for the INV gate. In case a logic gate consists of multiple cascaded pMOS transistors, both their physical location relative to the output node and the combination of $0 \rightarrow 1$ output transition impact the gate delay degradation. Each combination of gate input values is modelled by different values of the constants λ and μ in Equation 4.2. For an alternative technology and different parameters such as temperature and supply voltage, additional SPICE simulations are required to obtain the curves for modelling the gate-delay degradation.

4.2 Proposed approach: analysis and mitigation

To evaluate the aging effect based on the NBTI model introduced in the previous section a set of steps has to be performed. As a pre-processing step complex gates in the design have to be flattened or the design has to be synthesized to gate-level including only FFs and three types of inverting gates – NAND, NOR, INV. In that way computationally demanding SPICE simulations, curves and parameters have to be obtained only for these 3 types of gates. The first step is modeling the threshold voltage shift as a function of the transistor's gate input signal probability $\Delta V_{THp}(P_z)$ (Eq. (4.1)). The next step consists of obtaining a polynomial function that captures the dependence of gate delay degradation on the threshold voltage shift $\Delta t(\Delta V_{THp})$ for NAND, NOR and INV gates as described in [65] and summarized in Section 4.1. Intensive PSPICE simulations are run for the selected technology, electrical and environment parameters. Then, the design needs to be simulated to obtain signal probabilities of each gate input. Next, these signal probabilities are mapped to the curve parameters that were obtained in the first two steps to calculate NBTI-induced gate delays. In the final step, paths between FFs and primary inputs/outputs are extracted to find the critical one after aging. Since the design consists of inverting gates, when a path is activated, all gate inputs on the path will transition to another value. To find aged delay of a path, all gate delays on the path must be summed up. If the output transition of a gate is $0 \rightarrow 1$, NBTI-induced delay must be added to the nominal delay of the gate. Otherwise, only nominal delay of the gate is used when calculating path delays.

RSNs are interesting from the architectural point of view. Since an RSN is a dynamically reconfigurable network that provides a means for creating a hierarchy, organizing set of TDRs can be performed in numerous ways. It is always a question of trade-off, since this decision may depend on the frequency of access to a certain instrument and overhead in terms of time (clock cycles) required to perform any access in general. Simple restructuring of the network was performed by organizing complete set of TDRs in such a way that they can be accessed individually through serially connected SIBs as shown in the case study (Fig. 4.5). A SIB may provide access to a scan segment containing additional SIBs, thus deepening the hierarchy. Having a regular structure such as a series of SIBs alleviates identifying the critical logic path(s). It is always the one(s) leading to the FFs related to the longest register. The former, however, for accessing certain TDRs may require multiple reconfiguration operations, while in the latter one, only one configuration cycle is required to assert/de-assert a SIB and include/exclude corresponding TDR to/from the active path. On the other hand, serially connected SIBs produce overhead since Segment Insertion Bits (configuration bits) always belong to the active path. Removing hierarchy shortens logic paths in general and therefore reduces nominal delay. However, as it has been confirmed experimentally, it does not impact significantly the aging-induced delay.

As the structure of the RSNs can be quite regular, some paths share, i.e., they traverse the same gates and the same signal lines. Furthermore, it is common that for each of those paths, the remaining parts have the exactly same structure where the gates of the same type are encountered in the same order. As a consequence, an abundant number of paths has the same nominal delay and ages in the same mode.

To perform mitigation, an algorithm creates a list of most critical registers, i.e. sorts the registers in a descending order based on the value equal to the sum of the hierarchical level of the segment they are positioned in and their length since it impacts the fan-out of some gates that belong to the path between primary input or flip-flop output and the input of the flip-flop in the shift stage of the corresponding register. Further on, a function described in Algorithm 12 is called to generate the set of configurations for reaching desired register(s). From the internal network model, it is possible to find controllable module providing access to the particular scan segment and configuration bit(s) corresponding to the aforementioned module. *configureMux* is called recursively saving necessary states and updating the list of registers to be accessed removing the ones already encountered while accessing the current one.

Algorithm 12 Listing programmable modules (SIBs, SMs) and corresponding states to be applied in order to reach particular TDR

```
function generateConfigurations($gen$, $len$, $i$, $d$)
    $tdr \leftarrow regList(0)$
    $mux \leftarrow$ getMux($tdr$)
    while $mux \neq null$ do
        $currentMuxEnc \leftarrow mux$ current configuration
        $regSegEncoding \leftarrow tdr'$s segment encoding
        if $currentMuxEnc \neq regSegEncoding$ then
            configureMux($gen$, $mux$, $regSegEncoding$)
        putOnPath($mux$)
        $index \leftarrow index + 1$
        $tdr \leftarrow regList(i)$
        $mux \leftarrow$ getMux($tdr$)
```

One of the advantages of the RSNs is that the mitigation can be performed in parallel while executing required operations without affecting the final result. Of course, the overhead of such operations exists in terms of additional clock cycles required to access targeted registers and perform read/write operations.

4.3 Case study

The following case study illustrates how the most critical logic path at the gate level of the RSN was identified and how the effect of NBTI-induced aging in RSNs was analyzed. The mitigation technique is also applied to this example and obtained results are reported.

The considered RSN, shown in Fig. 4.3, is motivated from an automotive context and it consists of two sub-networks. The first one has four TDRs - R_1 to R_4 which are placed hierarchically behind 4 SIBs - SIB_1 to SIB_4 and are accessed less frequently. The second section contains registers that are to be accessed more often. It includes three TDRs - R_5 to R_7, two of which are placed behind SIB_5 and SIB_6 located on one input segment of the ScanMux (SM). The remaining register R_7 has been placed directly on the second input segment of the multiplexer. There are no remotely controlled modules, i.e, all of them are controlled in-line. As it has been outlined previously, the function of these instruments may vary since they can be used for monitoring (sensors), debug and calibration/configuration, or BIST control.

For the circuit shown in Fig. 4.3, a workload was created containing series of operations to change the network configurations and read, i.e., write to/from the registers R_5, R_6 and R_7. In the gate-level design the critical (longest) NBTI-degraded path was identified. It corresponds to the path having the longest total delay consisting of the nominal gate delay and NBTI-induced delays Δt for the gates along that path.

Gate-level description of the circuit shown in this case study contains 695 gates, while the total number of logic paths sums up to 2238. Fig. 4.4 shows the path with the largest nominal delay and is at the same time the longest NBTI-degraded path for the workload provided. When the structure of the circuit is taken into account some regularities can be observed. The longest path starts with the top level signal *SEL* and it passes several gates (U20, U19, U18 and U17) up to the *tosel_SIB4* for the SIB_4 module. This signal is used to gate signals *CE* and *SE* responsible for allowing capture, i.e., shift operations. Gates *R4xU138* and *R4xU137* have a fan-out since the signals at their output are used to control single FFs belonging to the same register R_4. In this circuit the register R_4 has the length of 13, and therefore consists of 13 FFs (the shift stage), gate R4xU138 has a fan-out $13 + 1 = 14$, while gate R4xU137 has a fan-out of 13. The section within the borders is a source of the signals and is shared between logic paths leading to the shift flip-flops in a register. The signals belonging to the section out of borders in Fig. 4.4 lead to the second FF in R_4. They are in control whether the same value is kept in the flip-flop if all operations are disabled, or the value from the preceding FF should be stored, or the

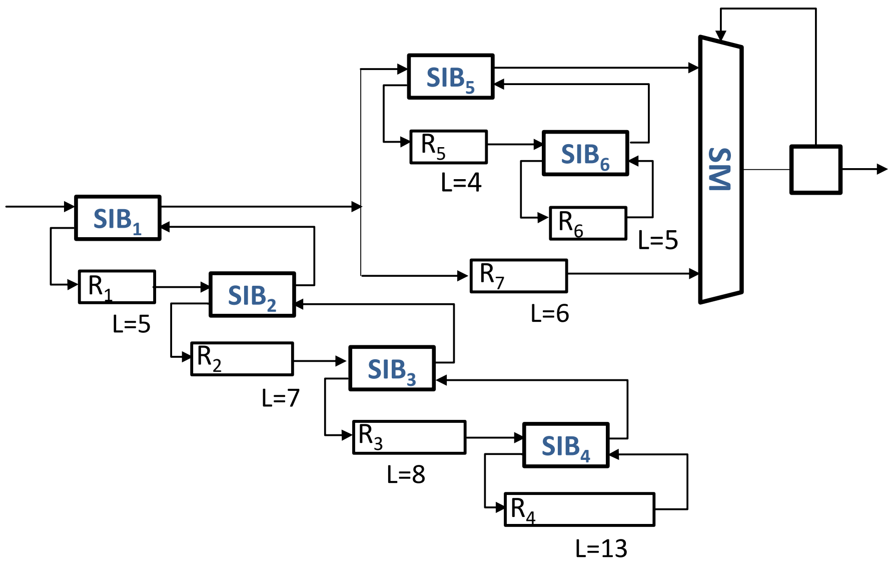


Figure 4.3: Case-study RSN with hierarchy levels - one SM, six SIBs and seven TDRs

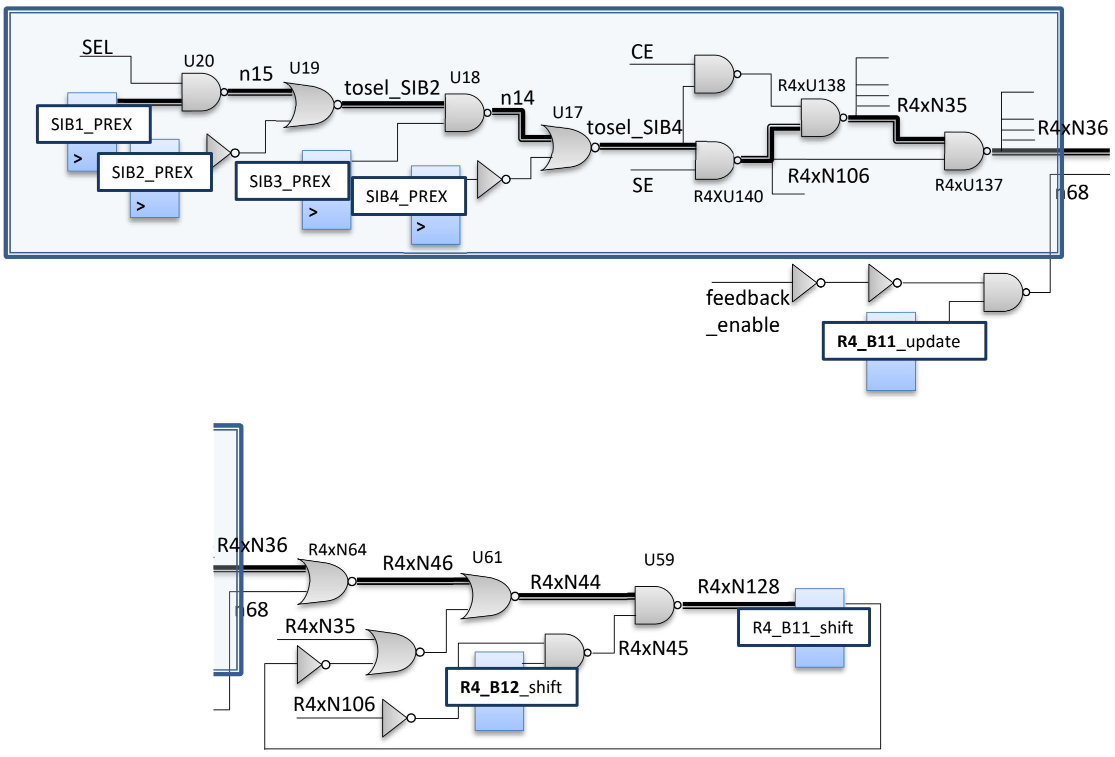


Figure 4.4: Gate-level schematic of the longest logic path in the case-study RSN

value from the update stage should be captured since the feedback functionality is used. The identification of the longest path takes into account the hierarchy levels and the length of the registers.

For circuits containing registers with a large length, i.e., with the high number of FFs, some gates may have a high fan-out. In such case, the synthesis tool may introduce additional gates such as buffers (double inverters) to reduce the overall load on the driving gate and decrease the transition time of the net.

For the longest path, the nominal delay is 86.67 time units, while the total delay after estimating NBTI-induced delay equals to 103.63 time units. Therefore, the delay increase given in percentages is 19.58%. After applying the mitigation technique as already discussed in Section 4.2, the overall delay after NBTI-degradation is 92.5 time units, thus resulting in only 6.73% of delay degradation, which is significantly less with respect to the value obtained when applying the original workload.2

4.4 Experimental results

The results are obtained on two networks from the ITC2016 set of benchmark networks [45] - Mingle and N17D3. For both benchmarks, a flattened network with a separate SIB for each of the registers is generated. The workload is adjusted preserving the access order, set of activated instruments and written values from the workload created for original networks. In Table 4.1 for each of the considered benchmarks and its flattened version the following data are reported: nominal delay (t_{nom}) and total delay ($t_{nom} + \Delta t$) with increase in percentages before and after mitigation. To show its effectiveness, the ratio of the delay increase in percentages is used, before and after mitigation. For example, in Mingle network, the NBTI-aging effect is reduced by 2.1 times, since the delay increase changed from 19.5% to 9.4%.

Synopsys Design Compiler tool was used for synthesising the circuit at the gate-level from the description in RTL (VHDL) with the flattening of the hierarchy and without any optimization. For this purpose, 65 nm technology library has been chosen with imposing certain constraints primarily related to the choice of primitive gates and flip-flops (2-input NAND gate, 2-input NOR gate, inverter gate, flip-flop with the reset functionality). The aforementioned comes from the limited availability of data regarding the model and aging characterization (65 nm PTM). For reporting results on different technology nodes the same procedure from Section 4.1 has to be repeated. Open-source tool zamiaCAD [73] has been used for simulation of the design. Simulation results are used to record the signal probabilities of the gate inputs. All calculations (nominal/NBTI-induced gate delays, path extraction and their delays) are automated in zamiaCAD using Python scripts.

Table 4.1: Experimental results

	Original circuit					Flattened hierarchy				
Network	t_{nom}	$t_{nom} + \Delta t$	t_{nom} (mit.)	$t_{nom} + \Delta t$ (mit.)	Decrease	t_{nom}	$t_{nom} + \Delta t$	t_{nom} (mit.)	$t_{nom} + \Delta t$ (mit.)	Decrease
Mingle	125.67	150.21 (19.5%)	125.67	137.54 (9.4%)	2.1	104.33	127.35 (22.1%)	104.33	115.13 (10.4%)	2.1
N17D3	126.67	150.76 (19.0%)	134.33	144.22 (7.4%)	2.6	115.33	135.86 (17.8%)	115.33	128.13 (11.1%)	1.6

Regarding the results, NBTI-critical path for the N17D3 network changes after mitigation. Before altering the workload, critical path did not correspond to the path with the largest nominal delay, since the latter lead to the register which was

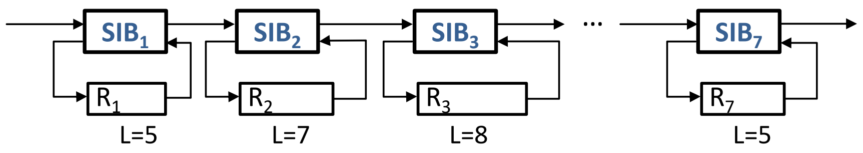


Figure 4.5: Simplified schematic of a SIB module (left) and its symbol (right)

accessed in the workload. After performing/adding additional TDR accesses to reduce the aging effect, the NBTI-critical path becomes the one with the largest nominal delay.

Although the mitigation approach is scalable and applicable for larger and more complex designs (runtime less than one second for the reported benchmarks), simulating design with the test-bench stimuli generated from the workload for obtaining signal probabilities and calculating delay is computationally expensive and therefore time-demanding. The whole framework was run on a modest laptop with a dual-core CPU.

4.5 Chapter summary

The chapter proposes a methodology for assessment and mitigation of NBTI aging induced delays in logic paths within IEEE 1687 IJTAG Reconfigurable Scan Networks. While RSNs are commonly used to provide fault management and embedded instrumentation access, such as safety mechanisms, in advanced safety- and mission-critical electronic systems, a failure in such infrastructure itself has a high severity. The methodology is based on a scalable hierarchical (transistor-to-architecture) modelling of the NBTI impact on timing-critical logic paths in RSN implementations. The evaluation implies analysis of gate input signal probabilities based on the configurations and test data selected for the RSN infrastructure. The details of the methodology are demonstrated by a case study on an example RSN and the feasibility and efficiency are validated by experiments on a subset of ITC2016 RSN benchmarks. The experimental results demonstrate that RSNs can be impacted by significant NBTI-induced logic path delays and a simple proposed mitigation technique can reduce such delays up to 2.6 times. The future work is aimed at a comparative analysis of aging in the RSN gates and the functional part of the circuit.

Chapter 5

Post-silicon validation

In previous chapters of this thesis new approaches to test and diagnose permanent faults have been introduced as well as the analysis of the NBTI-induced aging effect on logic paths inside an RSN. However, the correct operation of IJTAG-compliant infrastructure is a product of many aspects and components including the actual hardware on the chip, the respective standard descriptions, such as ICL (Instrument Connectivity Language) and PDL (Procedure Description Language) files as well as the software used to import the descriptions and control the hardware.

The importance of the problem is being escalated by previous experience of the electronics industry, which suffered from the inconsistency between description files and actual hardware implementation of an earlier similar standard: IEEE 1149.1. In surprisingly numerous cases, the BSDL (Boundary Scan Description Language) descriptions did not match the actual implementation of JTAG features in silicon. Most of those mismatches were caused by simply non-matching revisions of the silicon and the BSDL, but of course a certain number of problems were related to bugs and design errors in hardware. Even if the error-checking is performed before tape-out, it does not necessarily imply that the silicon will work or that the ICL matches the actual silicon implementation. Independent of particular reasons causing such mismatches, the task of proving full compliance between the silicon and the documentation is not trivial. Taking into account the fact that the infrastructure described by IEEE 1687 is certainly more complex than classical Boundary Scan, ensuring its correct operation is an important research topic.

The focus of this chapter is on the problem of checking the equivalence between the silicon implementation of IEEE 1687 RSNs and their respective ICL descriptions. The proposed method assumes that the former is a black box and the latter plays the role of specification, while no other information about the target system is available. Although observability of signals in simulation (for pre-silicon verification) is exceptional, this is unfortunately not the case when accessing the read device through its interface, i.e. we can only apply stimuli and observe responses through scan input and output ports.

Previous work that addresses this problem is very limited. The problem's general definition along with a trivial algorithm for simple RSNs was first proposed in EU FP7 BASTION project report [74]. An important contribution of that work was in defining three levels of validation thoroughness with respect to required test access and effort. At the base level ("Level 0") the RSN infrastructure is validated by checking scan chain length and capture values in various configurations ensuring that every instrument is correctly accessible.

The main contribution of this part of the thesis is twofold. First, a comprehensive fault model defined as a set of mismatches between the ICL description and the silicon implementation is introduced. Second, for a subset of mismatches falling into Level 0 category, a universal method and a tool of their detection based on observing the length of the active scan path is proposed. In addition, the mismatches are categorized with respect to the level of detection difficulty providing a list of those undetectable by the method. Experimental results based on the set of ITC'2016 RSN benchmarks [45] demonstrate that the proposed approach is broadly applicable as well as that the test tool is able to generate the sequences for detecting all target mismatches. The proposed validation tool is a part of an ecosystem of IEEE 1687 benchmarks and tools [75].

5.1 Proposed "black-box" approach to post-silicon validation

The proposed methodology is based on previous work focused on generating efficient patterns to perform end-of-manufacturing test for RSNs [76]. The detection mechanism introduced first in [25] and then adopted in several works [27], [28] has been modified to reduce significant overhead and has been made more suitable for addressing the presented problem.

5.1.1 Mismatch model

Well-established metrics exist for post-manufacturing tests (single-stuck-at coverage, transition fault coverage) and experimental results have demonstrated the effectiveness of such metrics. Although pre-silicon verification metrics are less standardized (syntactic (code coverage) and semantic (covering assertion goals)), metrics for post-silicon validation are still the subject of research. The list of considered mismatches was created after analyzing the literature and taking into account that the source of a mismatch is usually confined, such as a typo in the specification, or a localized hardware bug. It contains following items:

- TDR mismatches Fig. 5.2

  - A missing register
  - An added register
  - A wrong register of different length
  - A wrong register with the modified functionality
  - Exchanging two or more registers (their position)
- Reconfigurable modules position mismatches Fig. 5.3
  - Exchanging a register with a ScanMux or a SIB
  - Exchanging two SIBs belonging to the same segment
  - Exchanging two ScanMuxes
  - Exchanging a ScanMux and a SIB
- SIB type, SM control and input mismatches Fig. 5.4
  - Exchanging inputs or control lines of the ScanMux
  - Wrong SIB type (pre-SIB to post-SIB and vice versa)
- Wrong ScanMux configurations Fig. 5.5

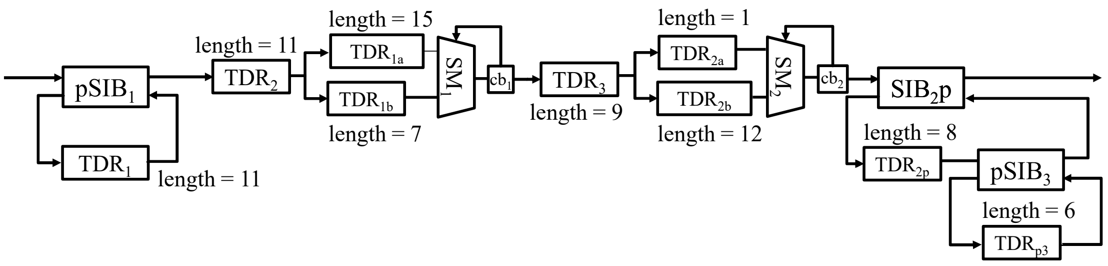


Figure 5.1: Example RSN network for generating a set of mismatches

For the network given as an example in Fig. 5.1, Table 5.1 provides a set of considered mismatches joined by their type.

Relying on the structural information provided by the ICL is sufficient for detecting a missing register at one of the scan segments, since this type of mismatch directly affects the length of the active scan path. An added register has the similar effect on the scan path length although this time it is reduced with respect to the expected one. A wrong register with the different length is equivalent to having a missing and/or added register. Exchanging the registers is being performed not only within the same scan segment but also outside of one domain. This modification can be modelled as having multiple wrong register length mismatches. Since the registers belong to different scan segments, not having both of them on the

Table 5.1: List of considered and injected mismatches for the network from Fig. 5.1

Mismatch Type	N	Mismatch set
SWAP_TDR_SIB	5	$[[pSIB_1, TDR_2],$ $[TDR_2, SIB_{2p}],$ $[TDR_3, SIB2p],$ $[TDR_{2p}, pSIB_3],$ $[pSIB_1, TDR_3]]$
SIB_PRE_POST	3	$[[pSIB_1], [SIB_2p], [pSIB_3]]$
SWAP_CB_CB	1	$[[cb_1, cb_2]]$
ADDED_REGISTER	27	
WRONG_REG_FUNC	9	not considered
WRONG_MUX_CONF	2	$[[SM_1\|0, TDR_{1b}; 1, TDR_{1a}],$ $[SM_2\|0, TDR_{2b}; 1, TDR_{2a}]]$
SWAP_TDR_TDR	1	$[[TDR_2, TDR_3]]$
WRONG_REG_LENGTH	9	all registers
SWAP_TDR_TDR_DD	35	from domain to domain
SWAP_SIB_SM	4	$[[pSIB_1, SM_1], [SM_2, SIB_2p],$ $[SM_1, SIB_2p], [pSIB_1, SM_2]]$
MISSING_REGISTER	9	all registers
SWAP_TDR_SM	4	$[[TDR_2, SM_1], [SM_1, TDR_3],$ $[TDR_3, SM_2], [TDR_2, SM_2]]$
SWAP_SIB_SIB	1	$[[pSIB_1, SIB_2p]]$
SWAP_SM_SM	1	$[[SM_1, SM_2]]$
SWAP_MUX_CONTROL		not modelled explicitly
SWAP_MUX_INPUTS		not modelled explicitly

same path for the first time they are accessed enables immediate detection of this particular mismatch. Fig. 5.2 depicts these situations.

Even though a mismatch of exchanged ScanMux control signals (Fig.5.4c) or inputs (Fig.5.4b) does not have an effect on the active path length it can still be detected. The configuration of the ScanMux is determined by the value in the control bit. The output signal from the update stage, apart from controlling the multiplexer can also be used to gate *shift*, *update* and *capture*. In that case, if upper input is selected, all data shifted at the input is supposed to go through TDR_1 and appear at the output. However, in case of exchanged control signals, although the segment is chosen according to the configuration, all operations are forbidden on that segment and allowed on the other one. Consequently, shifted values will not propagate to the output, resulting in all 0s or all 1s, depending on the value stored in the last scan cell in the selected input segment. Exchanging input connections is an equivalent mismatch and can be analyzed in a similar way. Guaranteeing that all scan segments are accessed at least once ensures that all mismatches of this type are detected.

Another type of considered mismatch is a ScanMux with configurations incorrectly assigned to its input segments. In Fig. 5.5, configurations 00, 01, 10 and 11 result in including registers TDR_0, TDR_1, TDR_2, TDR_3 into the scan path, respectively. In case of a mismatch, chosen registers appear in the different order: TDR_3, TDR_0, TDR_2, TDR_1. If at least one of the segments has a length different than the original one in the same position, the mismatch is detectable comparing the lengths. This type of mismatch also covers the modified order of control bits (cb_0, cb_1-10 with 01).

In case of the wrong SIB module type (pre-/post-), the length and the order of elements on the scan segment remains unchanged when SIB is de-asserted. If there are some control bits in the controlled segment-their order is shifted for one position, while the SIB itself is placed after, i.e., before the elements of the included segment (Fig. 5.4a).

In general, mismatches involving the modified order of TDRs, SIBs and ScanMuxes do not affect the length of the active path when the corresponding segment is included into it (Fig. 5.3). However, detecting them remains possible as long as certain configuration bits do not match original positions. Writing into them to set the desired configuration may result in writing into TDRs or some other configuration bits.

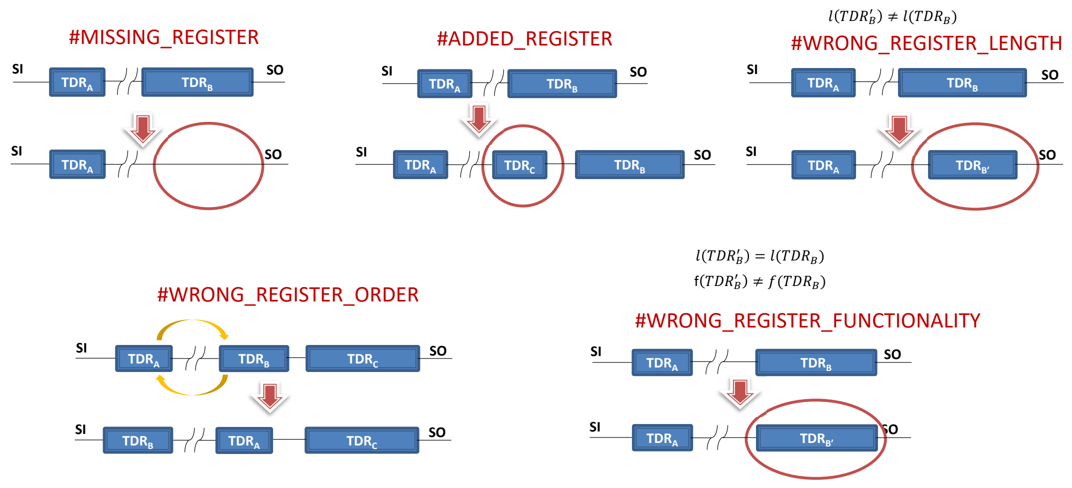


Figure 5.2: Mismatches involving TDRs

5.1.2 Undetectable mismatches

A mismatch is considered to be undetectable if applying whichever legal configuration results in observing expected length of the scan path.

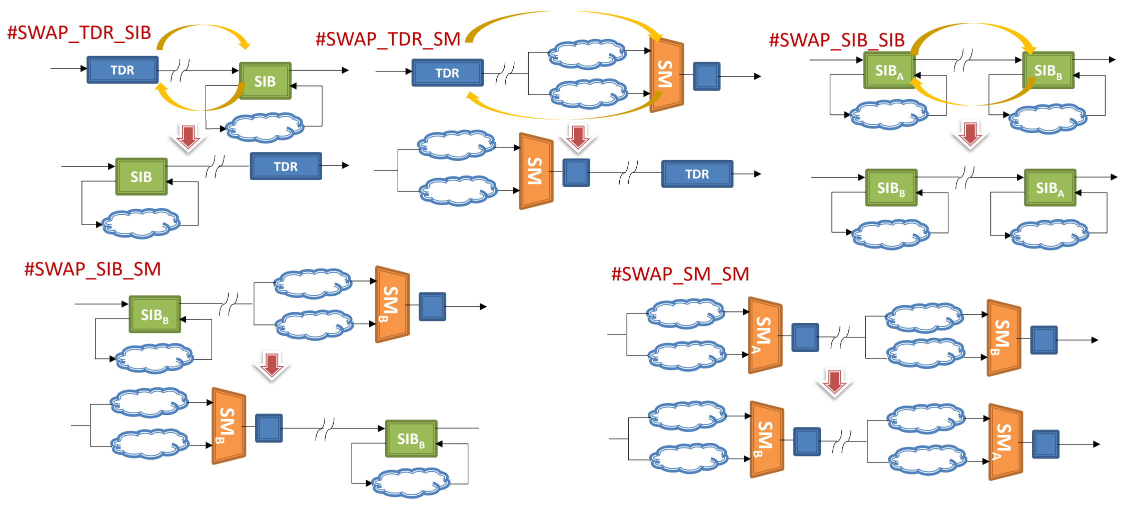


Figure 5.3: ScanMux and SIB position mismatches

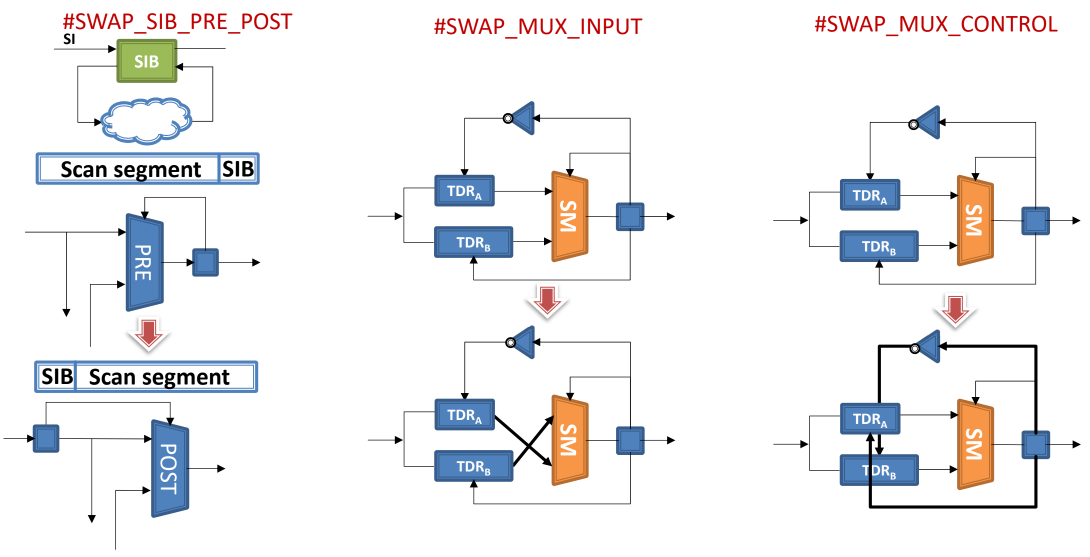


Figure 5.4: SIB type, ScanMux control lines and input mismatch

The modified functionality of the register has been modelled as a permutation of register's bit scan cells and it is equivalent to having permuted connections with the instrument. Since there is no effect on the length of the scan path, this type of mismatch is undetectable. Exchanged position of registers within the same scan segment may not always be detectable. In particular, this is the case if the registers are adjacent or have the same length. Furthermore, if there is not even a single control bit cell located in between, all configurations are properly applied and no mismatch is observed at the output. In the case of exchanging two or more SIBs within the same scan segment, the mismatch is undetectable if they have completely

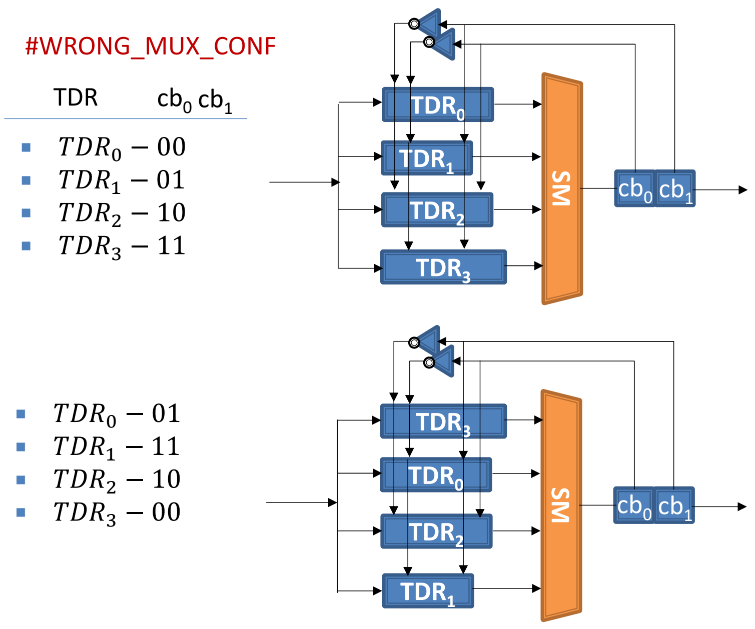


Figure 5.5: ScanMux configuration mismatches

same structure (the position, length and type of modules), except if they provide access to remote ScanMux control bits. Wrong configuration mismatch is also undetectable when all input segments have the same fixed length (only TDRs).

Potentially, by “Level 1” validation [74], some currently undetectable mismatches could be targeted. Correct reaction of instruments on PDL defined-actions has to be verified upon accessing them through performing read and write operations. Furthermore, the presence of undocumented or specially hidden structures can be targeted by “Level 2” validation (phantom detection) [74].

5.1.3 Detection mechanism

The procedure for detecting mismatches is organized as a set of sessions. A session consists of a *configuration* pattern to which an additional sequence of bits is appended. It contains values for defining the state of the network. The appended sequence is used as a key to validate that the expected path is connected between scan input and scan output pins. Configuration sequence of bits has the length of the currently active path. This sequence of bits is shifted into the scan chain, while in parallel the output pin is monitored. If the sequence observed at the output, long as it is the key sequence presented at the input, matches the very same key, it is considered that no potential mismatch could be detected in that session. This is due to the effect of a mismatch which can either corrupt the values of the key loaded into the network (e.g., all 0s or all 1s) or can change its position (postpone it or anticipate it at the output) with respect to the expected one.

Additionally, if a TAP controller is used to control and access the network, its state machine has to traverse *capture* and *update* states, while the *shift* state can be omitted. Therefore, before shifting in the sequence, a capture operation is performed. The values appearing at the output during the shift-in are capture values and can be used to enhance detection capability. The *pause* state, following the *shift* state of the TAP controller can be used to prevent performing update before checking the pattern at the output, thus avoiding undesired effects such as moving the network to an unknown state.

Cost of applying one session is equal to the number of clock cycles (shift operations) as long as the active path increased by the length of a key. It should be mentioned that after performing update to apply wanted configuration, reaching the shift state in a state machine requires certain number of clock cycles (JTAG protocol overhead).

5.1.4 Configuration generation procedure

The mismatch detection is solved as a problem of discriminating between a set of Finite State Machines (FSMs). One FSM is created for the original network without any mismatch present; for each mismatch, an additional FSM is created. Initial state is appended to each FSM, where the state corresponds to the current configuration of the network. Positions of configuration (control) bits with respect to the network's input are being tracked constantly. After applying the transition (reconfiguration operation), the state of each FSM is updated, while the new length (output symbol) is calculated based on the injected mismatch, if any. Additionally, positions of the configuration bits in a new state are calculated and updated accordingly.

Before generating input symbols, during the construction of the internal network model, every reconfigurable element (ScanMux, SIB) in the network is annotated with auxiliary information. First attribute is the hierarchical level (l_c) of the scan segments in which the module is positioned, while the second one is the highest hierarchical level (l_d) of all modules' levels that are positioned within scan segment(s) attached to the input(s) of the current module. In Fig. 5.6b a structural representation of the network from Fig. 5.6a is shown as an example in the form of a tree. A node coincides with the reconfigurable module, while encoding of the segment that is either insertable (SIB) or selectable (ScanMux) is given as a vertex. Nodes n_1, n_2 and n_3 are located at the top level scan segment, while nodes n_{31}, n_{32} and n_{33} are accessible through n_3 (located in its input segment(s)). Nodes n_{11}(1,1) and n_{13}(1,1) provide access only to empty segments and segments that include either TDRs or control bits (registers), which is obviously not the case with the node n_{12}(1,2). This node provides access to the segment at the second level of hierarchy with one node n_{121}(2,2) in it.

The mechanism for generating input symbols is able to determine if the network contains remotely controlled scan multiplexer architecture or all modules are controlled in-line. In the first case, the priority list for accessing nodes is shown in Fig. 5.6c. The n_i node's position in the list is based on the l_d^i value; the precedence is given to the node with the higher values. However, if the nodes n_i and n_j have equal value of the second parameter ($l_d^i = l_d^j$), the position is decided based on the first parameter value l_c. Finally, in case that $l_c^i = l_c^j$ the nodes whose parent nodes are closer to the input are chosen first e.g., n_{11} has precedence in comparison to n_{31}, while n_{31} is put before n_{33}.

The procedure (Algorithm 13) starts from the position of all configuration bits, taking into account only accessible ones when choosing the element from the priority list (Fig. 5.6c). When it is a SIB instance, if it is in a de-asserted state, new configuration sets it to assert, while if it is a ScanMux with no children nodes, one configuration is generated for every input segment in order to include it to the active path at least once. For a ScanMux with some reconfigurable nodes in its input segments the decision which configuration to set is based on the priority list. In that sense n_3 could be a SIB with three serially connected nodes n_{31}, n_{32}, n_{33} located at the same segment, but it could also be a ScanMux with three input segments; in first n_{31}, in second n_{32}, and in third n_{33}. When a segment is included to the active path it is marked as *tested*. If all children nodes for a parent SIB node are marked as *tested*, the next configuration will also de-assert that SIB.

In this work we also considered more complex networks involving remotely controlled scan multiplexer architecture. They are more difficult to manage in terms of module's controllability and observability: corresponding control bits have to be part of the active path in order to set desired configuration, while additional reconfiguration operations are required to include the module itself into the active path. Therefore, an algorithm (Algorithm 14) implemented with two recursive functions CONFIGUREMUX and PUTONPATH provides input symbols for guaranteeing that all scan segments are accessed at least once, while also detecting the full set of considered, detectable mismatches. The order in which multiplexers are provided is obtained using the same rule as in the previous algorithm with the difference that is performed once, globally, taking all multiplexers into consideration (Fig. 5.6d). To give an example of testing SM_1 from Fig. 5.7 that is in configuration "0" and on the active path (01 input segment of SM_0) a set of figures is provided. SM_1 needs to be brought to its "1" configuration which was immediately possible with modules controlled inline (such as SM_0). On the other hand with remotely controlled modules, such as SM_1, their configuration bit has to be accessed. CONFIGUREMUX will compute set of configuration transitions to do so:

- SM_0 needs to be brought to "11" state to include $pSIB_1$ (Fig. 5.8)
- once $pSIB_1$ is on the active path, it has to be asserted (Fig. 5.9)

- as bit cb_1 is now accessible it needs to be set to 1 (Fig. 5.10); configuration of SM_1 is now as wanted; still, it has to be included into the active path.

Finally, PUTONPATH will change the configuration of the SM_0 to 01 as this is enough to include to have SM_1 included into active path (Fig. 5.11).

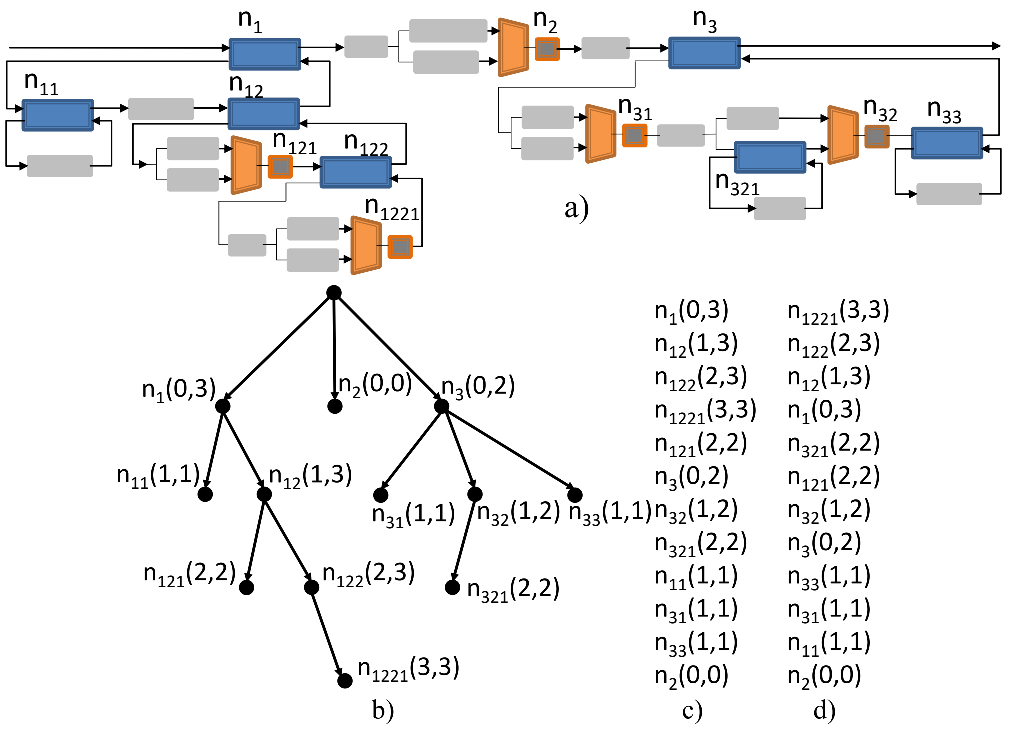


Figure 5.6: Hierarchical information on network's nodes

5.2 Experimental results

We developed a prototypical tool, able to build an internal simplified RSN model by reading the ICL description of the network. After generation of selected mismatches, pattern generation is run. Upon completion, a report is generated with the list of applied configurations, set of covered mismatches and those which are considered as undetectable.

Experimental results using the proposed mismatch model and pattern generation algorithms are reported for the sub-set of ITC2016 benchmark networks, since the tool currently does not support all the constructs. Some synthetic networks from the same set [45] have been translated from an internal XML description to

Algorithm 13 Deterministic algorithm for in-line RSN architecture

```
function GENERATEINPUTSYMBOL($prevState, state$)
    $i \leftarrow$ SIZE($bitBuffer$)                          ▷ number of control bits
    $genState \leftarrow$ ACCESSIBLE($state$)                  ▷ visible control bits
    while $i \geq 0$ do
        $mux \leftarrow$ GETMUX($StateVars, i$)                ▷ $mux$ corr. to $i$
        if ACCESSIBLE($i$) then                           ▷ control bit on act. path
            CHECKMUX($0, mux$)                            ▷ update $mux$ test status hier.
            if SIB then                                   ▷ $mux$ is SIB type
                if $bitBuffer[i]$ A then
                    if HIERTESTED($mux$) then
                        $bitBuffer[i] \leftarrow$ D               ▷ close SIB
                        SETTESTED($mux$)                  ▷ mark as tested
        $curL \leftarrow mux$ segment hier. level($l_c$)          ▷ 1nd parameter
        $deepL \leftarrow mux$ deepest hier. level($l_d$)         ▷ 2st parameter
        if ScanMux or (SIB and ($bitBuffer[i]$ D
        or ($bitBuffer[i]$ A and ISTESTED($mux$))) then
            if $deepL \geq maxD$ and $\neg$ISTESTED($mux$) then
                if $curL \geq maxC$ then
                    $sMux \leftarrow mux$                         ▷ save the multiplexer
                    update levels ($maxD$)
        $i \leftarrow i -$ SELECTIONCELLSIZE($mux$)
    if $\neg$ISTESTED($sMux$) then                          ▷ decide how to conf. $mux$
        if ScanMux then                                   ▷ traverse ScanMux inputs
            $bitBuffer \leftarrow$ apply next encoding
            if last encoding then
                SETTESTED($sMux$)                         ▷ mark as tested
        else
            if $bitBuffer[i]$ D then
                $bitBuffer[i] \leftarrow$ A
            else
                if ($mux, l_c = l_d$) then
                    SETTESTED($sMux$)                     ▷ mark as tested
```

the ICL format and can be found at the ecosystem's website. [1].

The experimental results are given in Table 5.2. For each of the benchmarks in column 1, number of generated configurations is reported in column 2. Column 3 gives the total cost in clock cycles for applying the generated sequence. The key length has been set to 32, while 5 clock cycles are added as JTAG overhead to each

[1]https://gitlab.com/IJTAG/benchmarks/tree/master/ICL

Algorithm 14 Generating configurations for remotely controlled RSN architecture

```
function GENERATECONFIGURATIONS($gen$, $len$, $i$, $d$)
    $mux \leftarrow muxList(0)$
    while $mux \neq null$ do
        $currentConfEnc \leftarrow mux$ current configuration
        for $encoding$ all mux encodings do
            if $currentConfEnc \neq encoding$ then
                CONFIGUREMUX($gen$, $mux$, $encoding$)
            PUTONPATH($mux$)
        $index \leftarrow index + 1$
        $mux \leftarrow muxList(index)$
```

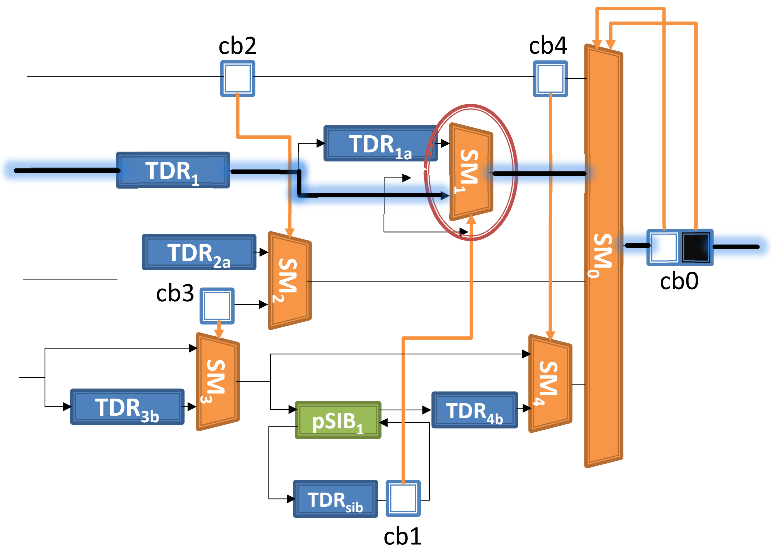


Figure 5.7: Testing SM_1 in steps: SM_1 with its input segment 0 on active path

of the sessions. Furthermore, the time required by the tool written in Java to apply the algorithm is given in column *Runtime*. Columns 5 and 6 report on total number of considered faults (excluding implicit ones) and the total number of *undetectable* faults, respectively. It is worth noting that in all cases applying generated sequence resulted in achieving full coverage.

5.3 Chapter summary

Reconfigurable Scan Networks provide flexible instrumentation access through dynamic reconfiguration. To ensure there is no mismatch between prototypical device and initial specifications, product life-cycle requires performing (post-silicon)

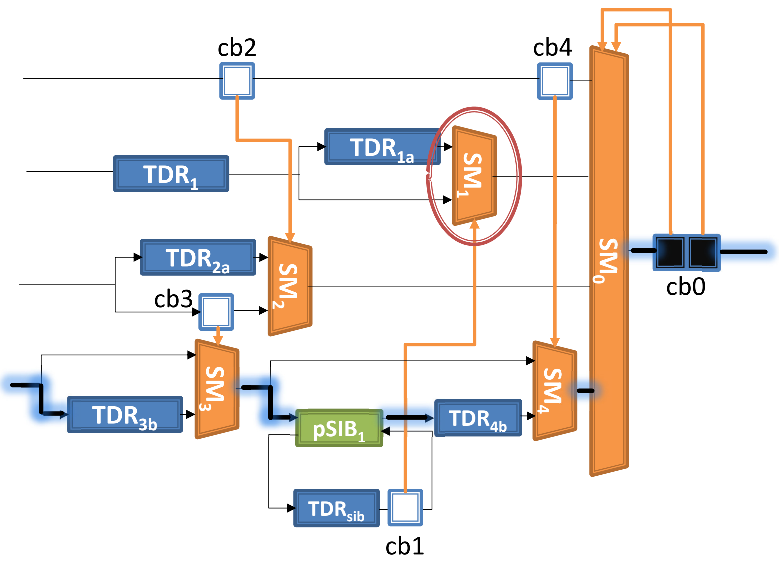


Figure 5.8: Testing SM_1 in steps: accessing its configuration bit cb_1; $pSIB_1$ has to be included into the active path

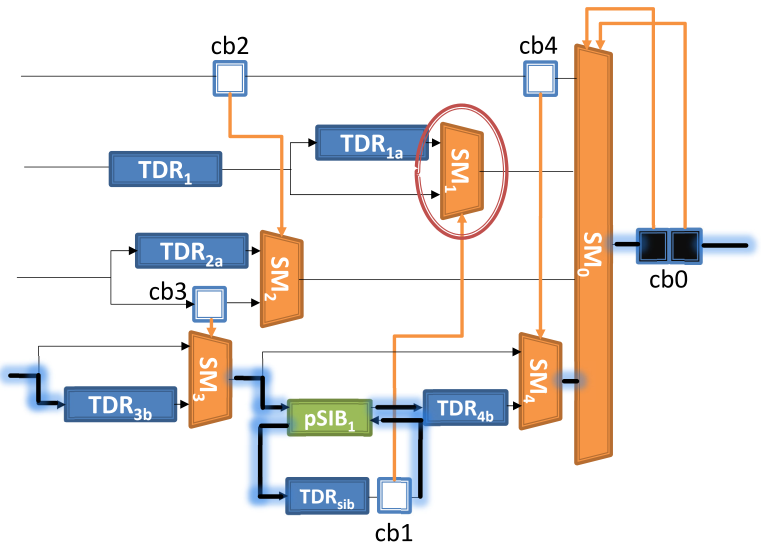


Figure 5.9: Testing SM_1 in steps: accessing its configuration bit cb_1; $pSIB_1$ has to be asserted

validation before going into the mass production. Such additional effort prevents rendering the whole infrastructure inoperable and avoids enormous re-design costs. In this chapter a mismatch model for post-silicon validation of RSNs was proposed.

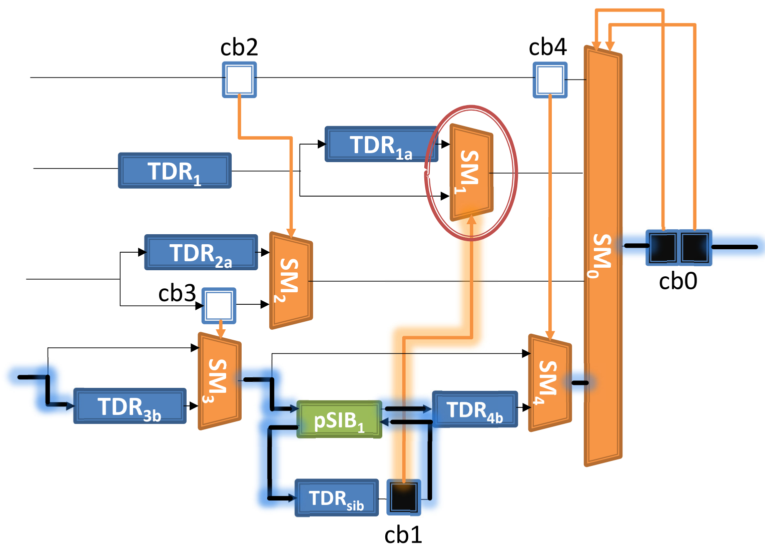


Figure 5.10: Testing SM_1 in steps: setting cb_1 to 1

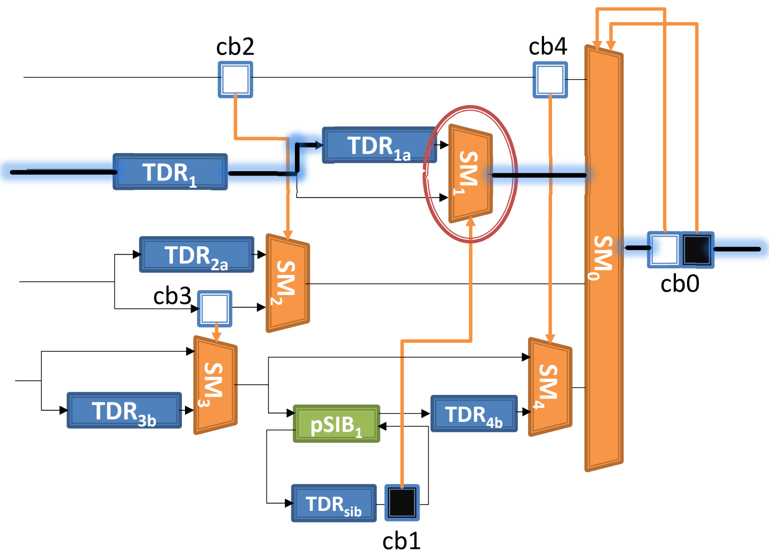


Figure 5.11: Testing SM_1 in steps: putting back SM_1 to the active path; input segment that is now selected is 1 (TDR_{1a})

Furthermore, two algorithms have been developed for generating configuration patterns to detect the set of considered mismatches. The ITC2016 benchmark networks were used to evaluate the proposed methodology. It was found that in all cases full detection coverage has been reached. For the detection procedure based on the active path length comparison, the tool is able to generate a list of undetectable mismatches. Furthermore, mismatch model is easily extendable, due to the nature

Table 5.2: Post-silicon validation experimental results

Network	Number of conf. steps	Total cost [cc]	Run-time	Total mismatches	Undetectable mismatches
Mingle	32	4,437	1s	124	29
TreeBalanced	62	19,391	9s	1,423	47
TreeFlat_Ex	102	16,446	10s	2,725	149
TreeUnbalanced	44	150,277	4s	874	50
a586710	106	1,588,162	7s	1,029	186
p22810	518	162,668	8m	36,371	851
p34392	310	2,853,219	100s	8,250	1,377
p93791	1,864	76,688,262	11h	203,832	21,342
q12710	50	54,760	4s	491	8
t512505	287	176,506	1m	10,246	557
N132D4	96	100,629	47s	18,140	492
N17D3	17	3,464	1s	553	11
N32D6	29	850,906	5s	1,290	4
N73D14	55	3,361,858	7s	4,836	21
NE600P150	432	1,685,759	37m	319,871	5,818
NE1200P430	854	10,696,840	4h	1,337,343	11,303

of the problem and internal model of the network extracted by the tool.

Chapter 6

Simulation-based equivalence checking between IEEE 1687 ICL and RTL

The standardization and portability the IEEE 1687 Standard introduced was supported by two description languages: the Instrument Connectivity Language (ICL) and the Procedure Description Language (PDL). The two languages together allow for instrument access procedures to be written once at the IP level, and then be applied regardless of where the IP was integrated and how many times was instantiated.

Role of the Instrument Connectivity Language (ICL) is to describe the new systems in a tool-friendly manner. This means that a given system will have two different descriptions: the RTL, used in the design flow to obtain the final circuit, and the ICL, to describe connections in high-level manner, between the IJTAG scan circuitry and the instrument ports, without instrument functionality. Depending on the company's internal design strategy, ICL and RTL could be developed in parallel starting from the same high-level specifications, ICL could be extracted from the RTL or vice-versa. In all cases, this implies to have two different descriptions of the same system: any incoherence between the two models might cause serious problems. This is especially true for a new standard like IEEE 1687-2014 : ICL can be complex and engineers are still learning it, making human error extremely likely.

In this chapter, an automated and reliable method for verifying equivalence between ICL and RTL descriptions is proposed. After giving a short introduction to IEEE-1687 ICL in Section 6.1, the approach is presented in Section 6.2. Section 6.3 will provide experimental results based on the ITC16 benchmark suite [45].

6.1 ICL

The Standard document states that ICL's purpose is "to describe the elements that comprise the instrument access network as well as their logical (though not necessarily their physical) connections to each other and to the instruments at the endpoints of the network" [77]. The connections between IJTAG scan circuitry and the instrument ports without instrument functionality is what essentially ICL supplies. To obtain this result, the language takes an approach quite similar to a "light-RTL": registers and instruments can be instantiated and parametrized, and connected through "ports" and "logic signals". Dynamic topologies can be described using "ScanMuxes", whose truth table is used to select the active path. Overall, ICL provides descriptions to find every IJTAG control and data bit on the chip. ICL designs can also be split into multiple files for easier maintenance and code reuse. The code base can therefore become quite big, and manual verification cannot be trusted : an automated and quantifiable Equivalence Checking tool is the only viable solution.

A fundamental entity in ICL is called a module. As an example, we provide a description for the pre-SIB module in Figure 6.2. Here structural description of the module is given together with the definitions of two scan interfaces: host and client interface. The module consists of one *ScanMux* primitive - SIBmux and *ScanRegister* primitive - SR, whose update stage is used to control the multiplexer. This control register (bit) is placed after the multiplexer since its source (*ScanIn-Source*) is defined as the output of SIBmux. *CaptureSource* and *ResetValue* fields are also specified for the register. SIBmux has two input segments, first used to bypass (client input - *SI*) and the second one to include the segment (host input - *fromSO*).

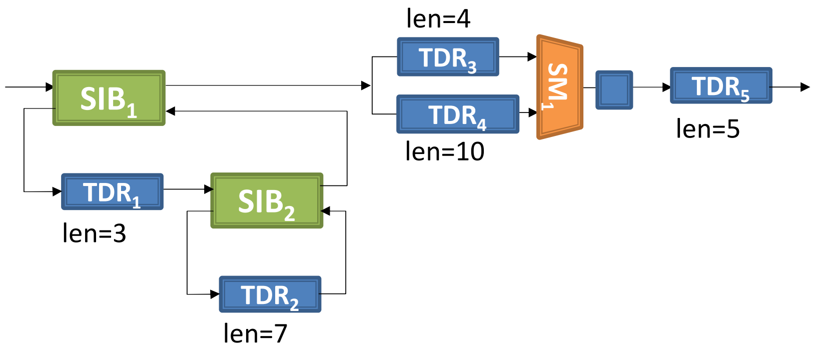


Figure 6.1: 1687 RSN example

A description of the device can contain instantiated modules. In Figures 6.3, 6.4 and 6.5, example network from Fig. 6.1 is partially described. Parameters are used to define the length of the registers (Fig. 6.3). It can be seen that not all ports of the instantiated modules are connected. Since ICL is an abstract language rather than a netlist language, they are considered to be connected implicitly. For example, SEL ports of some instances (TDR1 and TDR2) are produced implicitly, directly from the parent modules (SIB1 and SIB2) (Fig. 6.4). On the other hand, instance TDR3 placed behind the ScanMux has explicitly defined select signal where SM's SEL port is gated with the associated decode of the DO[0:0] signal used to select the active SO signal of the ScanMux (Fig. 6.5).

```
Module SIB_mux_pre
    ScanInPort SI;
    CaptureEnPort CE;
    ShiftEnPort SE;
    UpdateEnPort UE;
    SelectPort SEL;
    ResetPort RST;
    TCKPort TCK;
    ScanOutPort SO {
        Source SR;
    }
    ScanInterface client {
        Port SI;
        Port CE;
        Port SE;
        Port UE;
        Port SEL;
        Port RST;
        Port TCK;
        Port SO;
    }

    LogicSignal toSel_SR_SEL {
       SR[0] & SEL;
    }
    ScanInPort fromSO;
    ToCaptureEnPort toCE;
    ToShiftEnPort toSE;
```

```
    ToUpdateEnPort toUE;
    ToSelectPort toSEL {
       Source toSel_SR_SEL;
    }
    ToResetPort toRST;
    ToTCKPort toTCK;
    ScanOutPort toSI {
        Source SI;
    }
    ScanInterface host {
        Port fromSO;
        Port toCE;
        Port toSE;
        Port toUE;
        Port toSEL;
        Port toRST;
        Port toTCK;
        Port toSI;
    }
    ScanRegister SR {
        ScanInSource SIBmux;
        CaptureSource SR;
        ResetValue 1'b0;
    }
    ScanMux SIBmux SelectedBy SR {
        1'b0 : SI;
        1'b1 : fromSO;
    }
```

Figure 6.2: Description of the pre-SIB as module in ICL

```
Parameter lenR1= 3;
Parameter lenR2 = 7;
Parameter lenR3 = 4;
Parameter lenR4 = 10;
Parameter lenR5 = 5;
```

Figure 6.3: Parameteres in ICL

```
Instance sib1 Of SIB_mux_pre {
    InputPort SI = SI;
    InputPort fromSO = sib2.SO;
}
Instance TDR1 Of WrappedScan {
    InputPort SI = sib1.toSI;
    Parameter dataWidth = \$lenR1;
}
Instance sib2 Of SIB_mux_pre {
    InputPort SI = regR1.SO;
    InputPort fromSO = regR2.SO;
}
Instance TDR2 Of WrappedScan {
    InputPort SI = sib2.toSI;
    Parameter dataWidth = \$lenR2;
}
```

Figure 6.4: SIBs with TDRs in ICL

6.2 Proposed approach

6.2.1 Post-silicon validation approach

Chapter 5 describes an approach for the purpose of validating silicon implementation of the RSN against respective ICL descriptions. The method itself relies on the ICL as only specification, without any additional information available. Since the observability is reduced compared to gate level or RTL level in simulation, the device itself is considered to be a black-box. Therefore, the proposed approach relies solely on applying the stimuli at the input and observing responses at the output.

Although well-defined and experimentally verified metrics exist for post-manufacturing

```
Instance TDR3 Of WrappedScan {
    InputPort SI = sib1.SO;
    InputPort SEL = sel_SR3;
    Parameter dataWidth = \$lenR3;
}
ScanMux SM SelectedBy controlSM.DO {
    0: TDR3.SO;
    1: TDR4.SO;
}
Instance controlSM Of SCB {
    InputPort SI = SM;
}
LogicSignal sel_SR3 {
    SEL \& (controlSM.DO[0:0] == 1'b0);
}
```

Figure 6.5: ScanMux and TDR in ICL

tests and some less standardized semantic and syntactic for verification (pre-silicon), for post-silicon validation they are still the subject of research. Independent on the specific reasons for the existence of a mismatch between the specification and the implementation we defined a fault model containing a set of mismatches of different type: a missing register, an added register, a wrong register length, exchanged position of two modules (TDR, ScanMux & SIB), wrong configuration, exchanged inputs and control lines of the ScanMux, wrong SIB type. Additional constraint taken into account is that the TAP controller is used for controlling the network. Its state machine allows specific order of operations to be executed: either capture shift and then update, in cycles, with the possibility to avoid shift - only capture and then update. Therefore, we organized the procedure for detecting such mismatches as a set of steps. Every step consists of:

- Capture operation
- Shift operation - first a unique key sequence (32/64/128 bits) is inserted into the scan chain. Its purpose is to check the integrity of the scan chain and the length of the active scan path. To continue, a sequence of bits containing the new configuration is inserted while observing the values at the serial output.
- Update operation - to apply the wanted configuration

After analyzing potential mismatches a conclusion has been drawn that their effect is such that the checking sequence is either corrupted - its values are modified, or shifted in time - values are appearing later or earlier than expected. When a segment that includes modules (TDRs, SIBs and ScanMuxes) whose order is modified is included into the active for the first time, it does not have an effect on the length of the active path. However, detecting them remains possible as long as certain configuration bits do not match original positions. Writing into them to set the desired configuration may result in writing into TDRs or some other configuration bits.

The algorithm for generating configurations is deterministic. It starts from the internal network model and the set of considered mismatches. Some of the mismatches are considered detected implicitly with the condition that each scan segment has to be accessed at least once.

The network is modelled as a Finite State Machine (FSM):

- State is represented as the current configuration of the network.
- Output symbol is the length of currently active scan path.
- Input symbol is the bit stream shifted at the input
- Transitions are reconfiguration operations.

Apart for the original, unmodified network, one FSM is created for every mismatch. Such FSMs contain the set of segments that do not match those in the original network due to the injected mismatch. Additionally, the record of positions for all configuration bits is kept. Initial states are generated and set for all FSMs, while consecutive states are being created dynamically.

Reconfigurable modules are listed based on their position in the hierarchy of the network as well as on the highest hierarchical level they provide access to.

Two algorithms for generating configurations were developed in order to support both, networks with all modules controlled in-line and those incorporating remotely controlled configurable modules as well, where the ScanMux and associated set of control bits are either non-adjacent or do not belong to the same scan segment. The latter are more difficult to manage since for a certain network configuration to be applied, multiple reconfiguration operations may take place to first set the desired value(s) of relevant control bit(s) and then include the module into the active path.

6.2.2 Application to RTL Equivalence

In this paper, we propose to apply the same approach to the problem of RTL and ICL Equivalence: starting from the ICL description of the System Under Test we develop a testbench, which thanks to the properties detailed in the previous sub-section, when simulated against the correct RTL description provides a 100% coverage. In case of non-compliance between the RTL and the ICL, coverage will drop as the hypothesis behind the testbench generation are not true: this condition will therefore allow us detection, as will be detailed in the following Section.

6.3 Experimental Results

6.3.1 Setup

To validate the approach, an experimental setup has been devised [45]: for an RSN the following experimental procedure is followed:

1. First the ICL is parsed to obtain the internal network model;
2. From this model, an RTL description of the System-Under-Test is generated, which is correct and coherent with the ICL by construction. It is the golden reference;
3. By applying the method of Section 6.2.1, i.e., Chapter 5 a reference testbench is obtained;
4. The testbench is simulated in Questa®SIM, while enabling the calculation of code coverage and functional coverage;
5. Starting from the Golden Reference (Step 2), a set of erroneous RTL is generated by mutating the model against a set of possible error
6. For each mutated RTL, the testbench gets executed and the coverage gets recorded: the mutation is considered as detected if there is a coverage drop.

In Table 6.1 we report some basic information on the subset of ITC2016 benchmark networks [45], together with experimental results. The networks from the evaluation set differ in the number and type of programmable modules. For each network given in column 1 (*Network*) following information is reported:

- Columns 2 (*SIB*) and 3 (*SM*) - the total number of SIB and ScanMux reconfigurable modules, respectively.
- Column 4 (*Conf. bits*) - the total number of configuration bits in the network
- Column 5 (*Max. depth*) - the maximum hierarchical depth of the network (for SIB-based networks this value equals to the maximum number of nested SIBs, according to [45])
- Column 6 (*Long. path*) - the length of the longest scan path in the network
- Column 7 (*Scan Cells*) - the total length of all scan cells existing in the network
- Column 8 (*TB Gen*) - the time needed to generate the testbench for the given network

- Column 9 (*TB Exe*) - the time needed to execute the testbench for the given network

In all the considered benchmarks, the statement coverage, branch coverage and assertion coverage have reached 100% when simulating the Golden Reference RTL design.

To validate the approach, we selected the following set of possible errors:

- mismatch in one register's length
- wrong position of the controlling register of a SIB's: pre (i.e. ScanMux precedes the control register) - & post (the ScanMux comes after the control register
- Error in the order of scan segments connected to the inputs of scan multiplexers

This is only a subset of possible errors, but from their experience the authors deem it representative of typical human coding. As this selection only impact the experimental validation and not the benchmark generation itself, it would be extremely easy to verify coverage for other error types.

Fig. 6.7 depicts the output of an experimental run for a network being subjected to mutations for a register length, Fig. 6.8 for a network with wrong SIB types and Fig. 6.9 for mixed ScanMux inputs.

For each mutation, represented on the X-Axis, three coverage metrics are given: statement, branch and assertion. In all 63 cases but 2 (replica 50 and 56) at least one of the three types of coverage is under 100%, which is the mismatch detection condition. For this example, detection rate is therefore 88/90=97.8%.

Table 6.1: Benchmark networks list

Network	SIB	SM	Conf. bits	Max depth	Max path	Scan cells	TB Gen	TB Exe
Mingle	10	3	13	4	171	270	2s	5s
TreeBalanced	43	3	48	7	5,219	5,581	11s	16s
TreeFlat_Ex	57	3	62	5	5,100	5,195	12s	40s
TreeUnbalanced	28	-	28	11	42,630	42,630	6s	2m
a586710	-	32	32	4	42,381	42,410	20s	3m
q12710	27	-	27	2	26,185	26,185	5s	16s
N132D4	39	40	79	5	2,555	2,991	52s	95s
N17D3	7	8	15	4	372	462	2s	2s
N32D6	13	10	23	4	84,039	96,158	7s	4m
N73D14	29	17	46	12	190,526	218,869	9s	10m

Instruments are considered to be raw with defined data input and output ports, while the network has been designed with the *feedback* functionality to enable reading out the same values that were previously written. This is the main reason why full toggle coverage cannot be achieved on registers representing TDRs. Additional constraint is that we considered no information is available on which type of instruments are to be integrated or how they are operated.

6.3.2 Results

Table 6.2 resumes experimental results: for each network in Column 1, ranges for three types of coverage (A-Assertion, B-Branch, S-Statement) for each error type are reported in percentage of hits with respect to the total bin number. Furthermore, for each error type the number of detected errors and number of generated errors is given in the ratio form in Columns Det/Tot. Finally, for each network, the last column gives the detection rate taking into account all mutations generated for each error type.

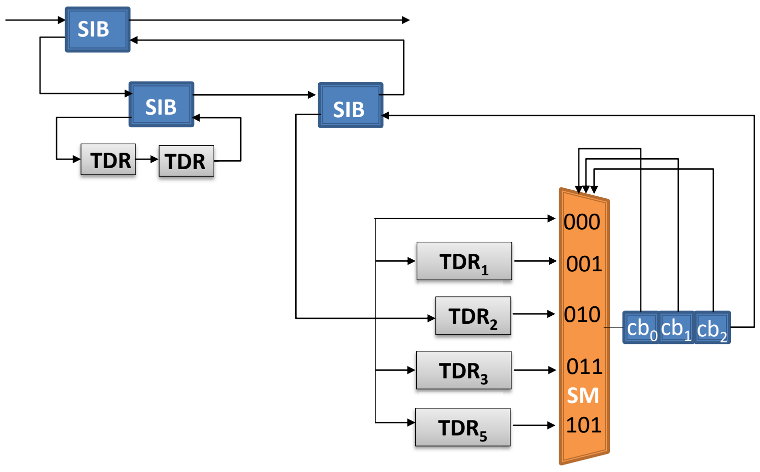


Figure 6.6: TreeBalanced ScanMux with equal length registers

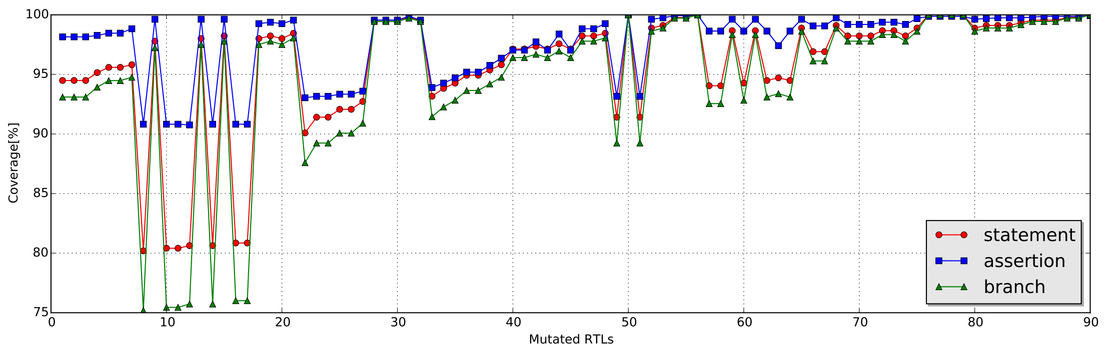


Figure 6.7: Coverage for mutated RTL designs with wrong register lengths – *N73D14* benchmark circuit

The approach provides extremely good performances: coverage is close to 100% in most cases, and execution times are extremely low, making its usage compatible with the Design flow without impacting development times. As of now, testbenches are executed until the end to obtain complete coverage metrics, but it is possible to pinpoint the moment the deviation from the Golden Reference occurs: future developments will focus on this aspect to identify the exact difference between ICL and RTL to help debugging. There are some test escapes, which need to be analyzed in more detail. In these cases, the mutated circuit cannot be differentiated from the original one because of symmetry inside the network. Take for instance network *TreeBalanced*, whose part is depicted in Fig. 6.6: the registers TDR_1, TDR_3 and TDR_5 have the same length, so even though the ScanMux has a selection error, it is impossible to tell them apart. Rather than limitations of our chosen detection algorithm, these escapes are pathological networks that result in untestable topologies, and which should be avoided in the final silicon.

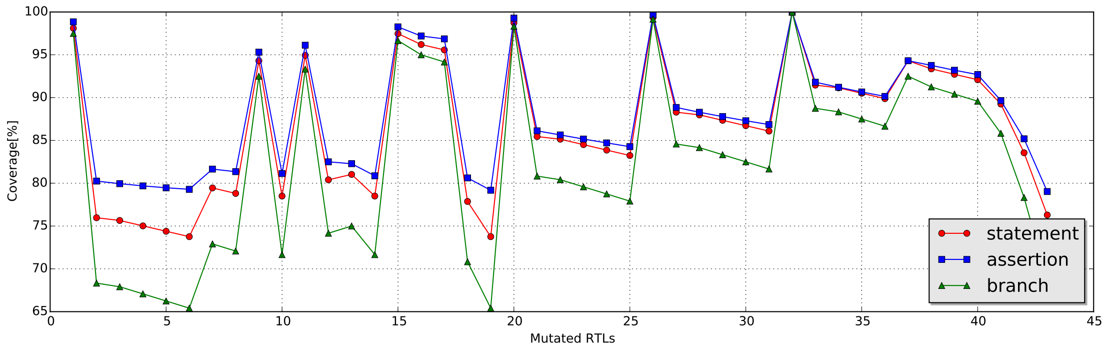


Figure 6.8: Coverage for mutated RTL designs with wrong SIB types – TreeBalanced benchmark circuit

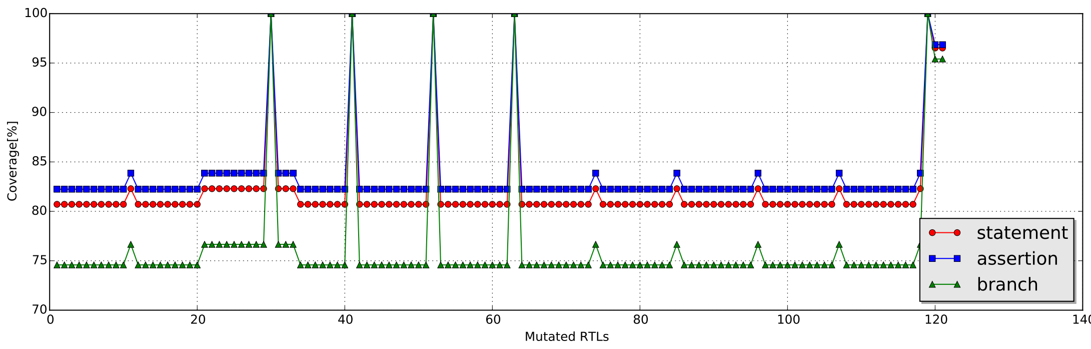


Figure 6.9: Coverage for mutated RTL designs with exchanged ScanMux input segments – TreeBalanced benchmark circuit

Table 6.2: Experimental results on RSN benchmarks for 3 types of mutations showing the range of obtained coverage for mutated RTLs

.

	SIB type mutations				Register length mutations				ScanMux segments mutations				
	Range [%]				Range [%]				Range [%]				
Network	A	B	S	Det/Tot	A	B	S	Det/Tot	A	B	S	Det/Tot	Detection rate
Mingle	58.5-92.2	89.8-100	92.5-100	10/10	60.2-99.2	89.8-100	92.5-100	9/9	59.7-69.4	84.6-93.2	88.7-94.5	3/3	100%
TreeBalanced	79.03-99.96	65.41-100	73.76-100	43/43	79.47-99.88	66.25-100	74.39-100	44/44	82.24-100	74.58-100	80.71-100	116/121	97.6%
TreeFlat_Ex	96.97-99.98	88.41-100	91.14-100	57/57	97.51-99.98	88.71-100	91.38-100	63/63	98.17-100	93.59-100	95.1-100	116/121	97.9%
TreeUnbalanced	84.6-99.92	68.45-100	75.61-100	28/28	85.31-99.92	70.83-100	77.45-100	35/35	-	-	-	-	100%
a586710	-	-	-	-	100-100	100-100	100-100	0/32	-	67.18-96.48	75-97.32	32/32	50%
p22810	?	?	?		?	?	?	?	-	-	-	-	?
q12710	93.89-99.92	85-100	88.72-100	27/27	93.89-99.85	86.5-100	89.84-100	23/23	-	-	-	-	100%
N132D4	97.12-99.97	91.18-100	92.91-100	39/39	95.1-100	79.08-100	83.17-100	167/172	95.73-100	79.08-100	87.17-100	37/40	96.8%
N17D3	79.69-98.49	80.35-99.1	84.43-99.29	7/7	72.43-100	80.35-100	84.43-100	25/27	76.94-88.72	78.57-97.32	83.01-97.87	8/8	95.2%
N32D6	85.64-99.74	79.85-100	83.95-100	13/13	82.56-100	73.13-100	78.6-100	43/44	81.02-97.69	78.73-98.88	83.06-99.1	10/10	98.%
N73D14	93.04-99.93	87.59-100	90.09-100	29/29	90.76-100	75.18-100	80.19-100	88/90	87.12-91.25	72.42-81.98	77.98-85.61	17/17	98.5%

6.4 Chapter summary

This chapter addressed the problem of detecting inconsistencies between ICL and RTL models of RSNs, resorting to simulation-based verification. Automatically generated test-benches for stimulating the RTL model are based on the patterns used for post-silicon validation of networks. The approach was verified through a series of experimental benchmarks, obtaining extremely high detection coverage with reduced execution time. We were also able to identify test escapes as pathological network configurations.

Summary of Part I

This part of the thesis addressed issues related to Reconfigurable Scan Networks. Relatively new standard IEEE 1687 describes architectural structures and two new languages. They are used to enable efficient embedding of various instruments that support test, debug, monitoring and calibration/configuration in the system. The tools for supporting their integration and operation are being developed as they continue to attract more and more interest from the industrial point of view.

Initially, the first part describes several techniques to minimize time to test, i.e. to check if reconfigurable modules work as they are supposed to as well as to perform diagnosis and identify the faulty module. Faults that have been observed are permanent and have been considered using a high-level fault model. Then it proposes novel approaches for post-silicon validation, equivalence checking between ICL and RTL descriptions, and finally, NBTI-aging effect analysis and mitigation in RSN logic paths.

Part II

Hardware Security: Hardware Trojans

Chapter 7

Background

The ever-growing complexity of modern devices and the fabrication costs led the Integrated Circuit (IC) industry to pursue a new global business model. In that regard, more companies around the world are deeply involved in all phases of the IC supply chain (Fig. 7.1). The outsourcing of part of the process to untrusted third-party entities raises increasing concerns about the hardware security of the products. The situation is becoming both critical and challenging and requires careful regard.

Specific measures need to be taken for detecting, avoiding, and mitigating potential threats based on a component's importance. Furthermore, security needs are driven by the evolving types of attacks, i.e., new adversary models, and type and intended use of the device. No single solution exists able to obtain complete protection. A common stance both in academia and industry is that such a solution should be a set of flexible technologically-driven solutions that are to be applied during the whole life-cycle of the device: development, deployment and operation.

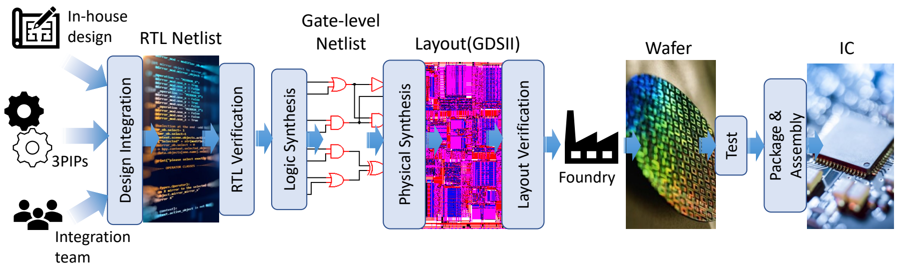


Figure 7.1: IC production flow

Apart from detecting and localizing accidental bugs as a part of design and production flow, it is necessary to identify intentionally placed malicious circuits. Different reports warn about such threats from both malicious and negligent actors and vulnerabilities they can exploit. Particularly, the so-called Hardware Trojans (HTs) continue to gain worldwide attention not only from industry (military) and academia, but also from government bodies [21].

A Hardware Trojan is defined as a malicious and intended alteration of a circuit, that endangers the trustworthiness and the security of the hardware, leading to unexpected behaviour. For instance, it may leak secret information, change the circuit functionality or degrade the performance. A typical HT is composed of a trigger and a payload circuit (Fig. 7.2). The trigger usually monitors specific signals or series of events under some internal or external conditions. When the trigger condition is met, it informs the payload circuit, which executes the malicious function. The trigger is usually hidden under rare conditions, so the HT is dormant for most of the time and the payload, inactive. In that case, the circuit acts as a Trojan-free circuit. If the activation does not depend on the trigger circuit, the Trojan belongs to another category, denoted as *always-on*. Such Trojan gets activated as soon as its host design is powered on. Techniques to deal with the latter exist on different levels of abstraction in IC design: from logic-level search for sequentially-deep states, to unexpected patterns of power consumption.

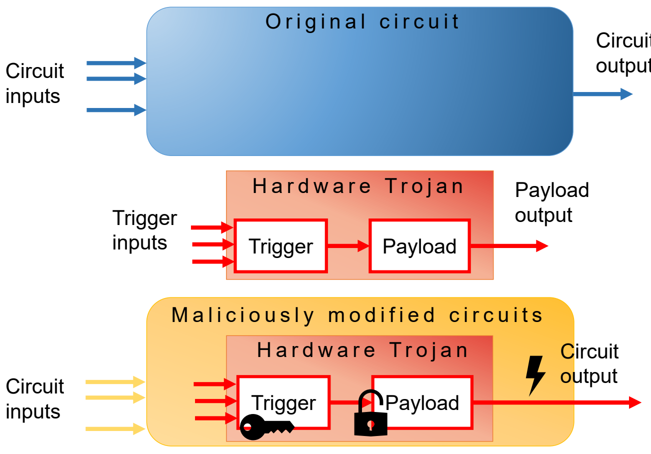


Figure 7.2: Hardware Trojan structure

The Section 7.1 provides an overview of the state-of-the-art regarding ML techniques for RT-Level HT detection.

7.1 Related Works

To better position the work on HTs, the following subsections introduce the most relevant state-of-the-art HT design methodologies and detection techniques.

7.1.1 HT Design

Some works have focused on design possibilities and proposed certain methodologies to create new types of HTs. In [78], authors discuss the design and implementation of RTL HTs to be hard to trigger and able to evade hardware trust verification based on unused circuit identification (UCI). They rely on specific coding style and trigger input selection. Additionally, signal controllability is examined from the attacker's perspective. In [79], authors explored different implementations of HTs with different combinations of triggers, payloads, as well as unique sections of the architecture that each HT attacks. They were all designed with a varying level of sophistication, allowing the attacker to trade-off design time, ability to evade detection, and payload. They concluded that RTL designs can be quite vulnerable to hardware attacks given the vast insertion space and functional testing can often be useless in detecting them. Apart from introducing a metric for quantifying HT activation and effect, [80] introduces a vulnerability analysis flow by determining hard-to-detect areas and provide public trust benchmarks. Some works proposed automatic techniques (malicious CAD tool) for HT insertion. To generate HTs using a highly configurable generation platform, authors in [81] use transition probability to identify the rarely activated internal nodes to target for HT insertion, rather than functional simulation as used in existing platforms. The platform has been tested to generate HT-infected circuits and then evaluated by the ML detection technique [82] — the Controllability and Observability for HT Detection (COTD).

7.1.2 Detection Techniques

Methodologies for detecting triggered-type HTs at RT-Level can be broadly classified as *dynamic* and *static*. The former considers the adoption of verification test patterns and dynamic type of analysis based on, for instance, code coverage metrics. On the other hand, static techniques rely exclusively on static proprieties of the target RTL model, without applying any stimuli. As regards the first class (dynamic), one of the first approaches dates back to 2010, when Hicks et al. [83] presented *BlueChip*, a hybrid design time/runtime system for detecting and neutralizing malicious circuits at RTL. BlueChip is based on the assumption that part of the circuit is dormant during the design verification and could therefore hide a HT. The UCI technique can flag a part of the circuit as suspicious and deactivate it by raising an exception when it becomes active. Its weakness has been demonstrated in [84], showing a class of HTs that evade detection. Though UCI technique may be able to discover many of the HTs shown in literature, it is sensitive to the actual coding style. From another perspective, authors in [85] describe a framework for generating directed test cases to activate HTs. It mixes concrete simulation and symbolic execution. The results are compared with EBMC[1], a state-of-the-art formal model checker, and demonstrate good scalability on large designs. A similar approach is presented in [86], where the authors proposed an automated test generation technique for activating multiple targets in RTL models by the means of concolic testing.

Concerning the static approaches, in [87], the authors exploit a sub-graph isomorphism algorithm for the detection of HTs inside an RTL model. Resorting to a static pre-defined library of known HTs, the algorithm searches for the occurrence of similar structures inside the Control Flow Graph (CFG) of the device under verification. A CFG is a representation in the form of a graph of all the paths in the RTL model that might be traversed during the execution. However, this approach produces a considerable number of false positives. To overcome this limitation, in [88] the authors combine it with a classifier based on a Probabilistic Neural Network, i.e., a feed-forward neural network usually used for classification tasks [89]. Even though the number of false positives is greatly reduced, the main drawback is still the difficulty in finding Trojans that are not included in the pre-defined library. power

[1]http://www.cprover.org/ebmc/

Chapter 8

A Benchmark Suite of RT-level Hardware Trojans for Pipelined Microprocessor Cores

A malicious alteration can be performed during any phase of the production cycle. One category of HTs are those inserted at the manufacturing stage. In this particular scenario, an adversary could access the mask and modify it to add malicious logic. It is supposed that such logic is inserted intelligently, difficult to activate with manufacturing tests given the combination of rare internal signals values that is used to trigger it. However, more interesting are Trojans inserted earlier in the design cycle, at the register transfer level (RTL) or gate-level. Apart from superfluous complex reverse engineering, an attacker inserting a Trojan early in the design process may take advantage of the vast design space. Also, such Trojan may potentially remain hidden even in the following generations of the device.

During the last years, a huge effort has been invested in developing detection methodologies as well as designing benchmark circuits to favour the advancements in research. Indeed, the research community has received a strong drive to adopt open benchmarks for validating their detection techniques. In this light, many HT models have been proposed [90], [91]. However, the growing complexity of modern devices as well as more mature and elaborate detection methodologies call for more complex benchmark circuits. Some of the authors in their studies proposed different HT taxonomies based on the insertion phase, location, abstraction level, activation mechanism, effects, etc. However, it is complicated to create a HT model given the whole spectrum of constantly evolving attacks and adversaries that are gaining access to more and more phases of the IC development.

A common trend is to use benchmarks released from the Trust-Hub platform [80], [92]. Considering HTs at RT-Level, only 8 typologies of benchmarks are currently available in Trust-Hub (Table 8.1), and none of them is applied to a pipelined processor similar to the ones used in the real-life, as the ones in the automotive

Table 8.1: Number of RTL Hardware Trojan benchmarks available on Trust-Hub [80] [92]

Design Class	AES	b19	BasicRSA	MC8051	memctrl	PIC	RS-232	wb_conmax	TOTAL
Number	21	3	4	7	1	4	10	2	52

applications. This is even more concerning, given higher flexibility for implementing different kinds of malicious functions at RTL. The available HTs are injected on a small 8-bit 8051 microprocessor, and a detection technique has already been proposed in [83]. Hence, even the state-of-the-art HT detection techniques are validated on obsolete benchmarks that do not reflect the true complexity of the modern embedded devices. As stated in [91], to further support the development of appropriate detection methods, the design and implementation of practical HTs need to be considered.

To fill this gap, a total of 28 Hardware Trojans Benchmarks targeting a pipelined RISC microprocessor core are released and presented in this chapter[1]. From the structure of 8 HTs placed in different CPU's locations, additional benchmarks were derived by modifying their trigger mechanism. Their design follows the guidelines for creating a hard-to-detect Trojan, presented in [78].

Section 8.1 describes the typology and general structure of new benchmarks. Section 8.2 deepens the HTs design and provides implementation details together with the impact such injection has on power, area and frequency.

8.1 Hardware Trojans

The proposed benchmarks are intellectual property (IP) level Hardware Trojans conceived for a pipelined Central Processing Unit (CPU). Such Trojans are implanted into an individual IP core of the SoC and can affect only the specific IP in which they are embedded [93]. The benchmarks comply with the taxonomy and the classification scheme outlined in [21], [80], [92]. Furthermore, the following attributes are outlined for each benchmark: abstraction level, insertion phase, location, activation mechanism, trigger, payload, effect. For the sake of completeness, the insertion phase of the HTs is the Design phase, while the abstraction level is the Register-Transfer level for all of the introduced benchmarks. Concerning the effects, the benchmarks might prove to be disastrous or introduce minor damage. Three different categories have been identified:

1. **Degrade Performance (DP)**: The availability of the system under attack

[1]The presented HT Benchmarks will be made available for the research community and will be uploaded on Trust-Hub platform.

might not be affected, remaining fully operational. However, the HT might damage the performance of an IC and, in the worst-case, cause it to fail.

2. **Denial Of Service (DoS)**: The HT when activated stops all the activities of the system.
3. **Change the Functionality (CF)**: The HT alters the functionalities of the system, causing it to perform malicious, unauthorized operations. The CF might also lead to a DP or DoS.

Table 8.2: Trojan Benchmarks Description

Name	Location	Trigger	Payload	Cat
OR1K-**T1**00	Decode Unit	Sequence of instructions	Periodically forcing signal values	DP
OR1K-**T2**00	Control Unit	Counters monitoring read accesses to SPRs	Entering the supervisor mode	DoS
OR1K-**T3**00	PIC Unit[2]	Counters for mask and status reg. write access	Disabling external interrupts	CF
OR1K-**T4**00	Control Unit	3 counters for monitoring instructions	Disabling control flag bit	CF
OR1K-**T5**00	Decode Unit	A specific sequence of instructions	Introducing "bubbles" to stall the pipeline	DP
OR1K-**T6**00	Data Cache	Counters monitoring Data Cache Final State Machine (FSM) transitions	Invalidating dcache content	DP
OR1K-**T7**00	Load & Store Unit	Instruction type, order and number	Exception on the data bus	DoS
OR1K-**T8**00	Instr. Cache	Counters monitoring Instr. Cache FSM transitions	Invalidating icache content	DoS

Regarding the trigger part in the introduced Trojan benchmarks, they can be grouped into two main categories. The first category is represented by a sequence of events that, when triggered, enable the payload. Such events can be related to different signals in the model, for instance, an exact sequence of instructions or a set of consecutive values observed on a given bus. There are different possibilities for implementing it; however, two main parts can be identified: a set of conditions that activate or deactivate a targeted flag, and the second one for registering that flag with some auxiliary combinational or sequential logic. Given the complexity of the condition, this type of trigger may be difficult to activate, and therefore may escape to standard verification approaches. The second category of triggers is used to create and check sub-conditions. Once all of them are satisfied, the payload is activated. They can be implemented by monitoring different processor resources, for example, by observing certain values on the bus, the order and/or the number of certain instructions, the read/write access to the registers, or tracking the value of control signals between different stages of the pipeline. Sub-conditions may also check the state of counters in charge of monitoring different activities in the processor. The implemented counters may be part of a separate process observing the aforementioned activities or be hidden, for instance, in an already existing state machine. This type of trigger gives the possibility to create wide-range of complex conditions. A HT would generally be expected to be as much controllable as possible from the attacker's perspective. However, working with a microcontroller, i.e., a System-on-chip (SoC) that integrate additional components such as peripherals, memories, etc. renders such access more difficult. Given that all of the benchmarks are developed for a processor core, and that no mechanisms are relying on the user input, i.e., component output, such as switches, keyboards or keywords in the input data stream to activate a Trojan, all of the HTs in our set are considered internally triggered. Moreover, they are activated either depending on the time-based events or on the instructions that are being executed. Table 8.2 reports all the essential details related to the newly developed benchmarks: their name, location, trigger and payload brief description and their category.

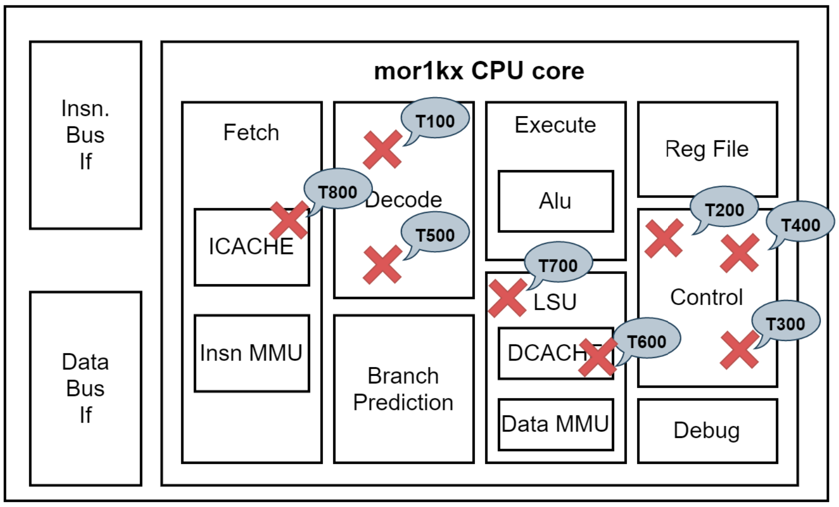
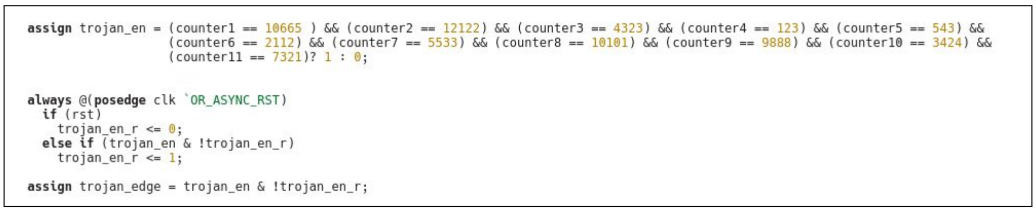


Figure 8.1: Proposed RTL Hardware Trojans in the *Cappuccino* configuration of the mor1kx CPU.

8.2 Trojan Implementation and Analysis

The proposed RTL Hardware Trojans are implemented in the mor1kx CPU, whose architecture and HTs' respective faulty location being depicted in Fig. 8.1. The mor1kx is an open-source core provided by the OpenRISC community; it is a configurable 32/64-bit load and store RISC architecture, written in Verilog Hardware Description Language (HDL).



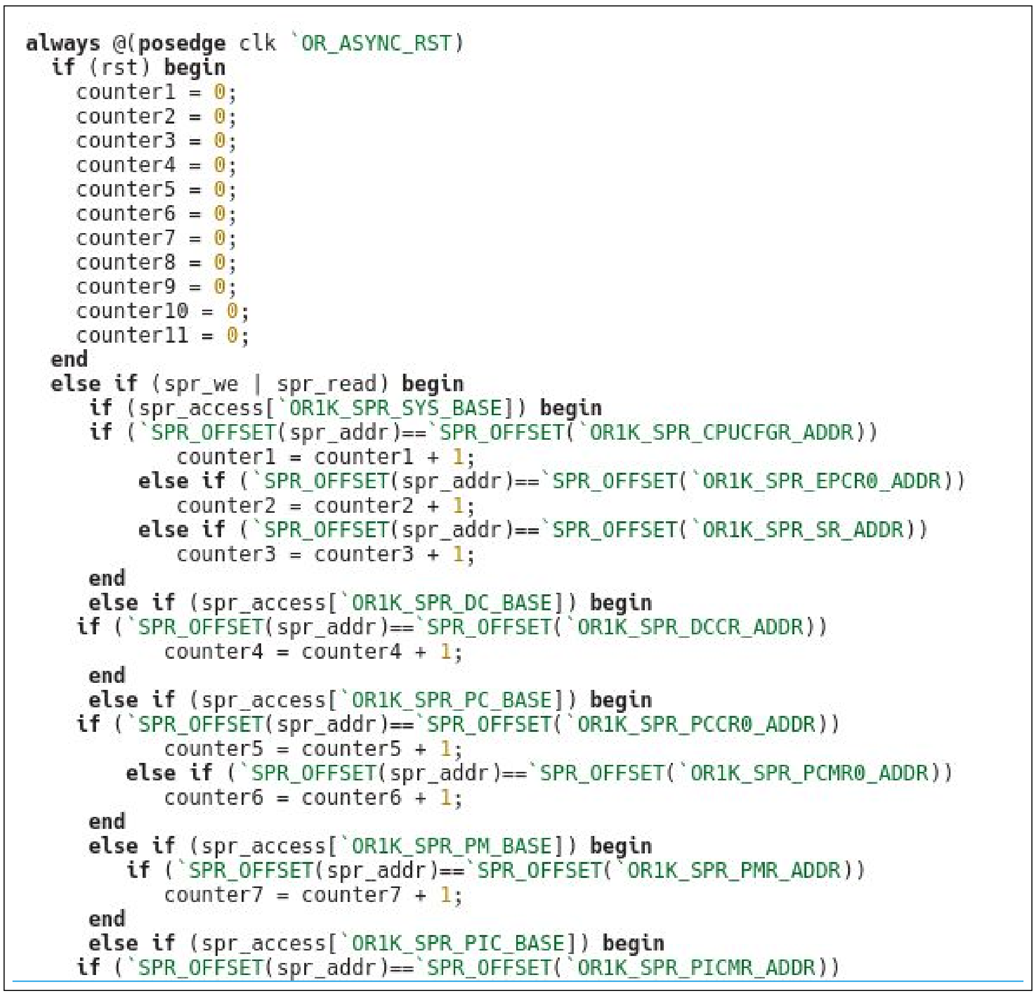
```
assign trojan_en = (counter1 == 10665 ) && (counter2 == 12122) && (counter3 == 4323) && (counter4 == 123) && (counter5 == 543) &&
                   (counter6 == 2112) && (counter7 == 5533) && (counter8 == 10101) && (counter9 == 9888) && (counter10 == 3424) &&
                   (counter11 == 7321)? 1 : 0;

always @(posedge clk `OR_ASYNC_RST)
  if (rst)
    trojan_en_r <= 0;
  else if (trojan_en & !trojan_en_r)
    trojan_en_r <= 1;

assign trojan_edge = trojan_en & !trojan_en_r;
```

Figure 8.2: Trigger T200 condition

Due to the high design flexibility, it is possible to customize the core by choosing the best trade-off between area and performance. The version selected in this work (*Cappuccino*) has a pipeline with 4 stages, supports delay slot and is tightly coupled with the caches. It also integrates a Programmable Interrupt Controller (PIC), a Tick Timer (TT) and Debug units. In this work, HTs are injected into the original HDL design, one at a time, by directly modifying the RTL code. On top of 8 primary HT designs, detailed in Table 8.2, we performed modifications concerning the complexity of trigger conditions and coding style to expand our benchmark library and to obtain additional 20 HT designs.
Trojan T100: This Trojan is located in the processor's decode-execute unit (decode to execute signal stage passing). A new process has been added to monitor the instructions being executed. An if-then-else nested structure controls the opcode value originating from the decode unit. Each time an instruction gets decoded, if the sequence is correct, a counter is incremented; if the sequence is interrupted, the counter is reset. The sequence of instructions is ORI-ADDI-AND-ORI-SUB-XOR-AND-XORI-ADD-OR. Once the counter reaches the value 10, i.e., consecutive instructions correspond to the above sequence, payload gets activated. In this case, pipeline is stalled indefinitely, thus disrupting the service.
Trojan T200: This implementation is located in the control unit of the processor. Eleven counters in the newly added process monitor read and write access of special purpose registers (CPUCFGR, EPCR0, SR, DCCR, PCCR0, PCMR0, PMR, PICMR, PICSR, TTMR, TTCR) (Fig. 8.2). With each access, a corresponding counter is incremented. When all of the counters reach pre-defined values, a trigger is activated (Fig. 8.3). The payload in this case is integrated into existing code by adding a single OR condition to go from user to supervisor mode. Such behaviour is typical when an exception occurs. The effect is interrupts and timer exceptions



```
always @(posedge clk `OR_ASYNC_RST)
  if (rst) begin
    counter1 = 0;
    counter2 = 0;
    counter3 = 0;
    counter4 = 0;
    counter5 = 0;
    counter6 = 0;
    counter7 = 0;
    counter8 = 0;
    counter9 = 0;
    counter10 = 0;
    counter11 = 0;
  end
  else if (spr_we | spr_read) begin
     if (spr_access[`OR1K_SPR_SYS_BASE]) begin
     if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_CPUCFGR_ADDR))
             counter1 = counter1 + 1;
         else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_EPCR0_ADDR))
             counter2 = counter2 + 1;
         else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_SR_ADDR))
             counter3 = counter3 + 1;
     end
     else if (spr_access[`OR1K_SPR_DC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_DCCR_ADDR))
            counter4 = counter4 + 1;
     end
     else if (spr_access[`OR1K_SPR_PC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PCCR0_ADDR))
            counter5 = counter5 + 1;
        else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PCMR0_ADDR))
            counter6 = counter6 + 1;
     end
     else if (spr_access[`OR1K_SPR_PM_BASE]) begin
        if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PMR_ADDR))
            counter7 = counter7 + 1;
     end
     else if (spr_access[`OR1K_SPR_PIC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PICMR_ADDR))
```

Figure 8.3: Trigger T200 counters

being disabled, as well as Data and Instruction MMU. Additionally, a device that is in the supervisor mode enables access to some sensitive registers.

Trojan T300: This HT is located in the programmable interrupt controller. Two counters are inserted to count write accesses to *picmr* (PIC mask) and *picsr* (PIC status) special-purpose supervisor-level registers. Once the trigger part is activated and there are no pending interrupts, payload gets to perform its role by masking all maskable interrupts, which may result in disastrous consequences in safety-critical systems. Reset needs to be performed to unmask such interrupts and disable the HT.

Trojan T400: Malicious trigger-part of this HT consists of three counters counting the number of 3 instructions in the control stage (rfe – return from exception, mfspr – move from special purpose register, mtspr – move to special purpose register). When all three counters count up to a predefined value, the payload is activated. Once activated, the malicious function is designed to prevent the first succeeding setting of the compare-conditional branch flag by adding a simple condition in the assign statement. However, the effect can be severe, given that often a processor when dealing with some instructions uses exactly this flag to calculate the address or/and choose the operand, which may disrupt the desired flow and cause serious problems depending on the application. Once the request for setting the flag arrives Trojan performs its malicious function and gets deactivated. Additionally, a reset signal resets the counters and deactivates the Trojan.

Table 8.3: Synthesis results

Design	Size					**δArea**[%]	Power mW				**δPower**[%]
	Ports	Nets	Cells	Comb./Seq.	Area		Intern.	Switch	Leak.	Total	
Orig.	9,679	931,538	924,619	601,116 323.252	4,777,062.18	–	257.62	4.93	69.42	331.99	–
$T100$	9,679	931,567	924,648	601,145 323,252	4,777,100.66	0.1×10^{-2}	257.62	4.93	69.42	331.99	-1.81×10^{-4}
$T200$	9.679	932,716	925,797	602,038 323,508	4,781,729.10	9.8×10^{-2}	257.82	4.94	69.48	332.24	7.60×10^{-2}
$T300$	9,679	931,899	924,980	601,412 323,317	4,778,437.10	2.9×10^{-2}	257.67	4.93	69.44	332.06	2.03×10^{-2}
$T400$	9,679	932,033	925,114	601,515 323,348	4,779,015.82	4.1×10^{-2}	257.70	4.93	69.45	332.09	3.07×10^{-2}
$T500$	9,679	931,793	924,874	601,333 323,290	4,777,998.70	2.0×10^{-2}	257.65	4.93	69.43	332.03	1.35×10^{-2}
$T600$	9,679	932,056	925,137	601,535 323,351	4,779,066.78	4.2×10^{-2}	257.69	4.93	69.43	332.06	2.11×10^{-2}
$T700$	9,679	931,787	924,867	601,330 323,286	4,777,932.66	1.8×10^{-2}	257.65	4.93	69.43	332.03	1.16×10^{-2}
$T800$	9,697	932,052	925,043	601,441 323,351	4,779,032.98	4.1×10^{-2}	257.70	4.94	69.45	332.09	3.18×10^{-2}

Trojan T500: In the decode to execute unit, a Trojan is implanted to monitor the consecutive instructions. Once the sequence of instructions corresponds to the sequence of 14 pre-defined instructions a trigger is activated. The difference with respect to some of the others HTs introduced in this paper is that this HT introduces two processes for registering the activation signal and producing a pulse. In that manner, the payload gets activated periodically. The payload is added to the condition to form the *decode_bubble_o* signal and insert periodically a bubble into the pipeline. The effect is no change in functionality of the processor. However, due to the stalls it becomes slower, thus, degrading the performance.
Trojan T600: This HT has been inserted into the data cache module. The trigger part consists of 3 counters inserted in the state machine. The Cache FSM has five states: IDLE, WRITE, READ, REFILL, INVALIDATE. The counters have

been inserted to count the transitions between the states: IDLE to INVALIDATE, READ to REFILL, WRITE to READ. Once all of the three counters reach certain values, cache invalidation is forced.

Trojan T700: This HT is located in the load-store unit. Trigger part consists of nested if-else examining the sequence of consecutive multiple load i.e., store operations with 3 different types of access: byte (8), half-word (16) and word (32). Once the complex condition gets satisfied, a pulse signal is generated to activate the payload. The payload in this case is integrated into the process dealing with the data bus exceptions. In this regard, once the payload becomes activated, it will execute its malicious function by simulating a data bus exception and stepping into the exception routine. As a result, the processor proceeds to the next instruction in the pipeline skipping the current one at the moment when the exception occurred. Such event may definitely disrupt the normal operation of the processor.

Trojan T800: This HT is implanted into the instruction cache unit. Its trigger part is incorporated within the FSM with counters following FSM state transitions. Once all the counters get set to predefined values, a payload is activated: the internal hit signal is tied to zero, therefore, every time a request is sent, the instruction cache reports a miss, i.e., not found in cache memory. Consequently, a refill operation is performed, thus significantly slowing down processor's performance.

To demonstrate the feasibility of performing the proposed modifications and inserting malicious code, we synthesized all of our 8 HT designs, including the original one, with a 65nm industrial technology. Successively, we collected reports regarding area, power and frequency. The results given in Table 8.3 clearly show that such insertions are negligible in terms of area and power overhead. The relative area difference is below 9.8×10^{-4}, while the total power relative difference is below 7.6×10^{-4}. Furthermore, we have confirmed that the critical path in the design does not change by introducing the proposed modifications.

Starting from the structure of these original 8 HTs, 20 additional benchmarks have been derived by making changes mainly on the trigger part (complexity of trigger conditions, changing the comparison values, and changing them structurally). For instance, if the trigger looks for a particular instructions sequence, this has been shortened or extended. Additional wire signals for controlling the conditions are introduced, and the position and number of counters is changed together with comparison values. Furthermore, if the trigger sequence was hosted in a single RTL process, it has been split up to use two or more processes, clearly maintaining the same sequence. For example, Trojan T200, originally uses the value of 11 counters to control the trigger condition. A modified version of this Trojan uses 14 counters for its activation. Their values are incremented within two separate processes (10 + 4). The aforementioned changes are especially useful for evading detection by some methodologies that rely on one particular coding style. On the whole, the benchmark set finally contains a total of 28 HT.

Functional testing is quite unlikely to detect malicious circuitry based on instruction or access sequences as the input space is too large. The number of instructions in 32-bit version of the processor is 96 (including custom ones). Therefore, the probability of activating Trojan T100 is 10×10^{-20} order of magnitude. Moreover, functional verification/testing is statistically useless trying to detect HTs observing multiple counter values. It is not only because of the large number of conditions but also given the large comparison values and limited time required to run the simulations. All of the listed HTs can get excited and are not completely dormant/silent in terms of activity. Nevertheless, the probability of activating the payload is extremely low without the knowledge of HT's structure inserted by the attacker. UCI detection technique has certain limitations. UCI can be avoided by inserting malicious circuits that affect unchecked outputs. Unchecked outputs could arise from incomplete test cases or from unspecified output states. Additionally, an attacker might exploit implementation-specific behavior and hide a HT in a module such as cache. Such affected outputs might be difficult for a testing program to check deterministically, thus causing malicious circuits to affect outputs and avoid UCI analysis.

8.3 Chapter summary

In this chapter, a set of 8 new principle HTs is introduced as well as their 20 modifications for a pipelined processor core. The proposed HTs have been injected into very different parts of the processor design. They differ in the trigger and payload. The synthesis reports show the negligible impact that the introduced modifications have on area, power and frequency.

In the author's modest opinion the set of benchmarks could be extremely useful for validating dynamic HT detection methodologies since the core is open-source and in the near future the HTs will be also made publicly available.

Chapter 9

Machine Learning for Hardware Security: Classifier-based Identification of Trojans in Pipelined Microprocessors

Researchers have faced the hardware-based security problems from several angles. The state-of-the-art detection techniques can be classified according to different factors: the Trojan typology, the insertion time, the abstraction level, the location, the activation mechanism, the effects, the physical characteristics, the need for a golden model, etc. As a result, comparing all the work that has been done is a challenging task. An interesting overview is provided in [21], where the authors summarize what has been covered and suggest a roadmap for future research in this field. Interestingly, many methodologies utilizing Machine Learning (ML) for HT defence have emerged [93]. Among the existing ML-based techniques, Artificial Neural Networks (ANN) are commonly used for predictive analysis. Another commonly used ML approach for the problem of binary classification is Support Vector Machine (SVM). The success and popularity of ML methodologies in various research domains has motivated both industrial and academic communities to explore the potential of applying them to the hardware security field. In particular, the main progress in ML-based techniques has been achieved in three widely used detection methodologies: reverse engineering [94], [95], circuit feature analysis [96], [97], and side-channel analysis [98], [99].

This chapter describes a way to exploit powerful and robust ML techniques for Hardware Trojan detection. The proposed methodology is applied at the pre-silicon phase of the supply chain, and is based on a deep-learning analysis of the dynamic and static properties extracted from the design RT-Level model. The former properties are gathered by exciting the model by executing software code; the latter uniquely depends on the structure of the model and the code/data dependency. In this article, these two properties are jointly used to feed the ML model which then performs classification, i.e., calculates the probability of input sample belonging to the malicious insertion, as it will be deepened in Section 9.2. Unlike common approaches, this one combines both static and dynamic properties for building a comprehensive detection methodology at the RT-Level.

Understanding the System-on-Chip (SoC) supply chain Fig. 7.1 is the first necessary step for delineating the possible attacks scenarios. In [21], the authors provide an interesting overview of the SoC development flow and all the entities that come into play. They identify three main phases: the *Intellectual Property (IP) Development*, the *SoC Integration* and the *Foundry*. The first one involves all the IPs providers. An SoC is typically comprised of more than one IP. To reduce research and development costs, some of them are built in-house, others are bought from third-party IP vendors. Once that all the IPs are available, the second phase consists in joining them in a single SoC, i.e., the *SoC Integration*. Both SoC designers and IP providers for facilitating the design process rely on EDA tools. At this point, all the side structures are integrated into the SoC, for example, Design-For-Testability modules, Debug Units, and Built-In Self-Test blocks are typically entrusted to third-party specialized vendors. Once the SoC post-layout phase is done, it is sent to the foundry for IC fabrication. The fabrication process is usually the most costly stage of the flow, thus, their fabrication is usually granted to external foundries. A malicious actor present in any stage can insert the HT at various levels of abstraction. The key issue lies exactly in understanding which of these entities are trusted and which are not. Once it has been established, the threat model can be drawn. In [92] and [21], the authors provide a comprehensive list of adversarial models showing exactly when, where and how a Trojan can be placed into an IC. The detection technique we propose here is ML-based, to be applied during the pre-silicon phase at the RT-level. Note that exploiting ML-based technique is likely to imply the creation of models from a set of historical data and then utilizing these trained models for prediction [93]. Two different learning tasks can be used: supervised learning and unsupervised learning. The former exploits labelled data to perform model training, the latter focuses on the relations between data when labels are unavailable.

9.1 Design Verification and ML concepts

The following section introduces four important concepts that underlie the proposed work. First, the fundamental characteristics of digital design verification are introduced (Section 9.1.1); this background knowledge is useful for describing the dynamic analysis used in our detection methodology. However, it may be superfluous for the readers that possess some basic knowledge on this topic. Next, an overview of Artificial Neural Networks is provided in Section 9.1.2 and finally, the

Support Vector Machine technique is presented in Section 9.1.3.

9.1.1 Digital Design Verification

Digital systems are created by following a series of steps that comprise several intermediate design phases. Clearly, the lower the abstraction level, the higher the complexity of the resulting model. Identifying and removing logic errors in a design is not a trivial task; in fact, nowadays, the development resources devoted to these tasks amount to about 50%-60% of the total cost in the design process [100]. A series of verification processes are required intending to guarantee that the design model meets the expected specifications [101]. Very different methodologies have been developed to generate verification stimuli. The main possibilities range from manual verification techniques to formal verification techniques, including random and semi-random approaches. In particular, simulation-based methodologies try to completely exercise the current model of the device to uncover design errors. Briefly, a simulation-based verification process is composed of three basic elements: input data (also called set of stimuli), the model of the device under evaluation (also called design or device under verification or DUT); and the response checker, which generates the pass/fail information regarding the current process by performing a comparison of the obtained results against the expected ones. To qualify a stimuli set, one of the most used methodologies is based on collecting a series of measurements obtained by computing the code coverage metrics during the simulation of the device while running the stimuli set. These metrics identify which code structures belonging to the circuit description are exercised by the set of stimuli, and whether the control flow graph corresponding to the code description has been thoroughly traversed. The structures exploited by code coverage metrics range from a single line of code to if-then-else constructs. Today CAD tools can measure among other the statement coverage, branch coverage, condition coverage, expression coverage, toggle coverage, and metrics based on Finite State Machine models [102].

9.1.2 Artificial Neural Networks

The origin of Neural Networks dates back to 1950s, when the basic building block of modern neural networks, the perceptron, was first proposed [103]. Over the years, key theoretical discoveries and technological advances allowed this concept to evolve into a brand new field. The original perceptron contains a single input layer and an output node, as shown in Figure 9.1, and may implement a linear binary classifier. The input layer does not perform any computation and thus it is not included in the count of the number of layers in a neural network. Therefore, in modern terminology, the perceptron would be considered a single-layer network. It is worth underlying that modern neural networks are certainly built with more than one computational layer.

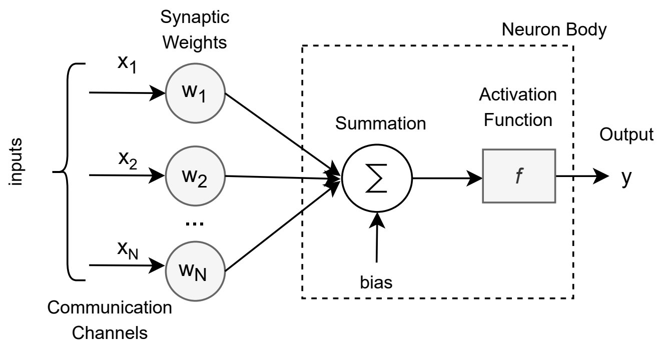


Figure 9.1: The basic architecture of the perceptron.

More in detail, the input layer of a perceptron contains N nodes that transmit the N features X = [x_1 ... x_N] with edges of weight W = [w_1...w_N] to an output node y. The prediction of the perceptron is computed as follows:

$$y = f(\sum_{j=1}^{N} w_\mathrm{j} x_\mathrm{j} + b) \tag{9.1}$$

Where x_j are the inputs, w_j the weights, b is the bias. The activation function (f) defines how the weighted sum of the input is transferred to the output node. The choice of f is considered a critical part of neural network design since it has a large impact on the capability and performance of the neural network. The interpretation of the perceptron as a computational unit is useful, and it allows us to combine multiple units (i.e., multi-layer perceptron) to develop far more efficient models [104]. Broadly speaking, Artificial Neural Networks are computing models composed of computing nodes, connected through communication links (Fig. 9.2). Nodes are arranged in layers, at least one input layer, one intermediate (or hidden), and one output layer.

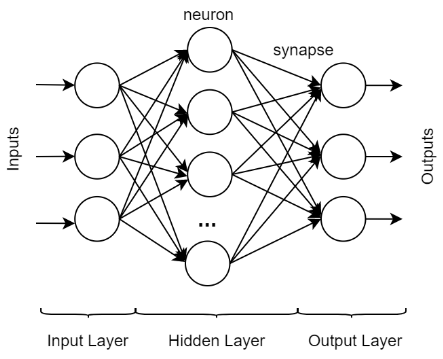


Figure 9.2: Artificial Neural Network: a basic representation.

Nowadays, the conventional machine-learning applications used for example to identify objects in images or transcribe speech into text make use of techniques stemming from NN and labelled as “deep learning” [105]. Thanks to the multiple levels of representations, quite complex functions can be learned; nevertheless, in the building blocks of such structures, it is still recognizable the old idea of perceptron.

Over the years, many different neural network architectures have been created depending on the layers and their organization, the activation functions and many other exploited features. Among the most common and widespread types are convolutional neural networks (CNNs) and residual neural networks (ResNet) for image classification and object detection tasks; recurrent neural networks (RNNs) for tasks that involve sequential inputs such as speech and language. Recent studies have demonstrated that state-of-the-art neural networks can surpass human-level performance: for example, in [106] the authors achieved 4.94% top-5 test error on the ImageNet classification dataset, and the human-level performance was 5.1%, according to Russakovsky et al. [107]. The potential of this kind of deep and complex neural networks (e.g., ResNet [108]) reflects the fact that biological neural networks gain much of their power from depth.

9.1.3 Support Vector Machine

An SVM, close to its current form, was described in [109] as a training algorithm that maximizes the margin between the training patterns and the decision boundary. It has been developed from the Statistical Learning Theory in the 1960’s [110]. The goal of the SVM algorithm is to define an optimal separating hyperplane for a two-class dataset. SVM tries to maximize the width of the margin between the so-called support vectors, that is, training samples that lie closest to the separating hyperplane (Figure 9.3).

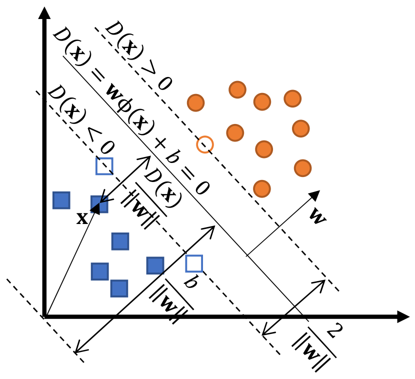

$D(\mathbf{x}) > 0$
$D(\mathbf{x}) = \mathbf{w}\phi(\mathbf{x}) + b = 0$
$D(\mathbf{x}) < 0$
$\frac{D(\mathbf{x})}{\|\mathbf{w}\|}$
$\mathbf{x}$
$\mathbf{w}$
$\frac{b}{\|\mathbf{w}\|}$
$\frac{2}{\|\mathbf{w}\|}$

Figure 9.3: Defining a border between classes using an SVM (support vectors are marked with □ and ∘)

Training input for the system can be represented as a set of r elements: $\{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), (\mathbf{x_3}, y_3)$ where $\mathbf{x_i}, i = 1{,}2, \ldots, r$ represents a n-dimensional input sample vector with the corresponding response value $y_i, i = 1{,}2, \ldots, r$ (9.2).

$$y_i = \begin{cases} 1, & \text{if } \mathbf{x} \in A \\ -1, & \text{if } \mathbf{x} \in B \end{cases} \tag{9.2}$$

At the end, after the training process is done, the class of the new input data vector $\mathbf{x}$, will depend on the decision function value, $D(\mathbf{x})$, in such a way that it represents a position below or under the hyperplane that separates two classes. This function can be expressed as a linear combination of parameters (9.3),

$$D(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j + b = \mathbf{wx} + b \tag{9.3}$$

where w_j are coefficients, $\mathbf{x}$ is input vector and b is a bias coefficient. Conditions for discrimination between input sample $\mathbf{x_i}$ being on one (9.4) or the other side (9.5) of the hyperplane, can be unified into a single condition (9.6).

$$\mathbf{wx_i} + b \geq 1, y_i = 1 \tag{9.4}$$

$$\mathbf{wx_i} + b \leq -1, y_i = -1 \tag{9.5}$$

$$y_i (\mathbf{wx_i} + b) \geq 1 \tag{9.6}$$

$$\mathbf{wx}_+ + b = 1 \tag{9.7}$$

$$\mathbf{wx}_- + b = -1 \tag{9.8}$$

$$\mathbf{w}(\mathbf{x}_+ - \mathbf{x}_-) = 2 \tag{9.9}$$

$$M = \frac{\mathbf{w}}{\|\mathbf{w}\|}(\mathbf{x}_+ - \mathbf{x}_-) = \frac{2}{\|\mathbf{w}\|} \tag{9.10}$$

Margin M is defined using a difference (9.9) of two samples $\mathbf{x}_+$ (9.7) and $\mathbf{x}_-$ (9.8) lying on two boundaries. The objective of this algorithm is to find the coefficient vector $\mathbf{w}$ to maximize the margin M (9.10). To summarize, the goal is minimizing $\frac{\|\mathbf{w}\|^2}{2}$ with the condition of correctly classifying all the points $y_i (\mathbf{wx_i} + b) \geq 1$.

Solution to the problem of minimizing $min_x f(x)$, with respect to $g_i(x) \leq 0$ for $i = 1, \ldots, n$ is equivalent to finding the solution to Lagrangian's zero gradient (9.11).

$$\begin{cases} \frac{\partial}{\partial x}\left(f(\mathbf{x}) + \sum_{i=0} \alpha_i g_i(\mathbf{x})\right) = 0, & \exists \alpha_i > 0, i = 1, \ldots, n \\ g_i(\mathbf{x}) \geq 0 \end{cases} \tag{9.11}$$

Comparing the method with the existing problem $f(\mathbf{x}) = \frac{\|\mathbf{w}\|}{2}$, $g_i(\mathbf{x}) = 1 - y_i(\mathbf{w}\mathbf{x_i} + b)$. After setting the Lagrangian's gradient to zero, we obtain essential relation between coefficients and the dual problem to be solved. That is $\max(W(\alpha))$ (9.12), which, in the end, is a quadratic optimization problem. Its solution are α coefficients.

$$W(\alpha) = \sum_{i=1}^{r} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{r} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \mathbf{x_j}, \; \sum_{i=1}^{r} \alpha_i y_i = 0, \tag{9.12}$$

After obtaining α_i coefficients, it is trivial to obtain $\mathbf{w}$ as well (9.13). Coefficient vector $\mathbf{w}$ is a linear combination of small number of input sample vectors, and therefore, many of the α_i coefficients are equal to zero. Input vectors $\mathbf{x_i}$ with nonzero α_i coefficients are called *Support Vectors*. From equations (9.3) and (9.13) final form for calculating the score is derived (9.14).

$$\mathbf{w} = \sum_{j=1}^{s} \alpha_{t_j} y_{t_j} \mathbf{x_{t_j}} \tag{9.13}$$

$$D(\mathbf{x}) = \sum_{j=1}^{s} \alpha_{t_j} y_{t_j} \mathbf{x} \mathbf{x_{t_j}} + b \tag{9.14}$$

Having a non-linearly separable classification problem, ϵ_i deviation can be introduced (9.15).

$$\begin{cases} \mathbf{w}\mathbf{x_i} + b \geq 1 - \epsilon_i, & y_i = 1 \\ \mathbf{w}\mathbf{x_i} + b \leq 1 + \epsilon_i, & y_i = -1 \\ \epsilon_i \geq 0, & \forall i \end{cases} \tag{9.15}$$

In many applications, constructing a hyperplane is not possible and will not result in successful classification of the input data. By applying the technique called "kernel trick", input space gets mapped into a higher dimensional, linearly separable feature space Fig. 9.4. Complex kernels that are most commonly used are polynomial, gaussian and sigmoid. Linear kernel is the simplest one, corresponding to the dot-product between two vectors and is used for linearly separable data to construct a hyperplane. A linear operation in the feature space is equivalent to a nonlinear operation in the input space.

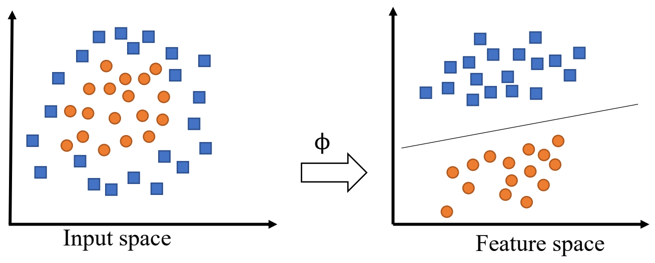


Figure 9.4: Using non-linear kernel functions to map input space

$$K(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$$
$$K_p(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^p$$
$$K_{rbf}(\mathbf{x}, \mathbf{y}) = \exp \frac{-\|\mathbf{x}^2 - \mathbf{y}^2\|}{2\sigma^2}$$
$$K_s(\mathbf{x}, \mathbf{y}) = \tanh(1 + \mathbf{x} \cdot \mathbf{y})^p$$

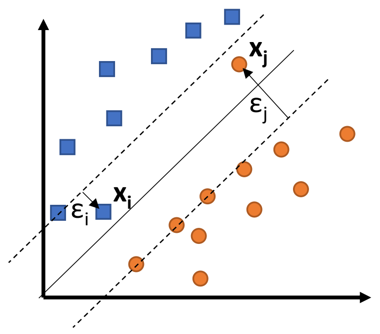


Figure 9.5: To prevent overfitting, avoiding narrow margin is recommended. This can sometimes be achieved by introducing a margin and allowing a certain degree of misclassification

Having a non-linearly separable classification problem, a deviation can be introduced, using a parameter to adjust the desired error/margin. Such parameter is usually referred to as probability threshold. Although we allow for a certain misclassification, the boundary margin is still retained (Fig. 9.5).

9.2 Proposed Detection Flow

The proposed methodology relies on a supervised learning scheme. It is necessary to underline that, apart from [88], the major part of ML-based techniques are applied at the gate-level. However, more and more examples of HTs inserted at RTL are available, due to the flexibility for implementing various malicious functions. Hence, there is a pressing need for more RTL HT detection techniques. To fill the above-mentioned gaps, this paper presents a ML-based methodology for detecting triggered-type Hardware Trojans. It combines a *dynamic* approach with a *static* analysis of the RTL model. Indeed, if a static approach analyzes the structure of the model looking for similarity with the structure of a Trojan, a dynamic method considers the true activity of the circuit. For this reason, the proposed work picks up the best of the two methods in order to cover a greater set of HTs and thus, generalize the detection approach.

The proposed flow is shown in Fig. 9.6. The input of the framework is the design that is about to be processed; it is the behavioural RTL model description. The output is a report indicating suspicious parts in the design, i.e., the code fragments that should be checked more thoroughly for malicious HTs insertion. The RTL design is processed in order to extract both dynamic and static information. While the dynamic is derived from observing the model behaviour under different stimuli, the static is obtained without any code execution and is related to the structure and control/data dependency in the code. The data extracted from the RTL model are embedded in CFGs. Static/Dynamic data are used as *attributes* to create input samples out of node sets for the classification task. At the end, ML-based binary classification is used for distinguishing between input samples originating from the CFGs. The proposed approach is based on the following steps:

- Control Flow Graphs Extraction:
    1. Static Attributes
    2. Assign DataFlow Map
    3. Dynamic Attributes
- Data Formatting
- Classification

In the following subsections, each of the steps is described in a more detailed manner.

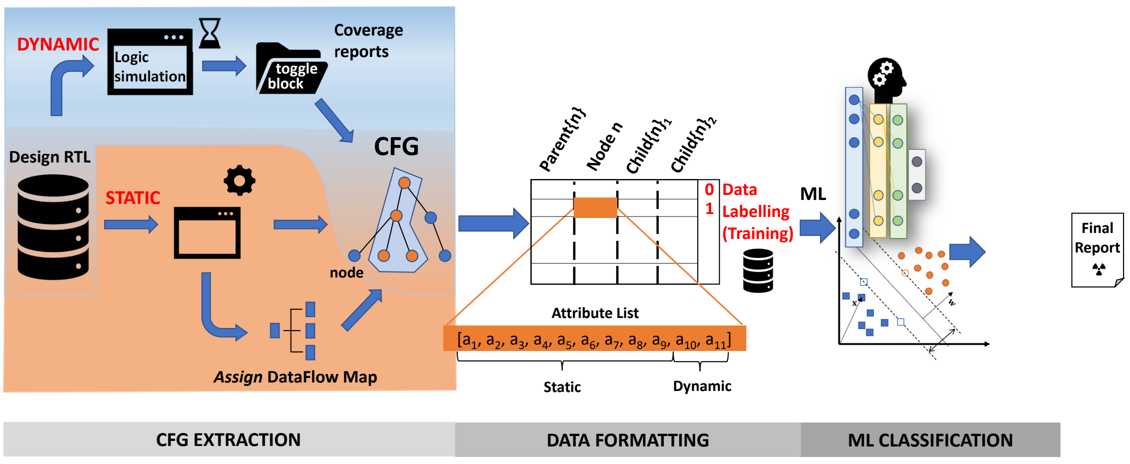


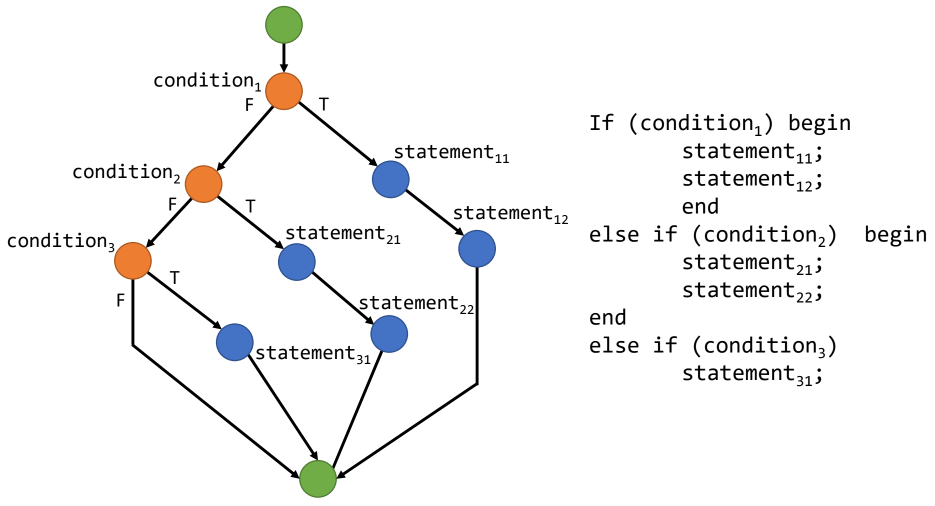
Figure 9.6: Detailed framework flow including three main steps: CFG extraction, Data formatting and ML classification

9.2.1 Control Flow Graphs Extraction

The RTL model of the design is described as a set of concurrent "processes". Two main hardware description languages are VHDL [111] and Verilog [112]. At the initial stage, the RTL design is represented in the form of a CFG, which incorporates key properties of the design: the static, dynamic and dataflow map. These are essential for the training of the NN which is responsible for identifying malicious insertion in the code.
A CFG is a directed graph $G = (V, E, in, out)$, where V is a set of vertices (nodes) and E set of edges. For each process P in the RTL design D, a CFG G can be extracted. A node $v \in V$ of the graph G can be:

- a single non-blocking statement – allow scheduling assignments without blocking the procedural flow;
- a conditional statement/loop (IF-ELSE, CASE, FOR, WHILE).

E is a finite subset of $V \times V$; e is an edge between the nodes $v1$, $v2$ if and only if $v2$ can be executed after $v1$ in the process P. in and out are the first and the last node in a CFG, respectively, used to mark entering the process and leaving the process. An example of the structure and its corresponding CFG are shown in Fig. 9.7. Then, each node in the CFG holds an attribute list, which will be created as described in the following.



```
If (condition1) begin
        statement11;
        statement12;
        end
else if (condition2)  begin
        statement21;
        statement22;
end
else if (condition3)
        statement31;
```

Figure 9.7: CFG with the corresponding code structure

Static Attributes

The static attributes have been extracted from the RTL desing by parsing the source code files. Given the complexity of the modern designs, such a task requires an automated tool. Usually, such tools provide as an output an abstract syntax tree (AST). AST is a convenient hierarchical tree-like representation of the abstract syntactic structure of source code. Then, syntax trees generated by the parser are traversed to perform the extraction of the CFGs in accordance with the definition that was introduced previously. It is worth noting that each of the source files may contain more than one process, which are all elaborated sequentially. The algorithm extracts the list of input signals, registers, wires, output signals, and parameters. A CFG node is identified by its unique name and a unique line number that get assigned inside the processes while creating nodes and attaching them to the corresponding graph. Since one node can represent either a conditional statement, i.e., a loop, or a non-blocking statement, it is possible to extract static properties from such constructs. These include the number of input signals, the number of output signals, the number of logic operators, relational and equality operators, arithmetic operators and numbers (constants). Additionally, each node has its depth in the CFG (level - the number of edges in the path from the root to the node).

Algorithm 15 Generating Assign DataFlow Map

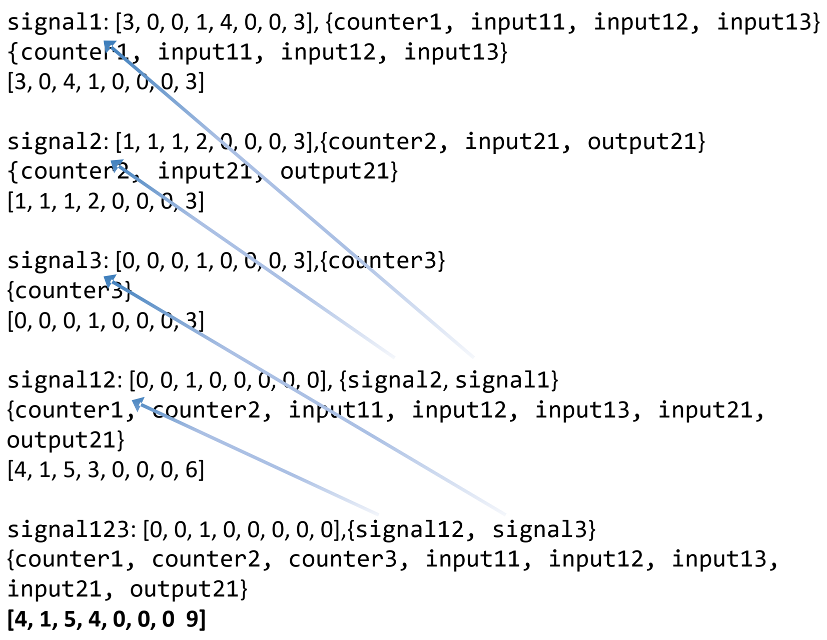
```
function GENERATEASSIGNDATAFLOWMAP($gen$, $len$, $i$, $d$)
    $assignMap \leftarrow [\ ]$
    while $statement$is $assign$ do
        $L, R \leftarrow statement$
        if $L$ in $assignMap_{keys}$ then
            $assignMap[L][0] \leftarrow assignMap[L][0] + R_{attributes}$
            $assignMap[L][1] \leftarrow assignMap[L][1] \cup R_{signals}$
        else
            $assignMap[L] \leftarrow [\ [attributes(R), set(signals(R))]\ ]$
```

Assign DataFlow Map

To deal with the combinational logic (e.g., the `assign` statements in Verilog), the proposed flow introduces an auxiliary structure. Creating an Assign DataFlow Map allows the information outside of the (sequential) processes to be captured and incorporated later into the CFGs Algorithm 15. The left part of the assign statement is used as a key to identify an item in such a structure, while the corresponding value is in a form of a list. Its first element is an array of properties that coincide with the ones for the statements inside the process (*static attributes*). The second one is a list of used signals, either inputs (*input*), registers of integers (*reg*, *integer*) or other wires (*wire*). The map is searched recursively for all of its key elements, summing up the attributes for a corresponding signal list. It stops when there are no more wire signals, i.e. if the remaining ones are a register, integer,

```
assign signal1 = input11 & !(input12 | input13) &
(counter1 == 121346) ? 1 : 0;
assign signal2 = (input21>output21) &
(counter2 == 314431) ? 1 : 0;
assign signal3 = (counter3 == 122214) ? 1 : 0;
assign signal12 = signal1 & signal2;
assign signal123 = signal12 & signal3;
```

Figure 9.8: Assign statements



```
signal1: [3, 0, 0, 1, 4, 0, 0, 3], {counter1, input11, input12, input13}
{counter1, input11, input12, input13}
[3, 0, 4, 1, 0, 0, 0, 3]

signal2: [1, 1, 1, 2, 0, 0, 0, 3],{counter2, input21, output21}
{counter2, input21, output21}
[1, 1, 1, 2, 0, 0, 0, 3]

signal3: [0, 0, 0, 1, 0, 0, 0, 3],{counter3}
{counter3}
[0, 0, 0, 1, 0, 0, 0, 3]

signal12: [0, 0, 1, 0, 0, 0, 0, 0], {signal2, signal1}
{counter1, counter2, input11, input12, input13, input21,
output21}
[4, 1, 5, 3, 0, 0, 0, 6]

signal123: [0, 0, 1, 0, 0, 0, 0, 0],{signal12, signal3}
{counter1, counter2, counter3, input11, input12, input13,
input21, output21}
[4, 1, 5, 4, 0, 0, 0 9]
```

Figure 9.9: Assign DataFlow Map

input or output. For example, in Fig. 9.9, for `signal123`, it adds the attributes of `signal12` and `signal3`, then it does the same for `signal12`, taking the attributes of `signal1` and `signal2`. On the other hand, `signal1`, `signal2`, and `signal3` do not contain in their signal set any keys from the map entries. While creating the CFGs and extracting their nodes' static attributes, the influence that a signal present in Assign DataFlow Map has on a statement inside the process is taken into account by adding its attributes from the corresponding value in the map entry.

Dynamic Attributes

Logic simulations of the design under assessment are performed to collect code coverage reports, based on standard metrics such as statement and toggle coverage. The idea is to gather information from a set of programs that thoroughly exercise the design under analysis. It is essential to outline that such a set of programs may have been written either as a part of pre-silicon or post-silicon verification, validation, or even manufacturing tests etc., targeting different parts and different features of the system. For every instance in the design, uncovered sequential statements belonging to a process are listed with their line number, source code, and type (*if* and *case* conditional structures, *for* and *while* loops together with non-blocking assign statements). The second type of reports focuses on the toggle activity of signals which are being used outside of sequential processes as inputs/outputs, to model combinational logic in assign statements. For each and every program in the library, a statement-coverage report is generated, while only one merged report for all runs regarding the signal toggling.

Hence, two additional fields have been created in the attribute list for such purpose: one for execution probability and one for signal toggling activity.

Regarding the former, a category is decided for each node (statement) based on the number of executions, i.e., how many times it was covered. This technique is an important tool for preparing numerical data for ML and is referred to as unsupervised discretization [113]. It consists of transforming data from continuous to discrete, using e.g., equally wide intervals. A typical use case is having many unique values to model effectively. In Eq. (9.16) that shows the range for deciding a category, n_{exec} is a number of times a statement has been covered out of M runs. N is the number of intermediate categories, set to 5. Consequently, apart from two extreme categories *never* (N) and *always* (A), there are other five: *almost never* (XS), *rarely* (S), *sometimes* (M), *often* (L), and *almost always* (XL).

$$i\frac{M}{N} \le cat(n_{exec}) < (i+1)\frac{M}{N}, i \in \{0,1,2,\cdots,N\} \tag{9.16}$$

As for the latter, toggle reports are merged for all runs into one report, showing if a wire signal has toggled in at least one run, fully or partially (rise and fall). The algorithm embeds such information into a node belonging to a process statement in the following manner: wire signals are listed in such statement, if any, otherwise score 0 is set; based on their total number t and the number of those that toggled d a ratio $R = \frac{d}{t}$ is calculated; R falls into one of the ranges, 0, $(0, \frac{1}{4}]$, $(\frac{1}{4}, \frac{2}{4}]$, $(\frac{2}{4}, \frac{3}{4}]$, $(\frac{3}{4}, 1)$, 1, and gets assigned a value from 6 to 1.

9.2.2 Input Data Formatting

To capture the dependency, structural and functional, between the nodes in a CFG, and bring in context and neighbourhood information into the predictions, the node's closest neighbours may be selected to form a set, i.e., to obtain an input sample. Obviously, such sets may vary in size, given the bound that is chosen for grouping the nodes. It is desirable not to be too generic neither too specific since this action will have an impact on the learning capabilities. For this reason, we considered a set of 4 nodes. Therefore, each node that has at least one parent and at least one child is processed. For nodes with more than one parent P and more than two children C, all the possible combinations are extracted $P \cdot \binom{C}{2}$. A child having no siblings is included in the selection two times. For all the CFGs, the algorithm implementing a set of above-mentioned rules extracts a set S of node selections $t_i = (p_i, n_i, c_{1i}, c_{2i})$. Subsequently, by expanding its nodes with their incorporated attributes gets transformed into $t_a i = (a(p_i)[\ ], a(n_i)[\ ], a(c_{1i})[\ ], a(c_{2i})[\ ])$. For the training, such input data have to be labelled relying on the set of Trojan Benchmarks introduced in [114]. If a central node n_i for which we select its environment belongs to the malicious insertion then, the set of 4 nodes is marked as positive. Otherwise, it is marked as negative, i.e., non-suspicious.

9.2.3 Classification

Once the data have been extracted, the problem may be tackled as a pure Machine Learning classification problem. The learning phase, i.e., the training process, relies on the features obtained from the data formatting. Here, we apply different paradigms to perform the classification and confront their performance in the following sections. The first one is using the SVM algorithm, while the second is based on a fully connected feed-forward neural network.

Classification with Support Vector Machine

SVM algorithm is used with different kernels to choose the one that fits the best for the problem in question. Often the differences in the scales across input variables may affect the training process and therefore the final result. A model might become unstable meaning that it would suffer from poor performance in both learning and validation/test phases, as a result of high sensitivity to input data and higher generalization error. Therefore, using pre-processing techniques such as scaling or normalizing input data is preferred when working with many ML algorithms. Normalization is a scaling of the data from the original range so that all values are within the new range between 0 and 1. It can be performed on an individual data sample (row-wise) or across data features (column-wise). Standardization, on the other hand, includes transforming data in such a way to change its distribution of values: the mean of observed values becomes 0 and the standard deviation 1. For this particular purpose we perform scaling across the features: $\overline{X} = \dfrac{X - \mu}{\sigma}$.

Classification with an Artificial Neural Network

Given the number of attributes, the number of inputs for a fully connected feed-forward neural network is set to 60, after expanding some of the features with one-hot-encoding. Following the common experience of machine learning experts, having too many layers when dealing with a limited number of training data (an order of magnitude of 1000 samples) may result in underfitting. Furthermore, the number of NN inputs is a limiting factor when defining the number of nodes in layers. Given the previous consideration as well as empirical analysis, the following topology has been adopted: (64, *tanh*), (32, *tanh*), (32, *relu*), (2, *sigmoid*). For the sake of clarity, the first number indicates the number of neurons that constitutes the fully-connected layer while the second parameter specifies the activation function, e.g., hyperbolic tangent, rectified linear unit , sigmoid.

For a fixed topology, tuning training parameters may significantly enhance the NN learning capabilities. Hence, K-fold cross-validation method is employed to find the best optimizer and select optimal parameters such as batch and number of epochs. One of the challenges faced in ML is memorizing the input samples, especially when having a small training dataset. However, the NNs have shown to be more resilient to such problem. In any case, to reduce the generalization error, i.e., to prevent overfitting, a Gaussian noise is added to the input. In this way, the training process is made more robust.

9.3 Experimental Evaluation

9.3.1 Experimental Setup

The selected platform is AutoSoC [115], an open-source SoC benchmark suite, conceived to serve the needs for standardization and benchmarking in the automotive area.

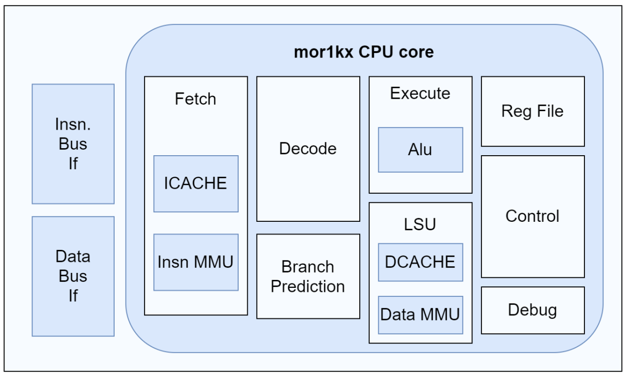


Figure 9.10: mor1kx CPU core in cappuccino configuration

For each one of the 28 benchmarks described previously, the following experimental procedure was used:

1. Parsing of the design model using a set of Python tools and an in-house developed tool to generate CFGs;
2. Performing the logic simulation and report generation using state-of-the-art commercial tools; then, adding the information originating from the coverage and toggle reports to the CFGs;
3. Node extraction: a selection of nodes with their neighbourhood is made (parents and children) to create textual files whose rows contain the attributes for each of the 4 nodes. For the training process, such data have to be labelled manually; repeating items, if any, are eliminated.

The whole setup has been developed to perform logic simulation and generate reports in Linux environment on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. The process itself is managed by a set of bash scripts taking care of design elaboration, design simulation, calculating the coverage and merging the reports. Given the fact that for the training process a certain number of simulations has to be performed on the number of designs with the different type and implementation of HTs, the time required for obtaining the reports can become significant. To speed up the execution time, a multi-process environment has been developed. For this purpose, the complete Test Program Library of *mor1kx* CPU has been simulated on all the 28 RTL *trojan models*. The Test Program Library includes 46 programs for a total of 64 KB. Launching a set of 46 program simulations on one design in this configuration requires 22 minutes on average. By merging the contribution of every single program, the entire Test Program Library achieves 85% of statement and toggle coverage on the golden design model (Fig. 9.11). It is worth underlining that the Test Program Library is not able to activate the Hardware Trojans, being coherent with the assumption that HTs hide under rare trigger conditions.

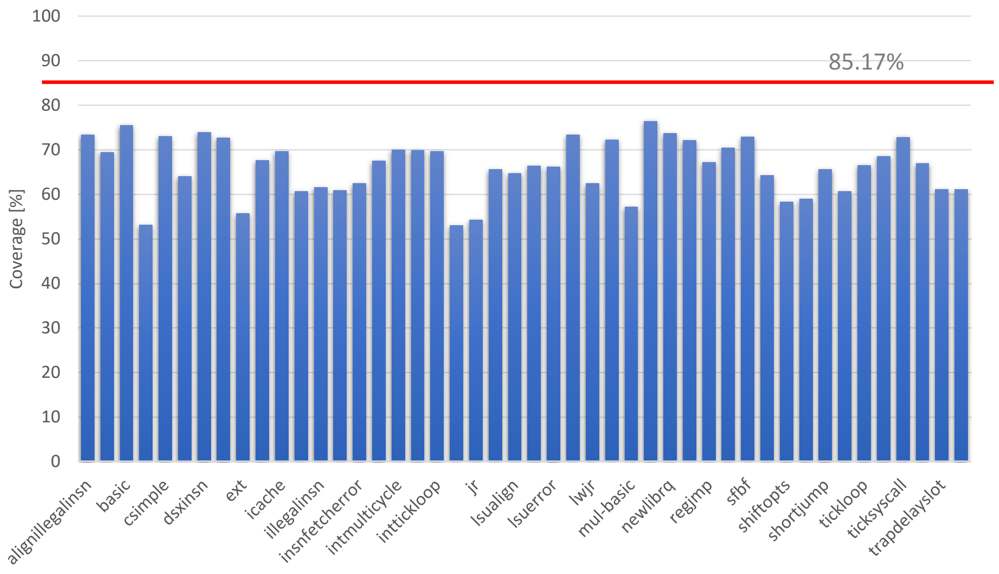


Figure 9.11: Individual coverage of each program on *mor1kx* core

In our approach, the tool for performing the task of parsing is Pyverilog [116]. It is a Python-based hardware design processing toolkit for Verilog HDL. The tool relies on Icarus,an open-source tool for performing the preprocessing. It flattens the hierarchy by implementing the `include` and `define` directives, producing the equivalent output related to such directives. Successively, Pyverilog reads the source code and generates Abstract Syntax Tree (AST) in the form of Python nested class objects. The parser is built upon PLY[1] which is used as a parser generator (compiler-compiler). PLY is a Python implementation of the Lex-Yacc lexical analyzer.

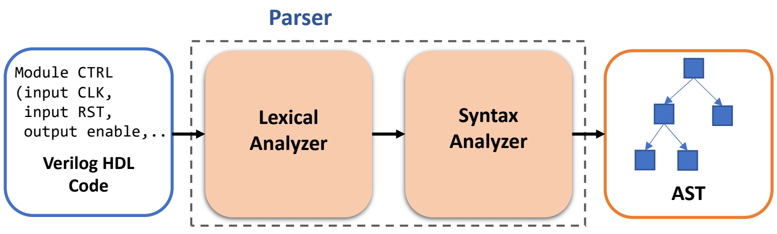


Figure 9.12: Pyverilog parser

[1]http://www.dabeaz.com/ply/

9.3.2 Experimental results with Support Vector Machine

The first set of experiments is intended to utilize SVM as a model to perform the classification of code sections given in the form of attributes belonging to the family of nodes. A common practice when working with supervised learning and data classification is to split the data set into three exclusive sets: training set, validation set and test set. However, by partitioning the available data into three sets, we drastically reduce the number of samples used for the learning phase. Consequently, such action might have a negative impact on the model's performance. Furthermore, the results can depend on a particular random choice when choosing/creating training and validation sets. A solution to this issue is using an approach called, k-fold cross-validation. It consists in splitting the training set into k smaller sets. The following procedure is followed for each of the k "folds": a model is trained using $k-1$ folds as input data for the training; the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure). Training/validation data and test data contain respectively, 80% and 20% of the complete data set.

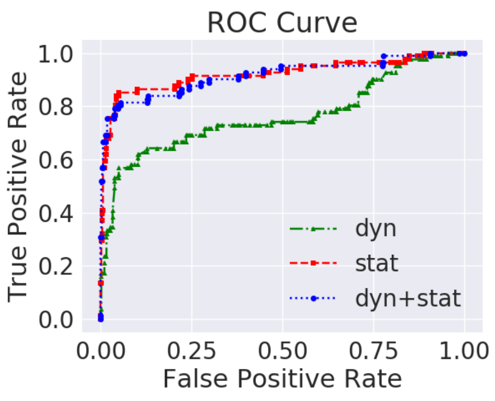
The average recall, precision, accuracy and F1-score [117] were calculated on cross-validation sets with 10 folds for each of the four classifiers and reported in the first four columns of the Table 9.1. Subsequently, the model was trained on the whole training data set (80%), with a particular model configuration. Next, we examined the models' strength by applying test data that had not been used previously, i.e., the remaining 20% of the initial complete set.

Table 9.1: Experimental results of the four SVM classifiers with different kernels and following metrics: Recall, Accuracy, Precision, and F1-score

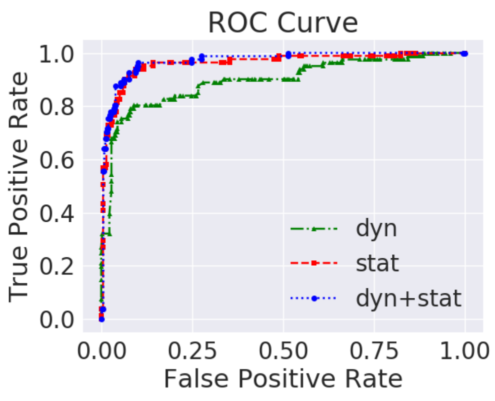
.

	Cross-Validation $10-fold$				Training [80%]		Test [20%]					
Kernel	Rec.	Prec.	Acc.	F1-sc.	Rec.	Prec.	Rec.	Prec.	Acc.	F1-sc.	TN FN	FP TP
Linear	0.79	0.60	0.87	0.69	0.80	0.64	0.81	0.61	0.87	0.70	314 15	42 66
Polynomial	0.49	0.90	0.90	0.63	0.57	0.97	0.64	0.95	0.93	0.76	353 29	3 52
RBF	**0.82**	**0.81**	**0.93**	**0.82**	**0.88**	**0.90**	**0.91**	**0.87**	**0.96**	**0.87**	**345** **7**	**11** **74**
Sigmoid	0.67	0.42	0.77	0.52	0.68	0.4	0.64	0.41	0.76	0.5	280 28	76 53

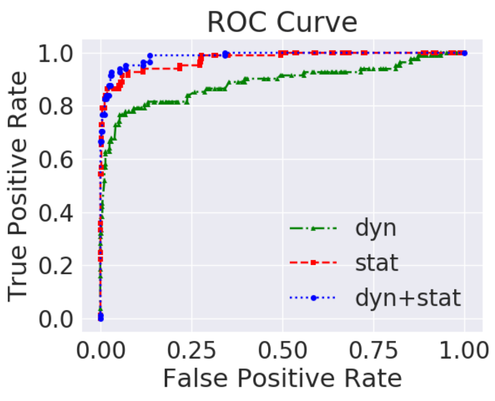
Receiver Operating Characteristic (ROC) curve is a graphical plot showing the influence of the threshold margin on the performance of the binary classifier system; it gives a trade-off between *sensitivity* (true positive rate) and *specificity* (1 - false positive rate). Classifiers with corresponding ROC curves closer to the top-left



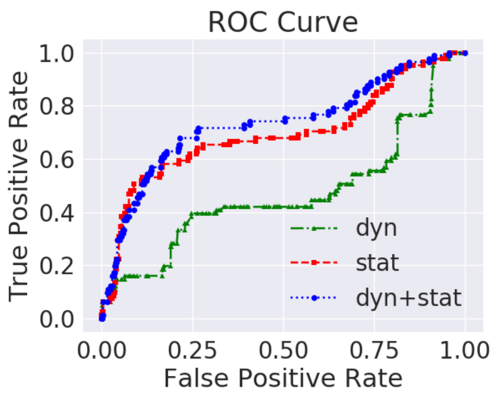
(a) ROC curve for linear kernel



(b) ROC curve for polynomial kernel



(c) ROC curve for rbf (gaussian) kernel



(d) ROC curve for sigmoid kernel

Figure 9.13: ROC curves for 4 different kernels including different set of extracted attributes (farther from the 45-diagonal, i.e., closer to the upper-left corner, the better)

corner indicate a better performance. On the other hand, the closer the curve comes to the 45-degree diagonal of the ROC space, which is used as a baseline for the random classifier, the less powerful the classifier becomes. Four Receiver Operating Characteristic (ROC) curves for linear, polynomial, rbf and sigmoid kernels are given in Fig. 9.13. They provide enough information to analyze the predictive power of a classifier and find the optimal threshold. Based on the aforementioned analysis, the threshold was set to 0.19. Moreover, the RBF kernel was chosen as the best one in terms of performance when compared to the other 3. This claim can be supported by observing the numerical values in Table 9.1, where we report recall and precision on the training set, and successively, recall, precision, accuracy and F1-score on the test set, together with the corresponding confusion matrix. Additionally, here we decided to split the attributes extracted from the set of nodes and examine their partial influence on the performance of the classifiers. As shown in the Fig. 9.13, we performed the training using exclusively static attributes (*stat*), then dynamic attributes (*dyn*) and finally, latter and former combined (*stat+dyn*). All of the classifiers clearly underperformed when relying only on the dynamic attributes. In case of the classifier with the RBF kernel, using the complete set of attributes instead of static attributes only resulted in improved classification power; in particular, 0.91 instead of 0.88 for recall, 0.87 instead of 0.8 for precision, 0.96 instead of 0.94 for accuracy and 0.87 instead of 0.84 for f1-score.

9.3.3 Experimental Results with Artificial Neural Networks

The second set of experiments is related to training the NN and evaluating its performance. For selecting the parameters of the NN during the training process, exhaustive experiments were run using LazyGrid[2], an open-source package that eases hyper-parameters tuning and comparing different machine-learning models.

To evaluate the effectiveness of the proposed NN approach , eight different experiments have been conducted, one for each group of HTs. To determine how the NN will generalize for an independent data set, we used cross validation technique. In other words, the NN has been trained on a set completely independent from the test one. The results show that even though the NN learns only on a category of HTs, it is able to discover different types as well.

In Table 9.2, we report for each training data set, the results obtained by evaluating the learning capabilities of the NN on the corresponding test sets. For a subset of benchmarks T_{k*} that is used later for test, we first train the NN on the whole set of all benchmarks ($\bigcup T$) *excluding* that one particular subset T_k and benchmarks derived from modifying it (T_{k*}). Confusion matrix terminology is used to present training and test performance given the predicted and expected classes for binary classification. The number of CFGs in a design (a set) is equal to the number of processes it contains. Finally, a false positive rate ($\frac{FP}{FP+TN}$) is given in the penultimate column of the table.

Elements of the confusion matrix in context of HT detection are given in Table 9.3, with their corresponding explanation and the effect from the user's point of view. The number of FPs (a non-trojan detected as trojan) should ideally be 0, i.e., in practice it should be kept as low as possible, together with the FNs. However, the obtained numbers (FP and FN) are still significantly low, given the total number of samples that have been evaluated (∼1.5k).

It is essential to outline that, first of all, the number of FPs remains significantly lower than the number of TNs, while being comparable to TPs. Therefore, checking all samples marked as positive (TPs + FPs), does not represent a huge effort.

[2]`https://github.com/glubbdubdrib/lazygrid`

Table 9.2: Experimental results of the NN

Training Dataset $\cup T\setminus$	Training performance				Test Dataset (n_{CFG})	Test performance				FP rate [%]	Det.
	TP	TN	FN	FP		TP	TN	FN	FP		
					$T_1(183)$	**23**	1493	7	1	1.1	✓
					$T_{11}(183)$	**18**	1493	6	1	1.1	✓
T_1*	260	1781	42	8	$T_{12}(183)$	**27**	1493	9	1	1.1	✓
					$T_{13}(184)$	**24**	1493	7	1	1.1	✓
					$T_{14}(185)$	**23**	1493	8	1	1.1	✓
					$T_2(182)$	**19**	1490	15	5	0.1	✓
T_2*	308	1764	13	14	$T_{21}(183)$	**24**	1491	16	6	0.1	✓
					$T_{22}(182)$	**19**	1493	11	5	0.1	✓
					$T_3(182)$	**8**	1487	1	10	0.3	✓
T_3*	358	1769	22	11	$T_{31}(183)$	**6**	1489	2	10	0.3	✓
					$T_{32}(184)$	**5**	1490	7	12	0.3	✓
					$T_4(182)$	**5**	1485	9	10	0.3	✓
T_4*	346	1770	25	16	$T_{41}(182)$	**4**	1487	10	10	0.3	✓
					$T_{42}(182)$	**40**	1494	11	10	0.3	✓
					$T_{43}(183)$	**3**	1487	11	10	0.3	✓
					$T_5(184)$	**35**	1487	7	8	1.4	✓
T_5*	305	1770	16	13	$T_{51}(184)$	**28**	1489	6	8	1.8	✓
					$T_{52}(184)$	**35**	1491	8	13	1.8	✓
					$T_{53}(185)$	**38**	1489	7	8	1.8	✓
					$T_6(181)$	**7**	1489	9	5	1.2	✓
T_6*	343	1641	30	9	$T_{61}(181)$	**9**	1492	5	5	1.1	✓
					$T_{62}(181)$	**9**	1486	15	5	1.3	✓
					$T_7(183)$	**23**	1494	2	3	1.0	✓
T_7*	358	1781	32	7	$T_{71}(183)$	**21**	1496	2	1	1.0	✓
					$T_{72}(184)$	**29**	1496	4	1	1.1	✓
					$T_8(181)$	**8**	1490	13	2	0.4	✓
T_8*	340	1656	34	5	$T_{81}(181)$	**10**	1493	8	2	0.4	✓
					$T_{82}(181)$	**10**	1493	8	7	0.5	✓

Classification	**Explanation**
True positive (TP)	Trojan code correctly recognized as malicious
True negative (TN)	Circuit code correctly considered safe
False positive (FP)	Safe circuit code believed to be malicious (i.e., a false alarm)
False negative (FN)	Malicious code that escaped detection (i.e., a major error)

Table 9.3: Meaning of the confusion matrix in the context of HT detection

Secondly, even though there are FNs, it does not mean some parts of malicious code escape the final analysis and remain undetected. As it can be seen from Fig. 9.14, a set of nodes marked in orange belongs to the HT (inserted malicious

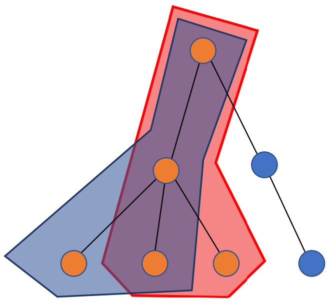


Figure 9.14: Set of nodes belonging to HT as TP and FN

code), while those in blue are not. Those nodes covered in red polygon are detected as malicious, therefore enter in TP category, while those in blue polygon are left undetected, belonging to the FN. By revealing one, others can be examined and by tracing back all TPs, a verification engineer can completely discover all of the maliciously inserted code. Thus, we can confirm that all of the Trojans in the test set have been discovered.

9.4 Chapter summary

This chapter addresses the problem of detecting RT-Level HTs resorting to ML-based techniques in a pipelined CPU. A mixed approach consisting of static and dynamic model analysis is presented where robust machine learning algorithms are used to perform classification. Experimental results prove the technique's efficacy: no HT was left undetected, showing that this technique could be used with similar complex industrial designs, in an automatized manner, reducing both effort and time. The in-house tool was built and integrated into the whole flow to provide a fast and efficient analysis. It is adjustable for other commercial tools that can simulate the design and generate a code coverage report. Additional items and rules can be introduced for feature extraction, as well as different CFG node environments to create the classification input. The final flow processing of the input, given as an RTL behavioral model, includes logic simulation, CFG extraction and annotation, and input formatting. The final result of the evaluation is the list of suspicious locations in the code. By "out-of-sample" testing it has been shown that the NN method is able to identify all HTs embedded in a complex design aggravating the detection process. Additionally, the performance of four different SVM classifiers was evaluated. The one using the RBF kernel was shown to generalize very well. Comparing the two models in terms of performance, SVM RBF kernel is more successful in discovering the set of nodes that are marked as malicious and also takes less time to train. Nevertheless, both of the approaches in the end detect each and every HT as an entity, following the discussion that a set of nodes might represent only one section of a HT. The relatively small amount of training data might be responsible for a poorer performance of the NN.

However, the principal limitation of the approach is that still some manual post-processing is required to analyze suspicious code and decide if it is a malicious insertion or not. Trojans evolve in structure and their location is unpredictable. A lot of effort is being invested into their classification and development. Since the supervised type of learning is used to train both SVM and ANN, it is uncertain how their classification performance will change with new types of HTs. Nevertheless, such new malicious insertions may be included into the training set.

Summary of Part II

The second and last part of the thesis deals with hardware security threats, in particular malicious insertions named Hardware Trojans. Due to the decentralization of the production flow, different phases have become more vulnerable and a possible target for an attacker to disrupt the security integrity of the product. Security concerns encouraged both industry and academia to dedicate more attention to this issue and come with countermeasures.

Publicly available HT benchmarks that are used for validating detection methodologies at the RT-level are not many and are obsolete, i.e., they do not reflect a complexity that can be found in modern, real industrial case processors used for cutting-edge applications. Therefore, a set of HTs has been implanted in different modules of a pipelined microprocessor core *mor1kx* and published. New benchmarks will hopefully facilitate the validation of novel detection approaches.

Furthermore, despite the considerable effort that has been invested in developing new detection methodologies, the growing complexity of modern devices always calls for sharper detection methodologies. This is especially true early in the design cycle, when an attacker may insert malicious circuitry at register transfer (RT) or gate level that may remain undetected even in the next generations of the device.

In this regard, a pre-silicon, simulation-based technique to detect HTs that exploits well-established machine learning algorithms is proposed. The validity of the approach has been demonstrated on the *AutoSoC* CPU, an industrial-grade, safety-oriented, automotive benchmark suite. Experimental results demonstrate the applicability and effectiveness of the approach: the proposed technique is highly accurate in pinpointing suspicious code sections. None of the HTs from the set has been left undetected.

Conclusions and Recommendations for Future Research

This thesis is composed out of two main parts. While the first part deals with different issues related to IEEE 1687 RSNs such as reliability and aging, permanent fault detection and identification, design validation and description equivalence checking, the second one focuses on hardware security, particularly malicious insertions – Hardware Trojans, by introducing a set of newly developed RT-level benchmarks and Machine Learning-based detection techniques.

In Chapter 1 background related to the first part of the thesis has been provided with a particular focus on IEEE 1687 standard and related constructs and features it supports. It also gives an overview of the state-of-the-art and basic information about ITC'16 benchmarks that were used to evaluate and validate the proposed methodologies.

Chapter 2 describes three different approaches to generate efficient sequences to test reconfigurable modules in an RSN. Two (basic and enhanced versions) can be defined as semi-formal because the FSA that models the circuit is exact but incomplete, and the search procedure is based on a greedy algorithm. Experimental results on the ITC'16 benchmark suite clearly demonstrate the effectiveness of the approach: the proposed technique can achieve better results with less computation effort than previous heuristics. The technique may be easily extended to handle different fault models and more complex scenarios, and experts' knowledge could be exploited by tweaking the FSA states and input alphabet. The third approach to minimize the test time is based on evolutionary computation Additionally, the problem of finding suitable test configurations has been converted into a circuit suitable for applying the automatic test pattern generation procedure. An optimized transition function and some techniques for post-processing the solution delivered by the evolutionary engine have also been presented. Experimental results on the standard set of benchmark networks show the effectiveness of the proposed approach since the test time has been reduced up to 27% in 14 out of 16 cases, particularly impacting the test time for large networks.

In Chapter 3 a technique to diagnose permanent faults in an RSN has been explained in detail and experimental results were provided for a subset of ITC'16 benchmarks. The approach resorts to an FSA model of the circuit and a greedy search algorithm. Experimental results demonstrate that the presented approach outperforms the previous ones in terms of number of clock cycles required to run the generated diagnostic sequence. Furthermore, this technique can be applied to a wide range of network types of different complexity since for all of the test cases and benchmark networks full diagnostic coverage has been reached while keeping the computation effort under control. Future work should focus on extending the technique to support different fault models.

A methodology for assessment and mitigation of NBTI aging-induced delays in logic paths within IEEE 1687 IJTAG Reconfigurable Scan Networks is described in Chapter 4. While RSNs are commonly used to provide fault management and embedded instrumentation access, such as safety mechanisms, in advanced safety- and mission-critical electronic systems, a failure in such infrastructure itself has a high severity. The methodology is based on a scalable hierarchical (transistor-to-architecture) modelling of the NBTI impact on timing-critical logic paths in RSN implementations. The evaluation implies analysis of gate input signal probabilities based on the configurations and test data selected for the RSN infrastructure. The details of the methodology are demonstrated by a case study on an example RSN and the feasibility and efficiency are validated by experiments on a subset of ITC2016 RSN benchmarks. The experimental results demonstrate that RSNs can be impacted by significant NBTI-induced logic path delays and a simple proposed mitigation technique can reduce such delays up to 2.6 times. The future work is aimed at a comparative analysis of aging in the RSN gates and the functional part of the circuit.

To ensure there is no mismatch between prototypical device and initial specifications, product life-cycle requires performing (post-silicon) validation before going into the mass production. Such additional effort prevents rendering the whole infrastructure inoperable and avoids enormous re-design costs. In Chapter 5 a mismatch model for post-silicon validation of RSNs. Furthermore, two algorithms have been developed for generating configuration patterns to detect the set of considered mismatches. The ITC2016 benchmark networks were used to evaluate the proposed methodology. It was found that in all cases full detection coverage has been reached. For the detection procedure based on the active path length comparison, the tool generates a list of undetectable mismatches. Furthermore, mismatch model is easily extendable, due to the nature of the problem and the internal model of the network extracted by the tool.

Chapter 6 addresses the problem of detecting inconsistencies between ICL and RTL models of RSNs, resorting to simulation-based verification. Automatically generated test-benches for stimulating the RTL model are based on the patterns used for post-silicon validation of networks. The approach has been verified through a series of experiments, obtaining extremely high detection coverage with reduced execution time. Test escapes have also been identified as pathological network configurations. Future work should extend the experimental verification to other error models such as, for instance, ICL connection errors, and to reinforce debug capabilities. Another direction would be the in-depth analysis of test escapes configurations to devise algorithms able to detect potentially untestable networks and warn the designer.

The research work related to IJTAG opens several research directions. The techniques to test and diagnose faults may be easily extended to handle different fault models and more complex scenarios, and experts' knowledge could be exploited by tweaking the FSA states and input alphabet. As for the NBTI aging effect and mitigation in an RSN, the future work could be aimed at a comparative analysis of aging in the RSN gates and the functional part of the circuit. The equivalence checking technique should be experimentally verified to other error models such as, for instance, ICL connection errors, and reinforcing debug capabilities. Another open research direction is the in-depth analysis of test escapes configurations to devise algorithms able to detect potentially untestable networks and warn the designer. Exploring the structure of RSNs and creating a set of Design-for-Test rules and recommendations would be extremely useful. A tool that is able to generate the information intended for helping the designer after analysing the ICL and PDL would resolve many issues; problems that are a consequence of symmetry in the structure of RSNs and related to test, diagnosis, validation and verification would be prevented.

Chapter 7 provides background information on hardware security with basic concepts and terminology related to Hardware Trojans as well as state-of-the-art overview for both design methodologies and detection techniques.

To deal with the issue of obsolete HT benchmarks that do not correspond to state-of-the-art devices used in real applications, a set of 8 new principle HTs and their 20 modifications for a pipelined processor core has been introduced in Chapter 8. The proposed HTs have been injected into very different parts of the processor design. They differ in the trigger and payload. The synthesis reports show the negligible impact that the introduced modifications have on area, power and frequency. Furthermore, the set of benchmarks could be extremely useful for validating dynamic HT detection methodologies since the core is open-source and in the near future the HTs will be also publicly available. The proposed set of HTs are easily modifiable and allows to create even more complex set of trigger conditions, while the space for inserting payloads is quite vast and allows to execute different type of malicious functions. That is why future work should be focused on diversifying and developing the HT Benchmarks Library even further.

Although most of the detection techniques work at the gate level, shifting the detection of HTs inserted at RTL to the gate level would result in increased design and verification costs. That is why a new technique was developed to detect

RT-Level HTs resorting to ML-based techniques in a pipelined CPU. Chapter 9 introduces the aforementioned mixed approach that is composed of both static and dynamic analysis of the model. The in-house tool was built and integrated into the whole flow to provide fast and efficient analysis. Flow processing of the input, given as an RTL behavioural model includes logic simulation and CFG extraction. The final result of the evaluation is the list of suspicious locations in the code. By “out-of-sample” testing it was shown that the NN method can identify all HTs embedded in a complex design aggravating the detection process. Additionally, the performance of four different SVM classifiers has been evaluated. The one using RBF kernel was shown to generalize very well. Comparing two models in terms of performance, SVM RBF kernel is more successful in discovering the set of nodes that is marked as malicious and also takes less time to train. Nevertheless, both of the approaches, in the end, detect each and every HT as an entity, following the discussion that a set of nodes might represent only one section of a HT. The relatively small amount of training data might be responsible for a poorer performance of the NN. However, the principal limitation of the approach is that still some manual post-processing is required to analyze suspicious code and decide if it is a malicious insertion.

The proposed set of HTs are easily modifiable and allow to create even more complex set of trigger conditions, while the space for inserting payloads is quite vast and allows to execute different types of malicious functions. That is why future work should be focused on diversifying and developing the HT Benchmarks Library even further. It is an utterly important objective, given the constant race of the *defenders* against the *attackers* and the need of complex benchmarks that will help validate new methodologies for detection and analysis of HTs. As for the HT detection technique, examining and extending the set of properties used for the analysis and validating the approach further with some other HTs represent potential research topics.

Bibliography

[1] *Nanotechnology – Energy, Transportation, Medical and Healthcare, Electronics and IT*. [Online]. Available: https://www.nano.gov/you/nanotechnology-benefits.

[2] C. Claeys, "Trends and challenges in micro- and nanoelectronics for the next decade", in *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2012*, 2012, pp. 37–42.

[3] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and Trends in Modern SoC Design Verification", *IEEE Design Test*, vol. 34, no. 5, pp. 7–22, 2017. DOI: 10.1109/MDAT.2017.2735383.

[4] I. Verbauwhede, "Security Adds an Extra Dimension to IC Design: Future IC Design Must Focus on Security in Addition to Low Power and Energy", *IEEE Solid-State Circuits Magazine*, vol. 9, no. 4, pp. 41–45, 2017. DOI: 10.1109/MSSC.2017.2745799.

[5] X. Lai, A. Balakrishnan, T. Lange, M. Jenihhin, T. Ghasempouri, J. Raik, and D. Alexandrescu, "Understanding multidimensional verification: Where functional meets non-functional", *Microprocessors and Microsystems*, vol. 71, p. 102 867, 2019, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102867. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933119300250.

[6] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th. USA: Addison-Wesley Publishing Company, 2010, ISBN: 0321547748.

[7] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*. USA: CRC Press, Inc., 2006, ISBN: 0849379237.

[8] E. Brunvand, *Digital VLSI Chip Design with Cadence and Synopsys CAD Tools*, 1st. USA: Addison-Wesley Publishing Company, 2009, ISBN: 0321547993.

[9] G. Gielen and R. Rutenbar, "Computer-aided design of analog and mixed-signal integrated circuits", *Proceedings of the IEEE*, vol. 88, no. 12, pp. 1825–1854, 2000. DOI: 10.1109/5.899053.

[10] 2. S. 112th Congress, “Inquiry into counterfeit electronic parts in the department of defense supply chain”, *in Report, Committee on Armed Services*, vol. U.S. Government Printing Office, pp. 112–167, 2012. [Online]. Available: https://www.armed-services.senate.gov/imo/media/doc/Counterfeit-Electronic-Parts.pdf.

[11] Y. Alkabani and F. Koushanfar, “Active control and digital rights management of integrated circuit IP cores”, Jan. 2008, pp. 227–234. DOI: 10.1145/1450095.1450129.

[12] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, “Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection”, in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 189–195. DOI: 10.1109/FPL.2007.4380646.

[13] G. Suh, C. O’Donnell, I. Sachdev, and S. Devadas, “Design and implementation of the AEGIS single-chip secure processor using physical random functions”, in *32nd International Symposium on Computer Architecture (ISCA05)*, 2005, pp. 25–36. DOI: 10.1109/ISCA.2005.22.

[14] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, “The EM Side-Channel(s)”, Berlin, Heidelberg: Springer-Verlag, 2002, ISBN: 3540004092.

[15] I. Polian, F. Altmann, T. Arul, C. Boit, R. Brederlow, L. Davi, R. Drechsler, N. Du, T. Eisenbarth, T. Guneysu, S. Hermann, M. Hiller, R. Leupers, F. Merchant, T. Mussenbrock, S. Katzenbeisser, A. Kumar, W. Kunz, T. Mikolajick, V. Pachauri, J.-P. Seifert, F. S. Torres, and J. Trommer, “Nano Security: From Nano-Electronics to Secure Systems”, in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1334–1339. DOI: 10.23919/DATE51398.2021.9474187.

[16] M. Rostami, F. Koushanfar, and R. Karri, “A Primer on Hardware Security: Models, Methods, and Metrics”, *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014. DOI: 10.1109/JPROC.2014.2335155.

[17] C. E. Stroud, L.-T. Wang, and Y.-W. Chang, “CHAPTER 1 - Introduction”, in *Electronic Design Automation*, Boston: Morgan Kaufmann, 2009, pp. 1–38, ISBN: 978-0-12-374364-0. DOI: https://doi.org/10.1016/B978-0-12-374364-0.50008-4.

[18] R. Bentley, “Foreword for "Formal Verification: An Essential Toolkit for Modern VLSI Design"”, in *Formal Verification*, E. Seligman, T. Schubert, and M. V. A. K. Kumar, Eds., Boston: Morgan Kaufmann, 2015, pp. xiii–xvi, ISBN: 978-0-12-800727-3. DOI: https://doi.org/10.1016/B978-0-12-800727-3.00016-2.

[19] "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device", *IEEE Std 1687-2014*, pp. 1–283, 2014. DOI: 10.1109/IEEESTD.2014.6974961.

[20] A. Syed, C. Timothy, L. Shrikant, S. Shungo, and B. McKinley, "Globality and Complexity of the Semiconductor Ecosystem", in *Accenture*. [Online]. Available: https://www.accenture.com/_acnmedia/PDF-119/Accenture-Globality-Semiconductor-Industry.pdf.

[21] K. Xiao *et al.*, "Hardware Trojans: Lessons Learned after One Decade of Research", *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, May 2016.

[22] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson, "Automatic generation of stimuli for fault diagnosis in IEEE 1687 networks", in *2016 22nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2016, pp. 167–172.

[23] F. G. Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Access time analysis for IEEE P1687", *IEEE Transactions on Computers*, vol. 61, no. 10, pp. 1459–1472, 2012.

[24] ——, "Design automation for IEEE P1687", in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, IEEE, 2011, pp. 1–6.

[25] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson, "On the testability of IEEE 1687 networks", in *2015 IEEE 24th Asian Test Symposium (ATS)*, IEEE, 2015, pp. 211–216.

[26] R. Cantoro, M. Palena, P. Pasini, and M. Sonza Reorda, "Test Time Minimization in Reconfigurable Scan Networks", in *2016 25th IEEE Asian Test Symposium (ATS)*, IEEE, 2016, pp. 119–124.

[27] R. Cantoro, L. San Paolo, M. Sonza Reorda, and G. Squillero, "New techniques for reducing the duration of Reconfigurable Scan Network test", in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2018 IEEE 21th International Symposium on*, IEEE, 2018.

[28] R. Cantoro, F. G. Zadegan, M. Palena, P. Pasini, E. Larsson, and M. S. Reorda, "Test of Reconfigurable Modules in Scan Networks", *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1806–1817, Dec. 2018, ISSN: 0018-9340. DOI: 10.1109/TC.2018.2834915.

[29] E. Larsson and K. šibin, "Fault management in an IEEE P1687 (IJTAG) environment", in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012, pp. 7–7. DOI: 10.1109/DDECS.2012.6219013.

[30] K. Shibin, S. Devadze, and A. Jutman, “Asynchronous Fault Detection in IEEE P1687 Instrument Network”, in *2014 IEEE 23rd North Atlantic Test Workshop*, 2014, pp. 73–78. DOI: 10.1109/NATW.2014.24.

[31] K. Petersén, D. Nikolov, U. Ingelsson, G. Carlsson, F. G. Zadegan, and E. Larsson, “Fault injection and fault handling: An MPSoC demonstrator using IEEE P1687”, in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, 2014, pp. 170–175. DOI: 10.1109/IOLTS.2014.6873664.

[32] A. Jutman, S. Devadze, and K. Shibin, “Effective Scalable IEEE 1687 Instrumentation Network for Fault Management”, *IEEE Design Test*, vol. 30, no. 5, pp. 26–35, Oct. 2013, ISSN: 2168-2356. DOI: 10.1109/MDAT.2013.2278535.

[33] F. G. Zadegan, D. Nikolov, and E. Larsson, “On-Chip Fault Monitoring Using Self-Reconfiguring IEEE 1687 Networks”, *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 237–251, 2018. DOI: 10.1109/TC.2017.2731338.

[34] ——, “A self-reconfiguring IEEE 1687 network for fault monitoring”, in *2016 21th IEEE European Test Symposium (ETS)*, 2016, pp. 1–6. DOI: 10.1109/ETS.2016.7519288.

[35] A. Jutman, K. Shibin, and S. Devadze, “Reliable health monitoring and fault management infrastructure based on embedded instrumentation and IEEE 1687”, in *2016 IEEE AUTOTESTCON*, IEEE, Sep. 2016, pp. 1–10. DOI: 10.1109/AUTEST.2016.7589605.

[36] 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 (TODAES)*, vol. 20, no. 2, p. 30, 2015.

[37] F. G. Zadegan, R. Krenz-Baath, and E. Larsson, “Upper-bound computation for optimal retargeting in IEEE1687 networks”, in *Test Conference (ITC), 2016 IEEE International*, IEEE, 2016, pp. 1–10.

[38] M. A. Kochte, R. Baranowski, M. Sauer, B. Becker, and H.-J. Wunderlich, “Formal verification of secure reconfigurable scan network infrastructure”, in *Test Symposium (ETS), 2016 21th IEEE European*, IEEE, 2016, pp. 1–6.

[39] D. Ull, M. Kochte, and H. J. Wunderlich, “Structure-Oriented Test of Reconfigurable Scan Networks”, in *2017 IEEE 26th Asian Test Symposium (ATS)*, IEEE, Nov. 2017, pp. 127–132. DOI: 10.1109/ATS.2017.34.

[40] J. Dworak, A. Crouch, J. Potter, A. Zygmontowicz, and M. Thornton, “Don’t forget to lock your SIB: hiding instruments using P1687”, in *2013 IEEE International Test Conference (ITC)*, Sep. 2013, pp. 1–10.

[41] A. Zygmontowicz, J. Dworak, A. Crouch, and J. Potter, “Making it harder to unlock an LSIB: Honeytraps and misdirection in a P1687 network”, in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2014, pp. 1–6.

[42] M. A. Kochte, M. Sauer, L. R. Gomez, P. Raiola, B. Becker, and H. Wunderlich, “Specification and verification of security in reconfigurable scan networks”, in *2017 22nd IEEE European Test Symposium (ETS)*, May 2017, pp. 1–6.

[43] R. Baranowski, M. A. Kochte, and H. Wunderlich, “Securing Access to Reconfigurable Scan Networks”, in *2013 22nd Asian Test Symposium*, Nov. 2013, pp. 295–300.

[44] A. Atteya, M. A. Kochte, M. Sauer, P. Raiola, B. Becker, and H. Wunderlich, “Online prevention of security violations in reconfigurable scan networks”, in *2018 IEEE 23rd European Test Symposium (ETS)*, May 2018.

[45] A. Tšertov, A. Jutman, S. Devadze, M. Sonza Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, “A suite of IEEE 1687 benchmark networks”, in *2016 IEEE International Test Conference (ITC)*, IEEE, 2016, pp. 1–10.

[46] A. T. Dahbura, M. U. Uyar, and C. W. Yau, “An optimal test sequence for the JTAG/IEEE P1149. 1 test access port controller”, in *International Test Conference, 1989. Proceedings. Meeting the Tests of Time*, IEEE, 1989, pp. 55–62.

[47] K.-J. Lee and M. A. Breuer, “A universal test sequence for CMOS scan registers”, in *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, IEEE, 1990, pp. 28–5.

[48] S. Maka and E. J. McCluskey, “ATPG for scan chain latches and flip-flops”, in *1997 15th IEEE VLSI Test Symposium*, IEEE, 1997, pp. 364–369.

[49] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz, “On the detectability of scan chain internal faults an industrial case study”, in *VLSI Test Symposium, 2008. VTS 2008. 26th IEEE*, IEEE, 2008, pp. 79–84.

[50] A. Tšertov, A. Jutman, S. Devadze, M. S. Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, “A suite of IEEE 1687 benchmark networks”, in *2016 IEEE International Test Conference (ITC)*, IEEE, 2016, pp. 1–10.

[51] R. Cantoro, A. Damljanovic, M. Sonza Reorda, and G. Squillero, “A Semi-Technique to Generate Effective Test Sequences for Reconfigurable Scan Networks”, in *Test Conference in Asia (ITC-Asia), 2017 International*, IEEE, 2018.

[52] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2015. DOI: 10.1007/978-3-662-44874-8. [Online]. Available: https://doi.org/10.1007/978-3-662-44874-8.

[53] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: the μGP toolkit*. Springer Science & Business Media, 2011.

[54] *Genetic operators*, https://sourceforge.net/p/ugp3/wiki/Genetic%20operators/.

[55] S. Kundu, S. Chattopadhyay, I. Sengupta, and R. Kapur, "Scan Chain Masking for Diagnosis of Multiple Chain Failures in a Space Compaction Environment", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1185–1195, Jul. 2015, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2333691.

[56] J. Ye, Y. Huang, Y. Hu, W. T. Cheng, R. Guo, L. Lai, T. P. Tai, X. Li, W. Changchien, D. M. Lee, J. J. Chen, S. C. Eruvathi, K. K. Kumara, C. Liu, and S. Pan, "Diagnosis and Layout Aware (DLA) Scan Chain Stitching", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 3, pp. 466–479, Mar. 2015, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2313563.

[57] Y. Huang, R. Guo, W. T. Cheng, and J. C. M. Li, "Survey of Scan Chain Diagnosis", *IEEE Design Test of Computers*, vol. 25, no. 3, pp. 240–248, May 2008, ISSN: 0740-7475. DOI: 10.1109/MDT.2008.83.

[58] Y. Huang, X. Fan, H. Tang, M. Sharma, W. T. Cheng, B. Benware, and S. M. Reddy, "Distributed dynamic partitioning based diagnosis of scan chain", in *2013 31st IEEE VLSI Test Symposium (VTS)*, Apr. 2013, pp. 1–6. DOI: 10.1109/VTS.2013.6548916.

[59] W. H. Lo, A. C. Hsieh, C. M. Lan, M. H. Lin, and T. Hwang, "Utilizing Circuit Structure for Scan Chain Diagnosis", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2766–2778, Dec. 2014, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2294712.

[60] H. Chen, Z. Qi, L. Wang, and C. Xu, "A scan chain optimization method for diagnosis", in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct. 2015, pp. 613–620. DOI: 10.1109/ICCD.2015.7357172.

[61] R. Cantoro, A. Damljanovic, M. Sonza Reorda, and G. Squillero, "A New Technique to Generate Test Sequences for Reconfigurable Scan Networks", in *2018 IEEE International Test Conference (ITC)*, IEEE, Oct. 2018.

[62] F. Ahmed and L. Milor, "Reliable cache design with on-chip monitoring of NBTI degradation in SRAM cells using BIST", in *2010 28th VLSI Test Symposium (VTS)*, Apr. 2010, pp. 63–68.

[63] C. Ferri, D. Papagiannopoulou, R. I. Bahar, and A. Calimera, "NBTI-aware data allocation strategies for scratchpad memory based embedded systems", in *2011 12th Latin American Test Workshop (LATW)*, Mar. 2011, pp. 1–6.

[64] H. Kükner *et al.*, "Comparison of Reaction-Diffusion and Atomistic Trap-Based BTI Models for Logic Gates", *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 1, pp. 182–193, Mar. 2014.

[65] M. Jenihhin *et al.*, "Identification and Rejuvenation of NBTI-Critical Logic Paths in Nanoscale Circuits", *Journal of Electronic Testin*, vol. 32, no. 3, pp. 273–289, Jun. 2016.

[66] S. Khan and S. Hamdioui, "Modeling and mitigating NBTI in nanoscale circuits", in *2011 IEEE 17th International On-Line Testing Symposium*, Jul. 2011, pp. 1–6.

[67] A. Tiwari and J. Torrellas, "Facelift: Hiding and slowing down aging in multicores", in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, Nov. 2008, pp. 129–140.

[68] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, "NBTI-Aware Synthesis of Digital Circuits", in *2007 44th ACM/IEEE Design Automation Conference*, Jun. 2007, pp. 370–375.

[69] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The NBTI-Aware Processor", in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec. 2007, pp. 85–96.

[70] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula, "Predictive Modeling of the NBTI Effect for Reliable Design", in *IEEE Custom Integrated Circuits Conference 2006*, Sep. 2006, pp. 189–192.

[71] H. Kükner, S. Khan, P. Weckx, P. Raghavan, S. Hamdioui, B. Kaczer, F. Catthoor, L. Van der Perre, R. Lauwereins, and G. Groeseneken, "Comparison of Reaction-Diffusion and Atomistic Trap-Based BTI Models for Logic Gates", *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 1, pp. 182–193, 2014. DOI: 10.1109/TDMR.2013.2267274.

[72] W. Wang *et al.*, "Compact Modeling and Simulation of Circuit Reliability for 65-nm CMOS Technology", *IEEE Transactions on Device and Materials Reliability*, vol. 7, no. 4, pp. 509–517, Dec. 2007.

[73] A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, and V. Tihhomirov, "A scalable model based RTL framework zamiaCAD for static analysis", in *2012 IEEE IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, 2012, pp. 171–176.

[74] "Report on structural analysis, verification and optimization methodology for ICL networks", in *EU FP7 BASTION project report*, Feb. 2016, pp. 1–42. [Online]. Available: https://cordis.europa.eu/docs/projects/cnect/1/619871/080/deliverables/001-BASTIOND23v204.pdf.

[75] A. Tsertov, A. Jutman, K. Shibin, and S. Devadze, "IEEE 1687 Compliant Ecosystem for Embedded Instrumentation Access and In-Field Health Monitoring", in *AUTOTESTCON 2018*, Sep. 2018, pp. 1–9. DOI: 10.1109/AUTEST.2018.8532559.

[76] R. Cantoro, A. Damljanovic, M. Sonza Reorda, and G. Squillero, "A New Technique to Generate Effective Test Sequences for Reconfigurable Scan Networks", in *IEEE International Test Conference (ITC), 2018*.

[77] "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device", *IEEE Std 1687-2014*, pp. 1–283, Dec. 2014. DOI: 10.1109/IEEESTD.2014.6974961.

[78] J. Zhang and Q. Xu, "On hardware Trojan design and implementation at register-transfer level", 2013.

[79] Y. Jin, N. Kupp, and Y. Makris, "Experiences in Hardware Trojan design and implementation", in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57. DOI: 10.1109/HST.2009.5224971.

[80] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development", in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 471–474.

[81] S. Yu, W. Liu, and M. O'Neill, "An Improved Automatic Hardware Trojan Generation Platform", in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 302–307. DOI: 10.1109/ISVLSI.2019.00062.

[82] H. Salmani, "COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist", *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 2, pp. 338–350, 2017. DOI: 10.1109/TIFS.2016.2613842.

[83] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically", in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 159–172.

[84] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building Stealthy and Malicious Hardware", in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 64–77.

[85] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, “Scalable Hardware Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution”, in *2018 IEEE International Test Conference (ITC)*, 2018, pp. 1–10.

[86] Y. Lyu, A. Ahmed, and P. Mishra, “Automated Activation of Multiple Targets in RTL Models using Concolic Testing”, in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 354–359.

[87] L. Piccolboni, A. Menon, and G. Pravadelli, “Efficient Control-Flow Subgraph Matching for Detecting Hardware Trojans in RTL Models”, *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017, ISSN: 1539-9087. DOI: 10.1145/3126552. [Online]. Available: https://doi.org/10.1145/3126552.

[88] F. Demrozi, R. Zucchelli, and G. Pravadelli, “Exploiting sub-graph isomorphism and probabilistic neural networks for the detection of hardware Trojans at RTL”, in *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2017, pp. 67–73.

[89] D. F. Specht, “Probabilistic Neural Networks”, *Neural Netw.*, vol. 3, no. 1, pp. 109–118, Jan. 1990.

[90] S. King *et al.*, “Designing and Implementing Malicious Hardware.”, Jan. 2008.

[91] Y. Jin, N. Kupp, and Y. Makris, “Experiences in Hardware Trojan design and implementation”, in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57.

[92] B. Shakya *et al.*, “Benchmarking of Hardware Trojans and Maliciously Affected Circuits”, *Journal of Hardware and Systems Security*, 2017.

[93] R. Elnaggar and K. Chakrabarty, “Machine Learning for Hardware Security: Opportunities and Risks”, *Journal of Electronic Testing*, 2018.

[94] C. Bao, D. Forte, and A. Srivastava, “On application of one-class SVM to reverse engineering-based hardware Trojan detection”, in *Fifteenth International Symposium on Quality Electronic Design*, 2014, pp. 47–54.

[95] W. Li, Z. Wasson, and S. A. Seshia, “Reverse engineering circuits using behavioral pattern mining”, in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2012, pp. 83–88.

[96] K. Hasegawa, M. Yanagisawa, and N. Togawa, “Trojan-feature extraction at gate-level netlists and its application to hardware-Trojan detection using random forest classifier”, in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[97] E. Zhou *et al.*, “A Novel Detection Method for Hardware Trojan in Third Party IP Cores”, in *2016 International Conference on Information System and Artificial Intelligence (ISAI)*.

[98] Y. Liu, Y. Jin, A. Nosratinia, and Y. Makris, “Silicon Demonstration of Hardware Trojan Design and Detection in Wireless Cryptographic ICs”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1506–1519, 2017.

[99] S. Wang *et al.*, “Hardware Trojan detection based on ELM neural network”, in *2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI)*, 2016, pp. 400–403.

[100] D. K. Pradhan and I. G. Harris, *Practical Design Verification*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511626913.

[101] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, 1st. Springer Publishing Company, Incorporated, 2007, ISBN: 0387739920.

[102] S. Tasiran and K. Keutzer, “Coverage Metrics for Functional Validation of Hardware Designs”, *IEEE Des. Test*, vol. 18, no. 4, pp. 36–45, Jul. 2001, ISSN: 0740-7475. DOI: 10.1109/54.936247. [Online]. Available: https://doi.org/10.1109/54.936247.

[103] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.”, en, *Psychological Review*, vol. 65, no. 6, 1958.

[104] C. C. Aggarwal, *Neural Networks and Deep Learning, A Textbook*. Springer, 2018, p. 497, ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0.

[105] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning”, *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539. [Online]. Available: https://doi.org/10.1038/nature14539.

[106] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV ’15, USA: IEEE Computer Society, 2015, pp. 1026–1034, ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.123. [Online]. Available: https://doi.org/10.1109/ICCV.2015.123.

[107] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge”, *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y.

[108] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90.

[109] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A Training Algorithm for Optimal Margin Classifiers", in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 144–152, ISBN: 089791497X. DOI: 10.1145/130385.130401. [Online]. Available: https://doi.org/10.1145/130385.130401.

[110] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Berlin, Heidelberg: Springer-Verlag, 1995, ISBN: 0387945598.

[111] "IEEE Standard for VHDL Language Reference Manual", *IEEE Std 1076-2019*, pp. 1–673, 2019. DOI: 10.1109/IEEESTD.2019.8938196.

[112] "IEEE Standard for Verilog Hardware Description Language", *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006. DOI: 10.1109/IEEESTD.2006.99495.

[113] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and Unsupervised Discretization of Continuous Features", in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds., San Francisco (CA): Morgan Kaufmann, 1995, pp. 194–202, ISBN: 978-1-55860-377-6. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50032-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558603776500323.

[114] A. Damljanovic, A. Ruospo, E. Sanchez, and G. Squillero, "A Benchmark Suite of RT-level Hardware Trojans for Pipelined Microprocessor Cores", *24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2021.

[115] F. Silva *et al.*, "Special Session: AutoSoC - A Suite of Open-Source Automotive SoC Benchmarks", Apr. 2020, pp. 1–9. DOI: 10.1109/VTS48691.2020.9107599.

[116] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL", in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040, Springer International Publishing, 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42.

[117] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Aug. 2019, ISBN: 1492032646.

[118] B. Shakya, M. T. He, H. Salmani, D. Forte, S. Bhunia, and M. M. Tehranipoor, "Benchmarking of Hardware Trojans and Maliciously Affected Circuits", *Journal of Hardware and Systems Security*, vol. 1, pp. 85–102, 2017.

[119] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson, "On the diagnostic analysis of IEEE 1687 networks", in *Test Symposium (ETS), 2016 21th IEEE European*, IEEE, 2016, pp. 1–2.

[120] S. Narayanan and M. A. Breuer, "Reconfigurable scan chains: A novel approach to reduce test application time", in *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, IEEE Computer Society Press, 1993, pp. 710–715.

[121] R. Krenz-Baath, F. G. Zadegan, and E. Larsson, "Access time minimization in IEEE 1687 networks", in *2015 IEEE International Test Conference (ITC)*, IEEE, Oct. 2015, pp. 1–10. DOI: 10.1109/TEST.2015.7342408.

[122] S. R. Makar and E. J. McCluskey, "On the testing of multiplexers", in *International Test Conference, 1988. Proceedings. New Frontiers in Testing*, IEEE, 1988, pp. 669–679.

[123] F. G. Zadegan, U. Ingelsson, G. Asani, G. Carlsson, and E. Larsson, "Test scheduling in an ieee p1687 environment with resource and power constraints", in *2011 20th IEEE Asian Test Symposium (ATS)*, IEEE, 2011, pp. 525–531.

[124] Y. Blaquiere, Y. Basile-Bellavance, S. Berrima, and Y. Savaria, "Design and validation of a novel reconfigurable and defect tolerant JTAG scan chain", in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2014, pp. 2559–2562.

[125] "IEEE Standard for Test Access Port and Boundary-Scan Architecture", *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013. DOI: 10.1109/IEEESTD.2013.6515989.

[126] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz, "Detectability of Internal Bridging Faults in Scan Chains", in *Asia and South Pacific Design Automation Conference*, Jan. 2009, pp. 678–683. DOI: 10.1109/ASPDAC.2009.4796558.

[127] R. Krenz-Baath, F. G. Zadegan, and E. Larsson, "Access time minimization in IEEE 1687 networks", in *2015 IEEE International Test Conference (ITC)*, Oct. 2015, pp. 1–10.

[128] A. Ibrahim and H. G. Kerkhoff, "Efficient utilization of hierarchical iJTAG networks for interrupts management", in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Sep. 2016, pp. 97–102.

[129] G. Ali, A. Badawy, and H. G. Kerkhoff, "Accessing on-chip temperature health monitors using the IEEE 1687 standard", in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Dec. 2016, pp. 776–779.

[130] F. G. Zadegan, D. Nikolov, and E. Larsson, "On-Chip Fault Monitoring Using Self-Reconfiguring IEEE 1687 Networks", *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 237–251, Feb. 2018.

[131] M. Portolan, "A novel test generation and application flow for functional access to IEEE 1687 instruments", in *2016 21th IEEE European Test Symposium (ETS)*, May 2016, pp. 1–6.

[132] A. Jutman *et al.*, "Effective Scalable IEEE 1687 Instrumentation Network for Fault Management", *IEEE Design Test*, vol. 30, no. 5, pp. 26–35, Oct. 2013.

[133] Y. Cao and W. Zhao, "Predictive Technology Model for Nano-CMOS Design Exploration", in *2006 1st International Conference on Nano-Networks and Workshops*, Sep. 2006, pp. 1–5. DOI: 10.1109/NANONET.2006.346227.

[134] A. Molina and O. Cadenas, "Functional verification: approaches and challenges", 2007.

[135] A. Damljanovic, A. Jutman, G. Squillero, and A. Tsertov, "Post-Silicon Validation of IEEE 1687 Reconfigurable Scan Networks", in *24th IEEE European Test Symposium (ETS) (to be published)*, IEEE, 2019.

[136] E. J. Marinissen, V. Iyengar, and K. Chakrabarty, "A set of benchmarks for modular testing of SOCs", in *Test Conference, 2002. Proceedings. International*, IEEE, 2002, pp. 519–528.

[137] M. K. Reddy and S. M. Reddy, "Detecting FET Stuck-Open Faults in CMOS Latches And Flip-Flops", *IEEE Design & Test of Computers*, vol. 3, no. 5, pp. 17–26, Oct. 1986.

[138] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz, "Detection of Internal Stuck-Open Faults in Scan Chains", in *IEEE International Test Conference*, Oct. 2008, pp. 1–10. DOI: 10.1109/TEST.2008.4700577.

[139] F. G. Zadegan, E. Larsson, A. Jutman, S. Devadze, and R. Krenz-Baath, "Design, Verification, and Application of IEEE 1687", in *Proceedings of the 2014 IEEE 23rd Asian Test Symposium*, ser. ATS '14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 93–100, ISBN: 978-1-4799-6030-9. DOI: 10.1109/ATS.2014.28. [Online]. Available: http://dx.doi.org/10.1109/ATS.2014.28.

[140] M. A. Kochte, R. Baranowski, M. Schaal, and H. Wunderlich, “Test Strategies for Reconfigurable Scan Networks”, in *2016 IEEE 25th Asian Test Symposium(ATS)*, vol. 00, Nov. 2016, pp. 113–118. DOI: 10.1109/ATS.2016.35. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ATS.2016.35.

[141] M. Alam and S. Mahapatra, “A comprehensive model of PMOS NBTI degradation”, *Microelectronics Reliability*, vol. 45, no. 1, pp. 71–81, 2005, ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2004.03.019.

[142] A. Ceratti, T. Copetti, L. Bolzani, and F. Vargas, “Investigating the use of an on-chip sensor to monitor NBTI effect in SRAM”, in *2012 13th Latin American Test Workshop (LATW)*, 2012, pp. 1–6. DOI: 10.1109/LATW.2012.6261238.

[143] A. Ruospo and E. Sanchez, “On the Detection of Always-On Hardware Trojans Supported by a Pre-Silicon Verification Methodology”, in *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, Dec. 2019, pp. 25–30. DOI: 10.1109/MTV48867.2019.00013.

[144] S. Pham, J. Dworak, and T. Manikas, “An Analysis of Differences between Trojans inserted at RTL and at Manufacturing with Implications for their Detectability”, in *2012 IEEE North Atlantic Test Workshop (NATW)*, 2012.

[145] J. Zhang and Q. Xu, “On hardware Trojan design and implementation at register-transfer level”, in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 107–112. DOI: 10.1109/HST.2013.6581574.

[146] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity”, *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943. DOI: 10.1007/BF02478259. [Online]. Available: https://doi.org/10.1007/BF02478259.

[147] S. V. Pham and J. Dworak, “An Analysis of Differences between Trojans inserted at RTL and at Manufacturing with Implications for their Detectability”, 2012.

[148] K. Xiao *et al.*, “Hardware Trojans: Lessons Learned after One Decade of Research”, *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, pp. 1–23, May 2016. DOI: 10.1145/2906147.

[149] R. Elnaggar and K. Chakrabarty, “Machine Learning for Hardware Security: Opportunities and Risks”, *Journal of Electronic Testing*, vol. 34, pp. 1–19, Apr. 2018. DOI: 10.1007/s10836-018-5726-9.

[150] J. Cramer, “The Origins of Logistic Regression”, en, *SSRN Electronic Journal*, 2003, ISSN: 1556-5068. DOI: 10.2139/ssrn.360300. [Online]. Available: http://www.ssrn.com/abstract=360300.

[151] A. Broder, "On the resemblance and containment of documents", in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[152] N. Altman and M. Krzywinski, "Sources of variation", en, *Nature Methods*, vol. 12, no. 1, pp. 5–6, 2015, ISSN: 1548-7091, 1548-7105. DOI: 10.1038/nmeth.3224. [Online]. Available: http://www.nature.com/articles/nmeth.3224.

[153] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, "AN OVERVIEW OF MACHINE LEARNING", en, in *Machine Learning*, Elsevier, 1983, pp. 3–23, ISBN: 978-0-08-051054-5. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780080510545500054.

[154] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance", in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 400–622. DOI: 10.1109/ISSCC.2008.4523226.

[155] T. B. Chan, P. Gupta, A. Kahng, and L. Lai, "DDRO: A Novel Performance Monitoring Methodology Based on Design-Dependent Ring Oscillators", pp. 633–640, May 2012. DOI: 10.1109/ISQED.2012.6187559.

[156] M. Cozzi, J.-M. Galliere, and P. Maurine, "Thermal Scans for Detecting Hardware Trojans", in. Jan. 2018, pp. 117–132, ISBN: 978-3-319-89640-3. DOI: 10.1007/978-3-319-89641-0_7.

[157] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte, and M. Tehranipoor, "Security-Aware FSM Design Flow for Identifying and Mitigating Vulnerabilities to Fault Attacks", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1003–1016, 2019.

[158] M. Rathmair, F. Schupfer, and C. Krieg, "Applied formal methods for hardware Trojan detection", in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 169–172.

[159] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores", in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011, pp. 67–70.

[160] J. Cruz, F. Farahmandi, A. Ahmed, and P. Mishra, "Hardware Trojan Detection Using ATPG and Model Checking", in *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, 2018, pp. 91–96.

[161] S. Yao, X. Chen, J. Zhang, Q. Liu, J. Wang, Q. Xu, Y. Wang, and H. Yang, "FASTrust: Feature analysis for third-party IP trust verification", in *2015 IEEE International Test Conference (ITC)*, 2015, pp. 1–10.

[162] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "VeriTrust: Verification for Hardware Trust", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, 2015.

[163] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis", Nov. 2013, pp. 697–708. DOI: 10.1145/2508859.2516654.

[164] L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, L. D. Jackel, Y. LeCun, U. A. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Comparison of classifier methods: a case study in handwritten digit recognition", in *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5)*, vol. 2, 1994, 77–82 vol.2. DOI: 10.1109/ICPR.1994.576879.

[165] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[166] A. Ruospo and E. Sanchez, "On the Detection of Always-On Hardware Trojans Supported by a Pre-Silicon Verification Methodology", in *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, 2019, pp. 25–30. DOI: 10.1109/MTV48867.2019.00013.

[167] K. Hasegawa, M. Yanagisawa, and N. Togawa, "A hardware Trojan classification method using machine learning at gate-level netlists based on Trojan features", Jul. 2017, pp. 1427–1438.

[168] A. Kulkarni, Y. Pino, and T. Mohsenin, "Adaptive real-time Trojan detection framework through machine learning", in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016.

[169] Y. Liu, Y. Jin, A. Nosratinia, and Y. Makris, "Silicon demonstration of hardware Trojan design and detection in wireless cryptographic ICs", 2017.

[170] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan, "Hardware trojan attacks: Threat analysis and countermeasures", Aug. 2014, pp. 1229–1247.

[171] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection", *Design & Test of Computers, IEEE*, 2010.

[172] A. Ruospo and E. Sanchez, "On the Detection of Always-On Hardware Trojans Supported by a Pre-Silicon Verification Methodology", in *2019 20th International Workshop on Microprocessor SoC Test, Security and Verification (MTV)*, 2019, pp. 25–30.

[173] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep Blue", en, *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, Jan. 2002, ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1.

[174] M. I. Jordan and T. M. Mitchell, "Machine Learning: Trends, Perspectives, and Prospects", en, *Science*, vol. 349, no. 6245, pp. 255–260, Jul. 2015, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aaa8415.

[175] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers", *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959, ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

[176] A. M. Turing, "COMPUTING MACHINERY AND INTELLIGENCE", en, *Mind*, vol. LIX, no. 236, pp. 433–460, 1950, ISSN: 1460-2113, 0026-4423. DOI: 10.1093/mind/LIX.236.433. [Online]. Available: https://academic.oup.com/mind/article/LIX/236/433/986238.

[177] *Icarus verilog*. [Online]. Available: http://iverilog.icarus.com/.

[178] *Ply (python lex-yacc)*. [Online]. Available: http://www.dabeaz.com/ply/.

[179] *Lazygrid*. [Online]. Available: https://pypi.org/project/lazygrid/.

[180] S. Linnainmaa, "Taylor expansion of the accumulated rounding error", en, *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976, ISSN: 1572-9125. DOI: 10.1007/BF01931367. [Online]. Available: https://doi.org/10.1007/BF01931367.

[181] K. von Arnim, C. Pacha, K. Hofmann, T. Schulz, K. Schrufer, and J. Berthold, "An Effective Switching Current Methodology to Predict the Performance of Complex Digital Circuits", in *2007 IEEE International Electron Devices Meeting*, 2007, pp. 483–486. DOI: 10.1109/IEDM.2007.4418979.

[182] G. Sannena and B. P. Das, "Low Overhead Warning Flip-Flop Based on Charge Sharing for Timing Slack Monitoring", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1223–1232, 2018, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2018.2810954.

[183] M. Cozzi, J.-M. Galliere, and P. Maurine, "Thermal Scans for Detecting Hardware Trojans", in. Jan. 2018, pp. 117–132, ISBN: 978-3-319-89640-3. DOI: 10.1007/978-3-319-89641-0_7.

[184] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking", *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

[185] K. Hasegawa, M. Yanagisawa, and N. Togawa, "Trojan-feature extraction at gate-level netlists and its application to hardware-Trojan detection using random forest classifier", in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[186] P. Zhao and Q. Liu, "Density-based Clustering Method for Hardware Trojan Detection Based on Gate-level Structural Features", in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2019, pp. 1–4.

[187] T. Inoue, K. Hasegawa, Y. Kobayashi, M. Yanagisawa, and N. Togawa, “Designing Subspecies of Hardware Trojans and Their Detection Using Neural Network Approach”, in *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 2018, pp. 1–4.

[188] K. Hasegawa, M. Yanagisawa, and N. Togawa, “Trojan-feature extraction at gate-level netlists and its application to hardware-Trojan detection using random forest classifier”, in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.