A Logic Test Chip for Optimal Test and Diagnosis

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Benjamin T. Niewenhuis
B.S., Engineering, Calvin College

Carnegie Mellon University
Pittsburgh, PA
May, 2018

Abstract

The benefits of the continued progress in integrated circuit manufacturing have been numerous, most notably in the explosion of computing power in devices ranging from cell phones to cars. Key to this success has been strategies to identify, manage, and mitigate yield loss. One such strategy is the use of test structures to identify sources of yield loss early in the development of a new manufacturing process. However, the aggressive scaling of feature dimensions, the integration of new materials, and the increase in structural complexity in modern technologies has challenged the capabilities of conventional test structures.

To help address these challenges, a new logic test chip, called the Carnegie Mellon Logic Characterization Vehicle (CM-LCV), has been developed. The CM-LCV utilizes a two-dimensional array of functional unit blocks (FUBs) that each implement an innovative functionality. Properties including fault coverage, logical and physical design features, and fault distinguishability are shown to be composable within the FUB array; that is, they exist regardless of the size and composition of the FUB array. A synthesis flow that leverages this composability to adapt the FUB array to a wide range of test chip design requirements is presented. The connection between the innovative FUB functionality and orthogonal Latin squares is identified and used to analyze the universe of possible FUB functions. Two additional variants to the FUB array are also developed: heterogenous FUB arrays utilize multiple FUB functions to improve the synthesis flow performance, while pipelined FUB arrays incorporate sequential circuit elements (e.g., flip-flops and latches) that are absent from the original combinational FUB array.

In addition to the design of the CM-LCV, methods for testing it are presented. Techniques to create minimal sets of test patterns that exhaustively exercise each FUB within the FUB array are developed. Additional constraints are described for the heterogenous and pipelined FUB arrays that allow these techniques to be applied for both variant FUB arrays. Furthermore, a simple built-in self test (BIST) scheme is described and applied to a reference design, resulting in a 88.0% reduction in the number of test cycles required without loss in fault coverage.

A hierarchical FUB array diagnosis methodology (HFAD) is also presented for the CM-LCV that leverages its unique properties to improve performance for multiple defects. Experiments demonstrate that this HFAD methodology is capable of perfect accuracy in 93.1% of simulations with two injected faults, an improvement on the state-of-the-art commercial diagnosis. Additionally, silicon fail data was collected from a CM-LCV manufactured using a 14nm process by an industry partner. A comparison of the diagnosis results for the 1,375 fail logs examined shows that the HFAD methodology discovers additional defects during multiple defect diagnosis that the commercial tool misses for 40 of the diagnosed fail logs. Examination of these cases shows that the additional defects found by the HFAD methodology can result in improved diagnosis confidence and more precise descriptions of the defect behavior(s).

The contributions of this dissertation can thus be summarized as the description of the design, test, and diagnosis of a new logic test chip for use in yield learning during process development. This CM-LCV can be adapted to meet a wide range of test chip requirements, can be efficiently and rigorously tested, and exhibits properties that can be used to improve diagnosis outcomes. All of these claims are validated through both simulated experiments and silicon data.

Acknowledgements

I would like to thank first and foremost my advisor, Prof. Shawn Blanton, for all of the mentorship and support through the past six years. I am also grateful to the members of my thesis committee: Prof. Andrzej Strojwas, Prof. Lawrence Pileggi, Dr. Enamul Amyeen, and Dr. Rao Desineni, for their assistance and feedback with this work.

Second, I would like to thank my colleagues and friends who have helped me along and brightened my days. In particular I would like to recognize Soumya Mittal and Zeye Liu, who have both been invaluable as both co-authors and friends. I am grateful to the other members of ACTL, both past and present, who have contributed to both my research and my growth as a person (in no particular order): Matthew Beckler, Cheng Xue, Xue Yang, P. K. Nag, Xuanle Ren, Brian Osbun, Shanghang Zhang, Philip Fynan, Xiang Lin, Cuong Nguyen, Jaime Kang, Qicheng Huang, Chenlei Fang, Carl Taylor, Danielle Duvalsaint, Anthony Xin, Hope Kenya, and Balaji Ravikumar. Finally, I would be remiss to not acknowledge my many friends, in particular: Milda Zizyte, Steve Matsumoto, Brian DeCost, Etkin Akkaya, Michael Taylor, and Cas Hutchinson.

Third, I would like to acknowledge my wonderful family, who has surrounded and sustained me with love and support. I am eternally grateful for my parents: my father Jim, who is a source of faith and encouragement, and my mother Loreen, who (among other things) suggested "Chips and Flurbs and Flips" as the working title for this dissertation. I am thankful for my brother Lucas, who remains a constant companion despite the distance between us. And this dissertation is dedicated to my lovely wife, Mary, who is a deep and

abiding source of joy in my life.

Finally, I would like to acknowledge the financial support which has made this work possible. In particular I would like to thank IBM and the Semiconductor Research Corporation for granting me the IBM SRC Robert H. Dennard Fellowship for the past three years. Additionally I would like to recognize the National Science Foundation (grants 1122272, 1121834, 1121868), the Portuguese Science and Technology Foundation, the Semiconductor Research Corporation (grants 1011237 and 1011276), the Northrop Grumman Graduate Student Fellowship, and the Bradford and Diane Smith Graduate Fellowship for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of any manufacturing process is to produce a physical instance, or part, according to a design. The semiconductor manufacturing process is no different in this respect: the design in this case is a complete plan for an integrated circuit. Encompassed in this design are the *design specifications*, which can be either functional (e.g., constraints on the input/output mapping) or performance-based (e.g., the timing of signals, power usage, etc.). Additional distinctions are made in this dissertation between the *physical design*, comprising the layout and the physical patterns to be manufactured, and the *logical design*, comprising the gate-level representation (e.g., the netlist).

Unfortunately no manufacturing process is perfect, resulting in parts that fail to meet the design specifications. The fraction of good parts produced is defined as the *yield* of the manufacturing process, and *test* is the filtering process by which good and bad parts are differentiated. Integral to the success of the semiconductor industry has been the capability to detect, characterize, and mitigate sources of yield loss [1, 2]. These sources of yield loss in integrated circuit manufacturing are numerous, including:

- **Poor design** - The design, when manufactured, may not produce parts that meet the design specifications. For example, poorly characterized device models may lead to a design that cannot meet timing specifications, or a logic error in the design may make

it unable to meet functional specifications.

- **Poor test** - A bad test may improperly classify good parts as bad. For example, testing a false path in a part (i.e., a circuit path that is never used in the functional application of the design) may cause an otherwise good part to be judged as failing to meet the timing specification [3].

- **Damage during test** - The testing process itself may cause damage to a part, causing it to fail to meet the design specifications. An example of this is damage to otherwise good parts that can occur during burn-in test performed at elevated temperatures and voltages [4, 5].

- **Defects** - A defect is a localized anomaly that causes a part to fail to meet the design specifications. Defects can be either *random* or *systematic*. Random defects arise from processes that are randomly distributed across a manufactured part; the classic example of a random defect is a contaminant particle. Systematic defects, in contrast, are defects that are correlated to some feature of the design. Defects are further classified as either *soft* (only produce failures in at specific operating conditions) or *hard* (produce failures in all operating conditions) in nature.

Minimizing yield loss due to defects has been challenging as manufacturing processes have continued to increase in complexity [6, 7]. The objective of this thesis is to contribute to this undertaking by presenting a new logic test chip design with optimal test and diagnosis characteristics for yield learning in new manufacturing technologies. The remainder of this chapter provides context for this objective. Section 1.1 reviews the use of test chips in the development of new manufacturing technologies. Section 1.2 provides background on existing test chips. Sections 1.2.1 and 1.2.2 review the role of digital test and diagnosis in the detection and characterization of defects. Finally, Section 1.3 presents the organization of the remainder of this dissertation.

1.1 Semiconductor Technology Development Cycle

A significant application for test chips is in the development of new technology nodes. This section will present a generic technology development flow for a modern semiconductor process with particular focus on the types and roles of the test chips used at each stage [8].

A new technology node begins with a set of objectives. These objectives can be related to the performance of the process (e.g., processing time, cost, complexity, etc.), the performance of the devices manufactured (e.g., power, performance, area, etc.), or the inclusion of specific features. Figure 1.1 describes the subsequent development process for a generic technology that has been divided into six separate stages, along with the types of test chips manufactured at each stage.

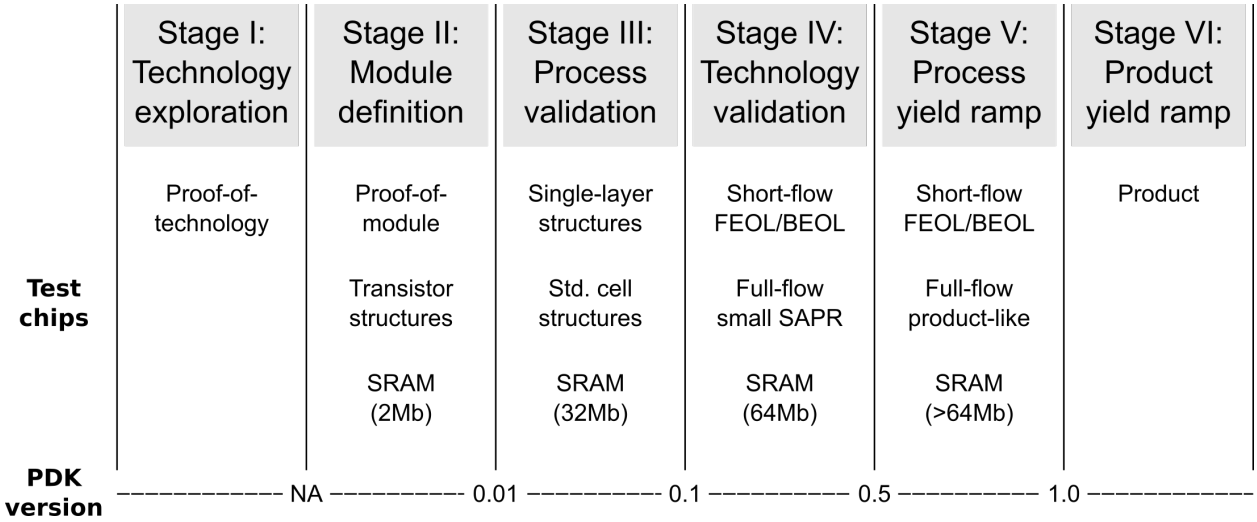


Figure 1.1: Outline of a generic semiconductor technology development process.

Stage I in the generic technology development process of Figure 1.1 is technology exploration, namely the identification of the techniques required to meet the objectives. These techniques may include the use of processing technologies (e.g., double patterning [9]), the creation of structures (e.g., FinFETs [10]), and/or the inclusion of specific materials (e.g., high-K gate dielectrics [11]). At this stage, test chips consist of simple proof-of-concept structures used to evaluate each technique independently.

Once a set of techniques has been identified, Stage II is module definition, which consists

of grouping sequences of processing steps into discrete modules. Electrical measurements and microscopy are used to examine simple test structures created using the individual modules. Measurements collected during this stage include the number of working devices per unit area and physical parameters such as the thickness and roughness of different materials. The first SRAM test structures are also manufactured, as their high transistor density, regular structure, and simple test and diagnosis make them well-suited for estimating overall process defectivity.

After module definition is complete, Stage III is process validation. The processing modules are integrated and used to create single-layer test structures and larger SRAMs. The initial process design kit (PDK) is defined at the beginning of this stage and rapidly refined using the data collected from the test structures. Development of the standard-cell library also begins, and various standard-cell test structures, including individually-probed standard-cell structures and ring oscillators, are manufactured. Defectivity is extremely high at this stage and is driven by systematic process sensitivities as well as random defects. Information from the test structures and SRAMs is used to reduce this defectivity through adjustment of the various design and process parameters.

After process validation is Stage IV, technology validation. Test structure complexity increases, with the deployment of short-flow test chips, which are manufactured using only a subset of the process layers, as well as medium-sized SRAMs (32-64Mb). These short-flow test chips are targeted towards either the front end of the line (FEOL) (i.e., transistor and contact layers) or back end of the line (BEOL) (i.e., metal interconnect). Additionally, small full-flow standard automated place-and-route (SAPR) logic test chips are manufactured using the developing PDK and are used to further refine it as the process begins to stabilize. Defectivity remains high at this stage, and significant effort is put into driving it down using the results derived from the test structures.

After process validation, Stage V is process yield learning. In addition to short-flow test chips and larger SRAMs, full-flow product-representative test structures are manufactured,

including large SAPR logic test chips. These product-representative test chips are used not only for continued yield learning, but also to demonstrate the capabilities of the new process. The final refinements are made to the PDK and the standard-cell library before release to the product designers. Leveraging test structure results to reduce process defectivity is crucial at this stage, as it can become significantly more difficult to make PDK-relevant process changes after the PDK has been released.

The final Stage VI in the technology development process is product yield learning. The PDK is released and used to develop product designs, and product manufacturing begins at reduced volumes. Initial product yield is expected to be low even with all of the previous emphasis on identifying and reducing process defectivity; it is crucial for the commercial viability of the technology that product yield losses be identified and corrected at this stage before concluding the technology development process and beginning full volume product manufacturing [2].

Two examples from industry underscore the importance of the full-flow logic test chips manufactured during process validation and process yield learning. First, in one instance a specific standard cell was never incorporated into a larger logic structure in any test silicon during the technology development process. When product designs utilizing this standard cell were manufactured, the cell failed at a high rate, resulting in significant yield loss [12]. Second, an industry member has indicated that, in modern technology nodes, the short-flow test structures have been observed to fail to detect some defects that they would otherwise be expected to detect (e.g., shorts in a short-flow test structure designed to detect shorts), indicating an increasing need for high-quality full-flow test chips [13].

1.2 Test Structures

A *test chip* (or *test structure*) is a design (or component of a design) that is manufactured for purposes other than use or sale. Common test chip applications include characterization

of process parameters, yield loss exploration, and estimation of process reliability. This dissertation focuses on test chips targeted towards yield loss. There exist many types of such test structures, ranging from passive structures (e.g., via arrays, comb drives, etc.) to large, complex circuits (e.g., ring oscillators, SRAM, etc.) [14]. The dissertation work here focuses on the SAPR logic test chips that are developed in the mid to late stages of the technology development process, and which are intended to catch sources of yield loss that impact the random logic used in product designs.

It is necessary to examine the types of yield loss mechanisms that are targeted by SAPR logic test chips. While random defects can cause yield losses in digital logic, random defectivity is largely under control in an ideal process development cycle by the time SAPR logic test chips are introduced, and other test structures (e.g., SRAMs, comb drives, serpentine paths, etc.) are often better suited to characterizing random defects. Instead, relevant yield losses in digital logic are largely systematic in nature, and are driven by the abundance of new physical patterns created during the automated place-and-route process (e.g., standard cell abutments, routing paths in metal interconnect, etc.). Each of these new physical patterns can potentially interact with one or more steps of the manufacturing process, resulting in systematic phenomena including:

- *Parametric effects* - neighboring physical patterns can impact the electrical performance of both devices and metal interconnect [15, 16]. If these effects are not modeled and accounted for in designs, they can cause significant parametric yield losses.

- *Systematic defects* - interactions between physical patterns and the manufacturing process can cause manufacturing defects (e.g., shorts, opens, partial voids in vias, etc.), which in turn can result in either hard or soft failures.

Another important aspect of these systematic phenomena is their potential multiplicity: if some physical pattern in a design is susceptible, every occurrence of that physical pattern can be impacted if the appropriate conditions arise. Thus, multiple defect test and diagnosis

has increased value for SAPR logic test chips, particularly in the earlier stages of the process development cycle when these systematic phenomena are expected to be more prevalent and severe.

Because of the constraints of digital logic test and diagnosis (discussed in Sections 1.2.1-1.2.2), the SAPR logic test chip is best suited for detection of hard systematic defects, though it may still be possible able to detect soft systematic defects and parametric failures in some cases. Despite these limitations, effective SAPR logic test chips are necessary not only for identifying systematic sources of yield loss, but also for evaluating the effects of different digital logic styles, including different place-and-route tool configurations and design for manufacturability (DFM) guidelines.

The most straightforward means of implementing a logic test chip is to use a product design, either in whole or in part. This ensures that the defects observed in the logic test chip are product-relevant (at least to the parts of the product included). However, this approach is not without its disadvantages. High quality testing of product designs incurs significant test cost [17] and is frustrated by the existence of redundancies [18, 19]. Another challenge is the difficulty in diagnosing defects in product designs, which can lead to poor diagnostic resolution and accuracy [20]. Missed or improperly characterized defects resulting from these issues can result in significant costs, including the need for more silicon dedicated to logic test chips and other test structures, more time and effort spent on yield learning, and ultimately the possibility of lower product yield.

Two additional works merit discussion in this section. In [21], the authors describe a test structure that consists of a combinational logic circuit and a scan-based "jig", which is surrounding circuitry that enables scan and performance-based testing of the combinational logic circuit. Performance testing is accomplished through a configurable ring oscillator mode. Specifically, a ring oscillator is established under the control of the jig by making either an inverting or non-inverting connection from a circuit output back to one of the inputs. While this jig approach is highly configurable, the ability of this test structure to

detect defects is dominated by the characteristics of the logic circuit used. Because this logic circuit is often a subcircuit drawn from product designs it suffers from the same test and diagnosis drawbacks of product designs mentioned previously.

In [22], the authors describe a test structure consisting of an array of flip-flops with SRAM-like connections (bit lines, word lines, etc.). By utilizing a memory-like I/O scheme, the authors apply memory test techniques to their design. Additionally, they argue that the flip-flop array can be made to reflect a product layout through synthesis with a standard-cell library, and the use of conventional place-and-route flows. Finally, product-like routing is mimicked by having multiple paths for the bit- and word-lines that are individually selected using multiplexers. While this approach is innovative in leveraging memory test techniques in a logic test chip, there are two primary drawbacks. First, it is difficult to use all of the cells in a standard-cell library in a flip-flop array without large numbers of untestable faults. Second, it remains an open question as to whether the proposed physical similarities between this test chip and product design layouts are sufficient to ensure that the same defect mechanisms will be observed in both.

1.2.1 Digital Integrated Circuit Test

Digital integrated circuit test consists of the application of input vectors to a manufactured part and the observation of its response. One of the fundamental problems in test is the selection of test patterns; perfect test would require that every valid input pattern be applied to each manufactured part. However, the number of patterns required to do this for even small designs quickly renders this approach impractical. Instead, sources of yield loss, including defects, are typically modeled at the structural level as *faults*. These fault models are then used to generate and grade test patterns.

However, the relationship between defects and fault models is not always straightforward. Because test is primarily concerned with *detection*, a fault model does not need to closely match the behavior of defects in a design in order for it to be effective. So long as the tests

produced using a fault model exercise the design in a manner that detects all of the relevant defects, any differences between the fault model and the defect behavior is irrelevant.

Despite this disconnect, fault models have historically matched fairly well with the defects observed in the semiconductor manufacturing process; in particular, the single stuck line (SSL) fault model [1], which assumes that any logic signal in the logic circuit representation can be shorted to either power or ground, has been used with great success. However, as the semiconductor manufacturing process has evolved, the defects observed have behaved less and less like SSL faults. In 1990, a paper examining a test chip fabricated using a $1.5\mu m$ process found that 79.2% of the detected defects exhibit behavior that matches an SSL fault [23]. A similar paper in 2008, working with results from a $90nm$ process, found that only 20% of the detected defects exhibit behavior that matches a SSL fault [24], and this trend towards increased defect complexity has continued in advanced manufacturing processes [25].

The evolving nature of defects has led to the development of a myriad of new fault models. A few examples include the stuck-open [26], path delay [27], transition [28], input pattern [29], and cell-aware [30] fault models. These fault models better reflect the evolving defect behavior, resulting in higher quality test sets with improved defect detection. However, they also impose additional costs. More complicated fault models impose additional constraints during automatic test pattern generation (ATPG), leading to increased difficulty and run-time. They also may require additional analysis of the design (e.g., identification of relevant circuit paths, enumeration of realistic bridges, etc.). They also result in larger test sets and may increase the complexity of the test application process, resulting in higher test execution costs.

In addition to new fault models, test metrics have also been proposed to help cope with the increased defect complexity. Test metrics are not meant to model defect behavior, but instead are a measure of the completeness of a test set, with a more complete test set resulting in better detection of defects (and other sources of yield loss). Test metrics can

be used much like fault models to grade and generate test sets. Examples of test metrics include N-detect [31], the gate exhaustive metric [32,33], and K longest paths per gate [34].

The logic test chips used for yield learning in process development present a unique test challenge. While they are typically small and simple (compared to typical product designs), the prevalence of multiple defects and complex defect behaviors, along with the importance of diagnosis accuracy during yield learning, result in a need for rigorous test. In this context, efficient test techniques allow for more complex fault models (resulting in improved probability of detecting complex defects) and more test data for diagnosis (resulting in improved diagnosis accuracy).

1.2.2 Digital Integrated Circuit Diagnosis

Diagnosis of digital integrated circuits involves the use of failing responses to identify the location and behavior of any defects present. Traditional fault diagnosis operates at the logic level, and attempts to match the observed defect behavior to one or more fault model instances (i.e., faults), where each fault and its corresponding location is typically referred to as a candidate. Diagnosis performance is typically considered in terms of two orthogonal metrics: *accuracy*, a measure of how well the reported candidate(s) correspond to the actual defect, and *resolution*, the number of candidates reported. There are two general approaches to fault diagnosis:

- *Effect-cause* approaches analyze the design and the observed failing response to narrow down the possible candidate faults, which are then simulated to derive the diagnosis candidates [35–37].

- *Cause-effect* approaches pre-compute the fault responses for a given test set and collection of faults, resulting in a fault dictionary. The failing response is then compared to each of the simulated fault responses, and the faults corresponding to the best matches are reported as the diagnosis candidates [38–40].

In general, cause-effect approaches (i.e., fault dictionaries) are expensive to compute and store, but can be optimized for quick run-time on a per-diagnosis basis. Effect-cause approaches, in contrast, have no up-front costs but require the design to be analyzed in each diagnosis run. Additionally, cause-effect approaches are more flexible, better able to handle changes in the test set (for which fault dictionaries must be recomputed), and typically produce candidates with increased accuracy and better resolution when defect behavior is more complex (e.g., multiple defects, fault models poorly capture defect behavior, etc.).

Candidates obtained from fault diagnosis represent a valuable source of information for defects. This information can be combined with the physical design and correlated across many failing parts (i.e., volume diagnosis) [41–43] and/or used to guide physical failure analysis (PFA), during which the defective parts are physically inspected using a variety of techniques to confirm the presence and physical properties of any defects. Issues with diagnostic accuracy and resolution both adversely affect PFA: if the diagnosis is not accurate, PFA may not find any physical defects in the implicated regions of the defective part. If the diagnostic resolution is poor, there may be too large of an area/volume in the defective part to inspect using PFA techniques, which can be destructive in nature.

For logic test chips used during process development the volumes of test chips manufactured may not allow for the application of volume diagnosis techniques, particularly if multiple process recipes are being explored. This means that yield learning during process development is heavily reliant on PFA results. Additionally, the potential for multiple defects and new defect behaviors make cause-effect diagnosis approaches less useful. Instead, effect-cause diagnosis techniques are used to obtain the best diagnosis result possible based on the observed tester response, and improvements in diagnostic accuracy and resolution, particularly for multiple defects and complex defect behaviors, are highly desirable.

1.3 Summary

The ability to detect, isolate, and characterize defects in the semiconductor manufacturing process has been crucial to its historical success. Test chips represent a key tool in achieving this capability. However, this capability is becoming more difficult to ensure as the process complexity has continued to increase with the development of new technology nodes. This dissertation presents a new SAPR test chip, the Carnegie Mellon Logic Characterization Vehicle (CM-LCV), as one means of securing this capability.

The remainder of this dissertation is organized as follows. The design of the CM-LCV is described in Chapter 2. Chapter 3 discusses the test properties of the CM-LCV, including an efficient built-in self test (BIST) architecture. Chapter 4 presents a custom diagnosis methodology for the CM-LCV that enables accurate, multiple-defect diagnosis. Finally, Chapter 5 summarizes the contributions of this dissertation and discusses promising directions for future work.

Chapter 2

CM-LCV Design

This chapter describes the Carnegie Mellon Logic Characterization Vehicle (CM-LCV). Section 2.1 describes the design architecture of the CM-LCV. The composability of various properties within the FUB array used in the CM-LCV is demonstrated in Section 2.2. A general synthesis flow for the CM-LCV that leverages this composability is presented in Section 2.3. Further discussion on the mathematical nature of the functions used in the CM-LCV is included in Section 2.4. Additional FUB array variants are developed in Section 2.5 to address several shortcomings of the original CM-LCV design. Finally, Section 2.6 summarizes the work presented in this chapter.

2.1 Design Architecture

At an abstract level, the physical design of an IC (i.e., the layout) can be viewed as a collection of sites for potential manufacturing defects. Different defect mechanisms stemming from the manufacturing process can potentially affect different subsets of these sites; for example, random conducting particles with radius r have the potential to cause shorts in locations with weighted critical areas of radius r. At each site within the physical design, a defect can be in one of three observability categories: undetected, testable but not diagnosable, or both testable and diagnosable.

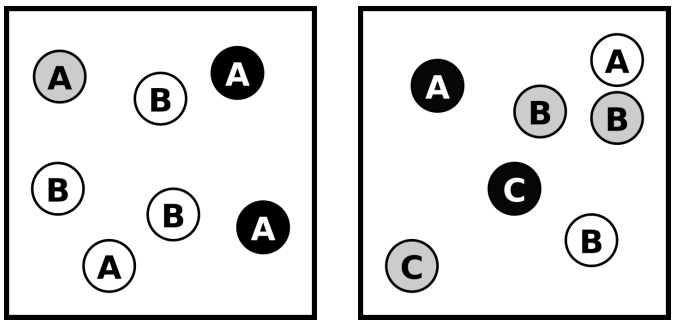


Figure 2.1: Distribution of potential defect sites within the physical designs for (a) a test chip and (b) a product. Each defect site is labeled according to the defect mechanism that affects it, and colored according to the degree of observability, with no shading representing untestable sites, light shading for testable but not diagnosable sites, and dark shading for sites that are both testable and diagnosable.

Figure 2.1 shows this abstract view for the physical designs of a test chip (Figure 2.1a) and a product (Figure 2.1b). Potential defect sites are represented as circles and are labeled according to the defect mechanisms that can affect them. The observability of each defect site is indicated by its shading, with no shading indicating untestable, light shading indicating testable but not diagnosable, and dark shading indicating both testable and diagnosable.

First, suppose some defect mechanism affects the sites labeled "A" in Figure 2.1. In this case, the test chip (Figure 2.1a) has some sites that are both testable and diagnosable, and some that are not. If this defect mechanism affects a random site in the test chip, there is a chance that it will not be tested (untestable sites), or be tested but impossible to localize to the appropriate site (undiagnosable sites). Both of these cases represent a missed opportunity to characterize this defect mechanism.

Next suppose some defect mechanism affects the sites labeled "B" in Figure 2.1. In this case the test chip of Figure 2.1a has some such sites, but none are testable, let alone diagnosable. The result is that the lack of testability and diagnosability has created a *defect gap* between the two designs; the test chip of Figure 2.1a can never be used to observe this defect mechanism which has the potential to impact the product design of Figure 2.1b.

Third, suppose some defect mechanism affects the sites labeled "C" in Figure 2.1. In this case the test chip of Figure 2.1a has no such sites, while the product design of Figure

2.1b includes several. The result is another defect gap, though the cause in this case is a lack of *physical relevance* between the two designs; that is, the physical design of the test chip contains no sites that are susceptible to a defect mechanism that can affect sites in the physical design of the product.

The hypothetical designs of Figure 2.1 are instructive on the relationship between test structures and product designs. Recall that the one of the objectives for test structures is to identify and characterize defects before they cause product yield loss. In the first case discussed (sites labeled "A"), a test chip design with the properties of Figure 2.1a is adequate but not ideal for a corresponding product design Figure 2.1b; a significant volume of parts may need to be manufactured before the defect mechanism affects one of the sites in the test chip design that can be properly tested, diagnosed, and PFA'ed for yield learning (depending on the rarity of the defect mechanism and the ratio of testable and detectable sites). The defect gap present in the other two cases (sites "B" and "C"), on the other hand, represent a fundamental inadequacy: the test chip design can never be used to observe the corresponding defect mechanisms, which will result in product yield loss. This defect gap can be caused by either a lack in observability at the affected sites (sites "B") or a lack of physical relevance between the two designs (sites "C"). Note that a defect gap can also exist in the other direction, where a test chip design is susceptible to a defect mechanism that cannot impact the product design. This is also problematic, in that it may lead to wasted engineering effort spent to correct a defect mechanism that would not result in product yield loss. In the worst case, the corrections to the manufacturing process may even introduce new defect mechanisms that could potentially affect the product design (essentially an optimization of the manufacturing process for the test chip design at the detriment of the product design).

Based on this analysis, the principles of physical relevance, testability, and diagnosability are key to a successful test chip design. Perfect physical relevance between the test chip design and the product design(s) to be manufactured implies that the defect mechanisms observed in the test chip design would cause product yield loss, and that all such defect

mechanisms affect the test chip design. Perfect testability and diagnosability ensures that all defect mechanisms that affect the test chip design will be testable and diagnosable, and potentially reduces the volume of test chips required for yield learning.

Given this context and with the goal of developing an improved logic test chip, this work assumes that defect mechanisms in the integrated circuit manufacturing process are sensitive to physical features that are only loosely correlated with the logical functionality of a design. This assumption is supported by the language used in the literature to describe systematic manufacturing defects; for example, patterns in metal interconnect [44] or specific standard cells [45]. In no case is the higher-level functionality suggested to be vital to the nature of the defect; at best, the design functionality is included to give some context to the work. Indeed, this separation is fundamental to the concept of a single foundry manufacturing a variety of different product designs.

Thus, given the negligible impact the logical functionality has on the susceptibility of a design to defect mechanisms, the following logical design decisions are made to maximize testability and diagnosability:

- **Regularity** - A set of functional unit blocks (FUBs) with regular connections can be designed to be C-testable [46], a property which ensures optimal test set size and fault coverage over the entire array, regardless of its size.

- **Two-dimensions** - Arranging a collection of FUBs in a two-dimensional unilateral array with vertical and horizontal connections enables error propagation in two directions, allowing for better localization of defects.

- **VH-bijectivity** - The FUB function is required to be bijective (e.g., a one-to-one mapping) and constrained such that any value change on exactly one of the sets of inputs (vertical or horizontal) must result in a change in the value on both sets of outputs. Requiring that the FUB function exhibit this property guarantees both vertical and horizontal propagation of errors through the defect-free FUBs.

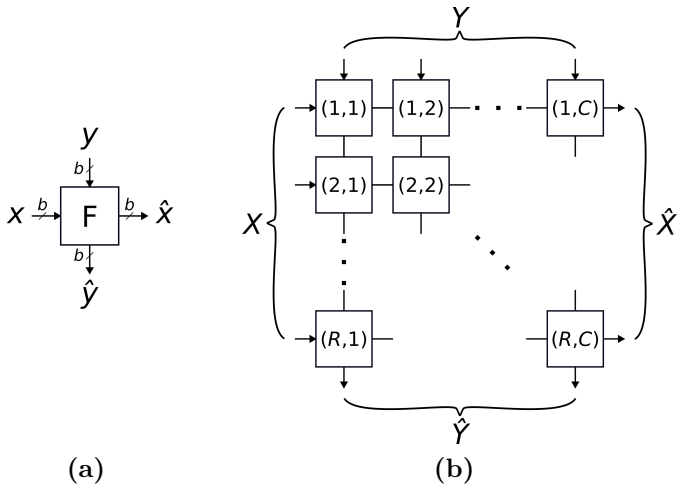


Figure 2.2: Notation for (a) the FUB and (b) the two-dimensional FUB array.

The result of these decisions is the *FUB array*: a collection of functional unit blocks, each of which implement a VH-bijective function, arranged in a two-dimensional array with unilateral connections along both the horizontal and vertical directions. Figure 2.2 shows the notation for the FUB (Figure 2.2a) and FUB array (Figure 2.2b) that will be used throughout this dissertation. The FUB function $F(x, y) \rightarrow (\hat{x}, \hat{y})$ implements a mapping between two input ports x, y and two output ports $\hat{x}, \hat{y}$, with each port having bit width b. Using this notation, VH-bijectivity can be formally stated as follows:

Definition 2.1. Let $F(x, y) \rightarrow (\hat{x}, \hat{y})$ be a bijective function with $x, y, \hat{x}, \hat{y} \in \{0, 1, \ldots, 2^b - 1\}$, and furthermore let $F(x_i, y_i) = (\hat{x_i}, \hat{y_i})$ and $F(x_j, y_j) = (\hat{x_j}, \hat{y_j})$. Function F is VH-bijective if and only if, for all $x_i = x_j$ and $y_i \neq y_j$: $\hat{x_i} \neq \hat{x_j}$ and $\hat{y_i} \neq \hat{y_j}$; and for all $x_i \neq x_j$ and $y_i = y_j$: $\hat{x_i} \neq \hat{x_j}$ and $\hat{y_i} \neq \hat{y_j}$.

VH-bijective FUBs are connected to form a FUB array with R rows and C columns, as shown in Figure 2.2b. A $(row, column)$ indexing notation is used to indicate specific locations within the FUB array, with $(1, 1)$ in the upper left corner and (R, C) in the lower right corner of the array. The FUB input and output ports are connected such that $\hat{x}_{(r,c)} = x_{(r,c+1)}$ and $\hat{y}_{(r,c)} = y_{(r+1,c)}$. The array inputs are collectively represented as $X = (x_{(1,1)}, x_{(2,1)}, \ldots, x_{(R,1)})$ and $Y = (y_{(1,1)}, y_{(1,2)}, \ldots, y_{(1,C)})$. Similarly the array outputs

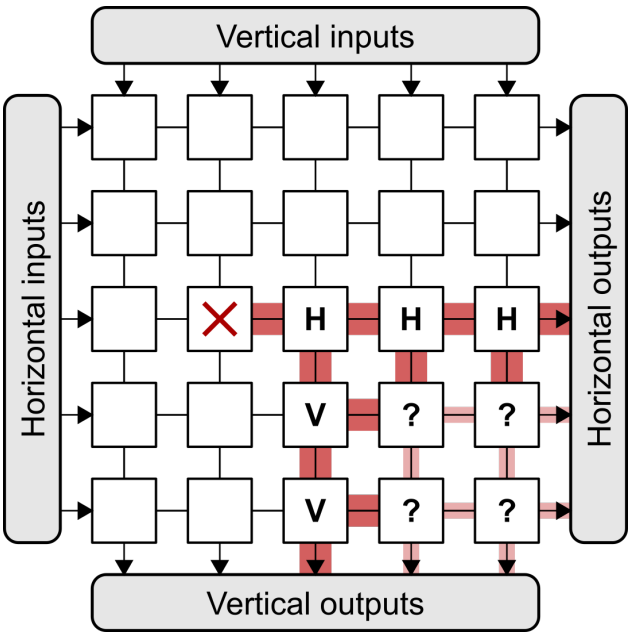


Figure 2.3: VH-bijectivity demonstrated in the presence of a defective FUB in a two-dimensional array. Signal propagation moves left to right, top to bottom. Guaranteed error propagation is represented by the connections with thick red highlighting, while unknown error propagation is represented by the connections with thin red highlighting.

are represented as $\hat{X} = (\hat{x}_{(1,C)}, \hat{x}_{(2,C)}, \ldots, \hat{x}_{(R,C)})$ and $\hat{Y} = (\hat{y}_{(R,1)}, \hat{y}_{(R,2)}, \ldots, \hat{y}_{(R,C)})$. Additionally, the FUB array is by convention assumed to be homogenous (all FUBs implement the same VH-bijective function) and entirely combinational (no sequential circuit elements); variants on the FUB array that are not homogenous or combinational will always be indicated as such when discussed in this dissertation.

The FUB array and its error propagation behavior for a single defective FUB is shown in Figure 2.3. Inputs to the array are applied on the top and left boundaries, and values propagate *downstream* (i.e., down and to the right) through each of the FUBs to the array outputs on the right and bottom boundaries. Assume the FUB instance in the array of Figure 2.3 marked with an "×" is defective, and produces an erroneous value (represented by the dark shading) for some test pattern at its horizontal output. All FUBs in the same row as the defective FUB (marked with an "H") fall under the VH-Bijective constraint: a difference in value present only at a horizontal input will propagate to both outputs. Furthermore, all FUBs in the first downstream column to the right of the defective FUB

(marked with a "V") also fall under the VH-bijective constraint: a difference in value that appears only at a vertical input will propagate to both outputs. The remaining downstream FUBs (marked with a "?") may experience errors on both inputs; because they implement a bijective function the error will propagate to at least one of the outputs, but this case is not covered by VH-bijectivity and thus the error propagation cannot be predicted (represented by the light shading) without exact knowledge of the test pattern and error values.

The CM-LCV is thus formally defined as a collection of one or more FUB arrays, along with the necessary mechanisms for testing (i.e., BIST, scan, etc.). It can be manufactured as a stand-alone test chip or as a macro within a larger design. Regardless of its architecture, it is the FUB array that determines the majority of the properties of interest in this dissertation.

2.2 Composability

A key benefit of the FUB array used in the CM-LCV is its *composability*: many properties exhibited by an individual FUB continue to exist when it is integrated into a FUB array. This means that FUBs can be independently selected for a given property, and the FUB array composed of these FUBs will also exhibit this property. The remainder of this section will describe various properties that are composable within the FUB array.

2.2.1 Design Features

First is the composability of logical design features. It is obvious that logical characteristics of a given FUB implementation, for example the number and type of standard cells, the gate fanin and fanout[1], and even the logical paths through the FUB, are independent of the array size and the inclusion of other, different FUB implementations in the FUB array.

Additionally, many physical design features are composable in the FUB array, depending on the techniques used to create the physical design. For example, in a case where a FUB

[1]Except for the fanin and fanout characteristics of the gates at the FUB primary inputs and outputs, which are indeed influenced by neighboring FUB designs.

array is assembled as a gate-level netlist and the physical design is created using a commercial place-and-route flow, the physical features within the standard cells used in a single FUB (e.g., the arrangement of transistors, contacts, and intra-cell interconnect) are independent of the array size and inclusion of other FUB implementations. More restrictive techniques can ensure additional physical design features are composable; for example, FUBs can be custom designed to fit a specific physical footprint (e.g., specified layout area with standardized connections) and then assembled to create the physical design. In this case any physical features defined to exist with the bounds of an individual FUB are independent of the composition of the remainder of the array. Intermediate techniques also exist; an example, described in [47], is to define small layout snippets of interconnect and standard cells as macro cells within the FUBs, which are then used as-is within a commercial place-and-route flow.

2.2.2 Fault Coverage

Fault coverage is also a property composable in the FUB array for certain types of fault models. In a typical logic design this would be extraordinary; it would imply that the fault coverage for a given sub-module is entirely independent of the rest of the design. This property can be proven for the FUB array for all faults detectable by a single test pattern, as shown in Theorem 2.1.

Theorem 2.1. *All faults within a FUB that are detectable by a single test pattern are detectable when it is incorporated into a VH-bijective FUB array, regardless of its location* (i, j) *with the array and the size parameters* (R, C) *of the array.*

Proof. The sufficiency of Theorem 2.1 can be demonstrated as follows. Recall that detection of a fault is contingent on both activation of the fault and propagation of at least one resulting error. Detection of faults in a FUB will be preserved in the FUB array if (a) all of the required test patterns can be applied to the FUB, and (b) at least one resulting error produced at the FUB outputs is propagated to the array outputs.

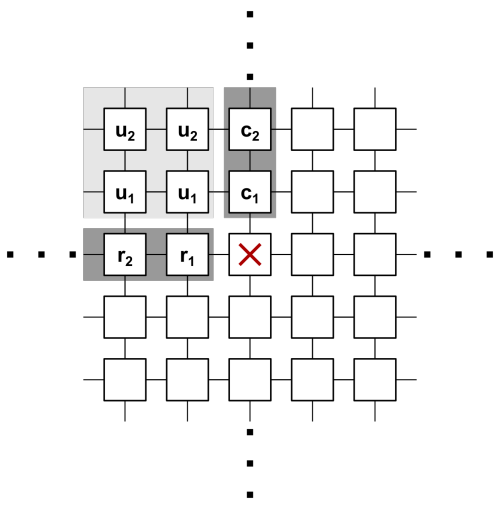


Figure 2.4: Representation of a segment of FUB array. The location of the FUB of interest (marked "×") defines three regions within the array: the FUBs in the same row (marked "r_k"), the FUBs in the same column (marked "c_m"), and the remaining upstream FUBs (marked "u_n").

Figure 2.4 is instructive in establishing the first proposition, namely that the inputs of the target module (shown marked with a "×") can be controlled to any arbitrary input pattern. First consider the vertical input to the target FUB that is controlled by the outputs of c_1: bijectivity guarantees that there is at least one input value that can be applied to the vertical neighbor c_1 to produce the desired value. Any one of these input values can be selected arbitrarily, and this process can be repeated for every FUB in the column (c_2, c_3, etc. in Figure 2.4), leading eventually to the vertical array inputs. The same holds true for the horizontal input value along the FUBs in the same row (i.e., r_1, r_2, etc. in Figure 2.4).

At this point all of the input and output values for the FUBs along the row and column (marked r_k and c_m, respectively, in Figure 2.4) have been specified. Because all of the FUB functions used in this array are bijective, for every possible FUB output value, there must exist exactly one FUB input value that will produce that output value. Consider the first upstream FUB u_1: both its vertical and horizontal output values have been specified (by FUBs r_1 and c_1, respectively), and thus by the bijectivity of the u_1 FUB function there is exactly one FUB input value that must be applied to the inputs of u_1. This process can be repeated for each FUB u_n in the upstream region (u_2, u_3, u_4, etc.) all the way to the array inputs. The result is an array input pattern that will produce the desired values at

the inputs of the FUB of interest.

Sufficient error propagation is also easily demonstrable. A detected fault must produce an error on at least one output of the target FUB. Any FUB that receives an error must propagate it to at least one of its outputs due to bijectivity, and the path of these errors must eventually arrive at one or more of the array outputs.

Therefore both fault activation and error propagation can be achieved for any fault detectable by a single input pattern within a given FUB in the FUB array. Thus the fault coverage for this class of faults is preserved within the array.

□

While this result is powerful, it is still limited to faults detectable with a single test pattern (e.g., SSL, bridge faults, input pattern, etc.). Any fault model that requires multiple test patterns or imposes timing requirements for detection (e.g., transition, path delay, etc.) is not covered by this proof. Furthermore, the efficiency of the array test patterns is not addressed; preserving fault coverage for *every* FUB in the array using the methods presented may require an increasing number of array test patterns as the number of FUBs in the array increases. These considerations are addressed in Section 3.1; at this point it is sufficient to prove that fault coverage in a standalone FUB is maintained given these limitations.

2.2.3 Fault Distinguishability

Finally, fault distinguishability is partially composable within the FUB array. Two faults are distinguishable if they produce distinct responses for an applied test set. Composability in this case implies that a fault that produces a unique response in a standalone FUB will continue to produce a unique response when incorporated into a FUB array. First, however, a definition of adjacency in the FUB array is required:

Definition 2.2. Two FUBs F_a, F_b at array locations $(r_a, c_a), (r_b, c_b)$, respectively, are defined to be adjacent in the FUB array if they share either a horizontal port connection or a vertical

port connection; that is, either $r_a = r_b$ and $|c_a - c_b| = 1$ or $|r_a - r_b| = 1$ and $c_a = c_b$.

Given this definition of adjacency, fault distinguishability can be readily proven for faults in non-adjacent FUBs.

Theorem 2.2. *Any pair of detected faults f_a, f_b in corresponding non-adjacent FUBs F_a, F_b in a VH-bijective FUB array will produce distinguishable array responses.*

Proof. The sufficiency of Theorem 2.2 can be demonstrated as follows. Recalling the error propagation paths observed in Figure 2.3, there exist three possible propagation cases within the FUB array for a given fault:

1. If the fault produces errors on both horizontal and vertical outputs of the FUB, the errors are guaranteed to propagate and be observed at the array outputs corresponding to both the row and column of the faulty FUB: (r, c).

2. If the fault produces errors at only the horizontal outputs of the FUB, the errors are guaranteed to propagate and be observed at the array outputs corresponding to the row and the downstream adjacent column of the faulty FUB: $(r, c + 1)$.

3. If the fault produces errors at only the vertical outputs of the FUB, the errors are guaranteed to propagate and be observed at the array outputs corresponding to the column and the downstream adjacent row of the faulty FUB: $(r + 1, c)$.

Let FUB F_a exist at array location (r_a, c_a). Detected fault f_a in FUB F_a will therefore always produce errors for some test pattern at the array outputs corresponding to the rows and columns: $(r_a, c_a), (r_a + 1, c_a), (r_a, c_a + 1)$. Similarly, if FUB F_b exists at array location (r_b, c_b), detected fault f_b will always produce errors for some test pattern at the array outputs for the rows and columns: $(r_b, c_b), (r_b + 1, c_b), (r_b, c_b + 1)$.

Now, assume that faults f_a, f_b cannot be distinguished. The guaranteed error propagation detailed above implies that, for f_a and f_b to consistently produce errors at the same array outputs, one of the following must be true:

1. $(r_a, c_a) = (r_b, c_b)$
2. $(r_a, c_a) = (r_b + 1, c_b)$
3. $(r_a, c_a) = (r_b, c_b + 1)$
4. $(r_a + 1, c_a) = (r_b, c_b)$
5. $(r_a + 1, c_a) = (r_b + 1, c_b)$
6. $(r_a + 1, c_a) = (r_b, c_b + 1)$
7. $(r_a, c_a + 1) = (r_b, c_b)$
8. $(r_a, c_a + 1) = (r_b + 1, c_b)$
9. $(r_a, c_a + 1) = (r_b, c_b + 1)$

Seven of these nine cases violate the assumption that F_a and F_b are non-adjacent in the array; only $(r_a, c_a + 1) = (r_b + 1, c_b)$ and $(r_a + 1, c_a) = (r_b, c_b + 1)$ result in non-adjacent locations for F_a, F_b. In both of these cases FUBs F_a, F_b are adjacent to and in an upstream position relative to the same FUB F_c, and faults f_a, f_b must emit errors only along the outputs that connect to FUB F_c. Because f_a is detected, it must emit an error for some test t that is applied to one of the input ports of F_c, and results in F_c outputting some values $(\hat{x_a}, \hat{y_a})$. For that same test t, it is not possible for fault f_b to emit any value such that F_c produces the same values $(\hat{x_a}, \hat{y_a})$. This is due to the fact that the bijectivity of F_c implies that fault f_b would have to apply exactly the same value to F_c; however, fault f_b in FUB F_b cannot produce the same values at the inputs of F_c as FUBs F_a and F_b do not control the same input ports of F_c. The result is that F_c will produce different values in the FUB array for test t, and those differences are guaranteed to propagate through the remainder of the FUB array to the array outputs. Thus, f_a and f_b will produce distinguishable array outputs even in these last two cases, contradicting the original assumption and completing the proof. □

Note that there will always be some indistinguishable faults between neighboring FUBs; in the trivial case, any single stuck line fault at the output of one FUB is equivalent to (and thus indistinguishable from) the corresponding single stuck line fault on the input of its immediate neighbor. The existence of additional indistinguishable faults is not guaranteed; in [48], analysis found that only 2.6% of approximately eight thousand 6-bit FUBs had any gate-level input-output faults that met the necessary conditions for indistinguishability. Even in these cases, however, indistinguishable faults only exist if both neighboring FUBs have faults that meet the necessary conditions, and the responses of these faults are indistinguishable for the applied array test set. Based on this observation, also included in [48] are techniques to minimize the number of indistinguishable faults by filtering and controlling the placement of FUBs in the FUB array.

2.3 Synthesis

This section describes the synthesis of the CM-LCV, that is, the process by which the general CM-LCV design architecture (as described in Section 2.1) is translated into a logical and physical design. This synthesis flow for the CM-LCV leverages the composability properties of the FUB array, and is depicted in Figure 2.5. First, a large number of FUB implementations (e.g., gate-level netlists) are created for a VH-bijective FUB function. These implementations can be created using a number of techniques, ranging from manual design to commercial EDA tools. These FUB netlists are then graded according to the target criteria (e.g., standard cell composition, fault coverage, fault diagnosability, etc.) and compiled to create what is termed the FUB library. These same target criteria are then used to express a set of constraints for the FUB array; an example would be to include every standard cell at least twice while maximizing the fault coverage. The creation of the FUB array is formulated as an optimization problem, namely the selection of the set of FUB implementations from the FUB library that best satisfies these constraints. This optimization problem

is then solved using an appropriate optimization software tool. The FUB implementations selected, referred to as the FUB template, are connected to form a netlist for the FUB array. This netlist can then be translated to the physical layout using a variety of methods, most commonly commercial place-and-route tools (see the discussion on physical design features in Section 2.2).

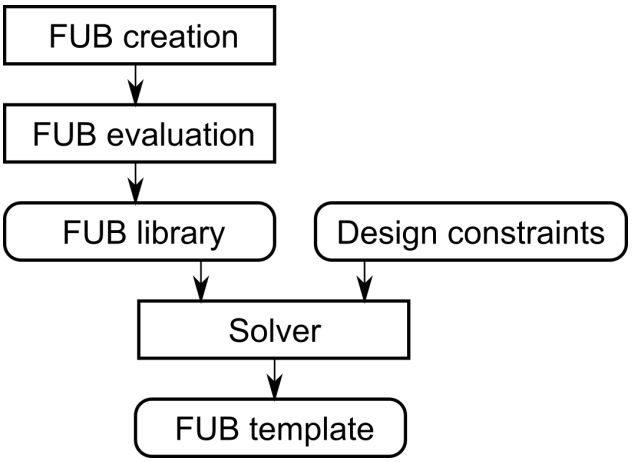
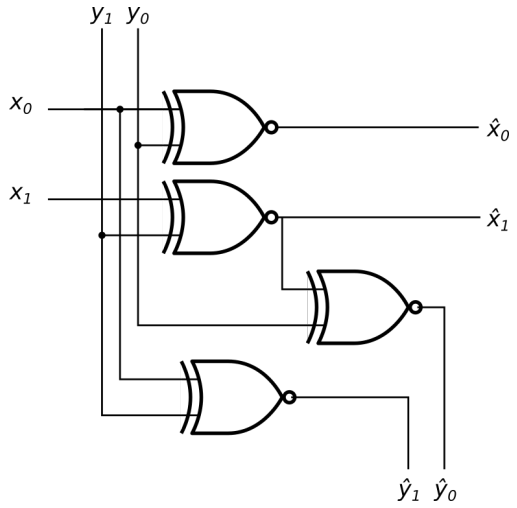


Figure 2.5: Synthesis flow used to create the FUB array.

Variations on the details of this implementation flow have been explored in previous work. In [49], the focus is on the standard cell composition of the FUB array, with particular emphasis on generating FUBs that utilize all of the standard cells and creating FUB templates that match a desired standard-cell distribution. Both [48, 50] extend the design constraints to include cell-aware fault coverage [30] and diagnostic coverage [51]. A different approach is explored in [52]: the metal interconnect is removed from an existing physical design, resulting in a collection of placed standard cells. A FUB template is then synthesized to utilize these placed standard cells, which are then routed together to create the final FUB array. While these implementation details are vital to the overall performance of the CM-LCV and continue to be the subject of future work, they remain orthogonal to the topics central to this dissertation, namely the test and diagnosis of the FUB array.

Input (x, y)	Output $(\hat{x}, \hat{y})$
00 00	00 00
00 01	01 11
00 10	10 10
00 11	11 01
01 00	01 10
01 01	00 01
01 10	11 00
01 11	10 11
10 00	10 01
10 01	11 10
10 10	00 11
10 11	01 00
11 00	11 11
11 01	10 00
11 10	01 01
11 11	00 10

(a)



(b)

Figure 2.6: (a) Truth table and (b) netlist implementation for the first VH-bijective function used in this work.

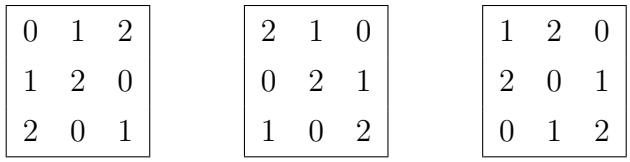
2.4 VH-bijective Functions

One of the defining characteristics of this thesis is the conception and development VH-bijectivity. This section examines VH-bijectivity in greater depth, specifically the mathematics involved in Section 2.4.1, and the enumeration of possible VH-bijective functions in Section 2.4.2.

2.4.1 VH-bijectivity and Orthogonal Latin Squares

The initial explorations that led to the identification of VH-bijectivity were entirely practical in nature (i.e., search for the characteristics of the FUB function that led to the best error propagation in a two-dimensional array). The development of the initial VH-bijective functions relied upon creating small circuits utilizing the XOR function (which is balanced (equal numbers of zeros and ones) and possesses the necessary error propagation properties) and subsequently checking for VH-bijectivity. Figure 2.6 shows both the truth table and XOR netlist implementation of the first VH-bijective function discovered in the course of this thesis work.

However, even a cursory examination suggests significant mathematical underpinnings to



(a) **(b)** **(c)**

Figure 2.7: Latin squares of order $n = 3$.

this property. Euler was the first to extensively describe these mathematical underpinnings with his work on Latin squares [53]. A Latin square is defined as an $n \times n$ matrix filled with symbols drawn from a set of size n such that, for each row and each column, each of the n symbols appears exactly once (for convenience this work will use the integers $\mathbb{Z}_n$ instead of arbitrary symbols). Figure 2.7 shows several such Latin squares for $n = 3$.

Note that a Latin square can be interpreted as defining a function on two inputs. The first input selects the row $x \in \mathbb{Z}_n$; the second input selects the column $y \in \mathbb{Z}_n$; and the output is the symbol found at that location in the Latin square, also an element of $\mathbb{Z}_n$. Because of the unique row and column properties of the Latin square, a change in either row or column (but not both) will result in a change in the output value. This closely mirrors some of the desired functional properties in the definition of VH-bijectivity; the key difference is that a single Latin square L cannot define a bijective function as its domain is larger than its range: $L : \mathbb{Z}_n^2 \to \mathbb{Z}_n$.

A simple way to overcome this shortcoming is to examine a pair of Latin squares. Two Latin squares L_1, L_2 together define a function $L_1L_2 : \mathbb{Z}_n^2 \to \mathbb{Z}_n^2$ with its domain equivalent to its range. Two Latin squares of order n are said to be *orthogonal* if all of the n^2 ordered pairs produced by their combination are distinct. **Only pairs of orthogonal Latin squares define a bijective function**.

A VH-bijective function is thus equivalent to a pair of orthogonal Latin squares (POLS) when n is a power of two (allowing for full binary encoding of the n symbols). This work is thus focused on a very small subset of orthogonal Latin squares. Mathematicians have historically been drawn to other aspects of these topics; for example, orthogonality is not lim-

ited to just pairs of Latin squares, but can be extended to sets of mutually orthogonal Latin squares (MOLS). Significant effort has been spent searching for and enumerating the largest MOLS for a given order [54, 55]. Another problem of interest has been the non-existence of orthogonal Latin squares of certain orders. Euler originally hypothesized that orthogonal Latin squares do not exist for any order $n = 2 + 4k$, $k \geq 1$ after encountering difficulty in finding any of order $n = 6$ [53]. Later works confirmed the non-existence orthogonal Latin squares of order $n = 6$ [56] but disproved the remainder of Euler's conjecture, first with counter-examples of order $n = 22$ in [57] and later of order $n = 10$ in [58]. A relevant survey on this history of this conjecture can be found in [59]. Other applications for Latin squares include the statistical design of experiments [60, 61] and error correcting codes [62–66]. An excellent overview of Latin squares and orthogonality, along with other related concepts, can be found in [67, 68].

2.4.2 Enumerating VH-bijective Functions

Given the importance of these functions for the CM-LCV, questions remain as to how many such functions exist and which ones are best for the purposes of this work. The mathematical theory discussed in the previous section is helpful in addressing these questions. Creating new POLS is straightforward once one such pair has been constructed; some operations that preserve both "Latin"-ness and orthogonality [69] are:

- Permutation of the symbols of either square (i.e., mapping each symbol in a given square to a new symbol using a permutation).
- Permutation of the rows of both squares (i.e., swapping the order of the rows in the same manner for both squares).
- Permutation of the columns of both squares (i.e., swapping the order of the columns in the same manner for both squares).

n	Isotopism
2	1
3	1
4	1
5	3
6	0
7	34
8	45,927
9	2,203,310,919

Table 2.1: Number of reduced lists of 2-MOLS(n).

Given these operations, each of which is a permutation of order n, a loose upper bound on the number of possible POLS is thus $(n!)^4$. The actual number of unique POLS will be reduced by symmetries among the operations (i.e. combinations of permutations creating the same result). A loose lower bound of $(n!)^2$ can be derived by observing that the first operation, permutation of symbols of either square, defines two independent permutations that can never produce the same result. Even this loose lower bound is extremely large for the orders that are relevant in this work (most notably $n = 8$).

However, these operations are not sufficient to discover all possible POLS of a given order. Instead, these operations define a single symmetry class of POLS, and multiple such classes may exist for a given order. J. Egan and I. Wanless quantify how many such classes exist for orders $n \leq 9$ in [69]. In their terminology, the operations listed above define an isotopism, and POLS of order n are equivalent to lists of 2-tuples of mutually orthogonal Latin squares of order n (abbreviated 2-MOLS(n)). Reproduced in Table 2.1 is the number of isotopisms of lists of 2-MOLS(n) for various orders. While only one such isotopic class exists for order $n = 4$, 45,927 exist for order $n = 8$; considering the permutations possible within each isotopism, the number of possible VH-bijective functions quickly challenges computational limits even for relatively small orders.

The question remains, however, if the choice of function has an impact on the overall performance of the CM-LCV. Figure 2.8 is a histogram showing the estimated logic cost for ten thousand different POLS of order $n = 8$. The estimated logic cost was calculated using

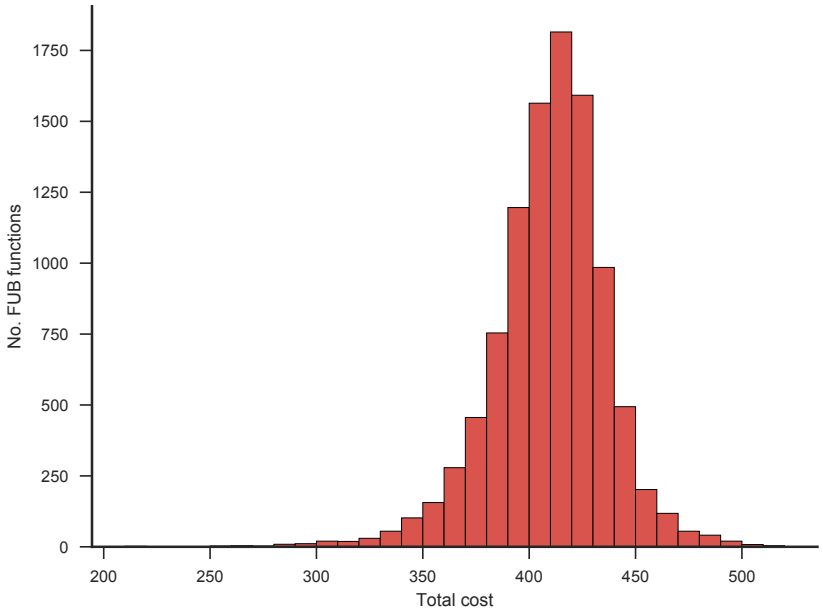


Figure 2.8: Histogram of the logic cost as estimated using the Espresso-MV logic minimizer tool for ten thousand POLS of order $n = 8$.

00	01	10	11
01	10	11	00
10	11	00	01
11	00	01	10

Figure 2.9: Latin square corresponding to the sum output of a 2-bit binary adder function.

the Espresso-MV logic minimization tool [70] and is equivalent to the total number of gate inputs necessary for a two-level logic implementation.

While it is evident that there is significant variance in the logical complexity required to implement different VH-bijective functions, this result at best only suggests that some variance in performance is expected for FUB arrays consisting of different VH-bijective functions. A more concrete example of the possible improvement in performance can be found by examining the adder circuit. The sum output bits of a binary adder function are equivalent to a Latin square, as shown in Figure 2.9 for 2-bits.

Thus, if this adder Latin square could be paired with another orthogonal Latin square, the resulting VH-bijective function could be implemented very efficiently using the a 2-bit adder. Unfortunately no orthogonal mate exists for this square, or for any other adder Latin square of even order, a result that was proven by Euler [53]. Nevertheless, this example illustrates one of the possible advantages that different VH-bijective functions have to offer, namely efficient usage of more complicated logic circuits that can be difficult to incorporate using current methods.

2.5 Design Variants

The FUB arrays discussed up to this point are subject to several limitations, the most significant of which are the use of a single FUB function throughout the array and the lack of sequential cells (e.g., flip-flops and latches). This section describes two specific variants that have been developed to address these shortcomings.

2.5.1 Heterogenous Arrays

Homogeneous arrays, that is, arrays composed of a single FUB function, are limited by the performance of the synthesized implementations of that FUB function. This has been a fundamental constraint on the performance of the FUB array, and has driven extensive effort in synthesizing and constructing additional variants of the FUB that address specific shortfalls. The use of multiple FUB functions within a single array provides another degree of freedom in the optimization problem, and thus is expected to result in better-performing designs overall.

A heterogenous FUB array consists of FUBs implementing any number of different VH-bijective functions. Note that the error propagation behavior (discussed in Section 2.1) and the composability of various properties (discussed in Section 2.2) do not rely on the particular VH-bijective FUB function being the same throughout the array, and thus are still applicable

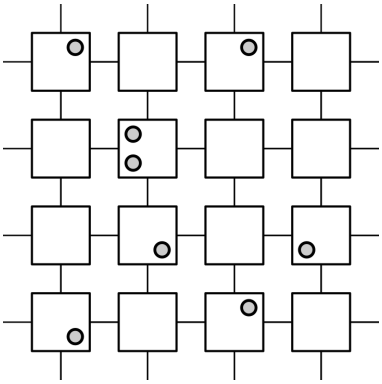


Figure 2.10: A 4×4 FUB array with sequential cells (represented as "$\circ$") inserted along random signal lines.

to heterogenous arrays. The construction of an array test set, however, is complicated by different FUB functions, and is discussed in detail in Section 3.1.3.

2.5.2 Sequential Arrays

The absence of sequential cells (i.e., flip-flops and latches) in the FUB array is problematic for several reasons. First, the defect relevance of the FUB array is decreased, that is, any defects that only affect sequential cells cannot by definition be detected if those cells are not present. Additionally, many defects cause changes in timing behavior, which can be difficult to detect when testing a purely combinational FUB array due to the limited observation capabilities of the test environment (i.e., the tester only observes the array outputs at certain designated time intervals). Sequential cells can assist in detecting defective timing behavior by transforming timing irregularities into irregularities in state, which is much simpler to propagate and observe from a test perspective. For example, a D-type flip-flop (DFF) is sensitive to changes in signal timing around the capture edge of the clock; if a transition to the data input arrives late, the improper value is captured by the DFF. Thus there is significant incentive to incorporate these sequential cells in the FUB array.

In a first approach, shown in Figure 2.10, sequential cells are inserted along random signal lines within the FUBs themselves. Regardless of the size of the array and number of sequential cells inserted, after a sufficient number of clock cycles, the state of the sequential

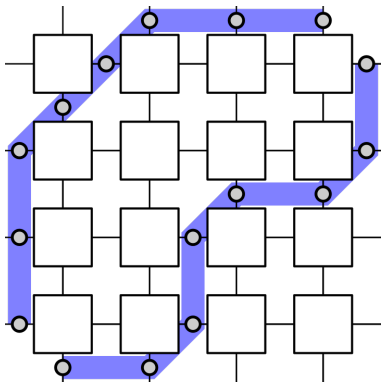


Figure 2.11: A 4×4 FUB array with sequential cells (represented as "○") inserted along two boundaries (represented by the shading) in the array. Any path from the array inputs to the array outputs must now traverse exactly two sequential cells.

cells will stabilize, and the array state will be identical to that of the original combinational FUB array (i.e., without inclusion of any sequential cells) with the same test pattern applied. Thus, any test or diagnosis scheme that is applicable to the combinational FUB array is also applicable to this sequential design. However, a significant downside to this approach is the complete lack of constraints on timing. In a conventional design, the clock speed is determined by the longest observed path delay, and changes in delay along a given path must exceed the slack (i.e., the difference between the clock period and the path delay) in order to be detected. Because the lengths of the paths between sequential cells in the FUB array of Figure 2.10 are determined by the random insertion process, it is impossible to design for a specific clock speed or delay sensitivity using this approach.

In a second approach, shown in Figure 2.11, sequential cells are inserted at every FUB port along a boundary in the array. If this boundary is chosen so as to properly bisect the flow of signals in the array (as is the case for both boundaries inserted into the array of Figure 2.11), all paths through the array will cross the same number of boundaries, and thus require the same number of clock cycles to propagate a transition through the entire array. However, the timing between the inserted sequential cells remains unconstrained as in the previous approach.

In a final approach, shown in Figure 2.12, sequential cells are inserted at every FUB

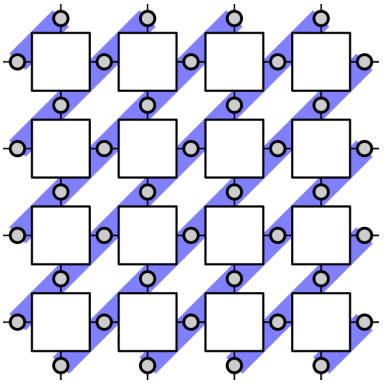


Figure 2.12: A 4×4 pipelined FUB array with sequential cells (represented as "$\circ$") inserted at every inter-FUB port. Every path from sequential cell to sequential cell in the array now traverses exactly one FUB.

boundary, resulting in a *pipelined* FUB array. This is effectively the second approach with the maximum number of boundaries inserted, and results in a significant benefit: timing in the FUB array is now composable. Because every path from one sequential element to another in the array traverses exactly one FUB, if the paths through each FUB are designed to meet a timing constraint, all paths in the pipelined FUB array will also meet that timing constraint. Furthermore, the composability of fault coverage can be extended to include faults detectable by two-pattern tests (e.g., transition, path delay, etc.): propagation remains guaranteed (due to the bijectivity of the FUB array), and controllability for two-pattern tests is demonstrated in Section 3.1.

It must be noted that there are practical limitations for at-speed test of the pipelined FUB array. The FUBs used in the CM-LCV are typically small (ranging from tens to a few hundred standard cells) with relatively few logic levels; at-speed test for such a low logic depth in modern technology nodes requires an extremely fast clock, which would be nearly impossible to achieve in a test environment. Nevertheless, this approach remains valuable. Below-speed two-pattern tests can still detect gross timing or sequential behavior that might escape a single-pattern test scheme. Additionally, techniques such as increasing the logical depth of the FUBs (for example, by increasing the size (number of input bits) of the FUB function used and adding buffer cells to the FUB implementations) or slowing signals by

reducing the supply voltage may allow for a slower clock to be sufficient.

Given the numerous advantages of the pipelined FUB array approach, the incorporation of sequential cells in the remainder of this work will exclusively utilize it. Additionally, the only sequential cells considered outside of this section will be D-type flip-flops (DFFs) in order to simplify discussion.

2.6 Summary

This chapter has described a new logic test chip design, called the Carnegie Mellon Logic Characterization Vehicle. Unlike conventional logic test chips, the CM-LCV begins with an innovative logic design and implements it using a physical design adapted to meet the test chip requirements. This innovative logic design is a two-dimensional array of functional unit blocks (FUBs). A new property, VH-bijectivity, is defined and applied to the FUB functions, allowing for C-testability and the maximal propagation of errors within the FUB array. Additionally, composability is demonstrated within the FUB array for logic and physical design features, fault coverage, and fault distinguishability (with some limitations). This composability is leveraged to create a synthesis flow for the CM-LCV. Additionally, the connections between orthogonal Latin squares and VH-bijectivity are defined and used to demonstrate the large number of possible VH-bijective functions. Finally, two variants on the FUB array used in the CM-LCV are proposed: the heterogenous FUB array (with different FUB functions at different array locations) and the pipelined FUB array (with DFFs added to the FUB array connections).

Chapter 3

CM-LCV Test

The testability of the Carnegie Mellon Logic Characterization Vehicle (CM-LCV) is described in this chapter. Techniques for the construction of high-quality test patterns for the FUB array and its variants are presented in Section 3.1. A built-in self test (BIST) architecture that leverages the unique properties of the FUB array is described in Section 3.2. Finally, Section 3.3 summarizes the contributions described in this chapter.

3.1 Test Construction

While fault coverage within the FUB array is composable (Section 2.2), the construction of efficient test patterns for the FUB array is addressed here. The remainder of this section describes methods for test pattern construction for the FUB array and its heterogenous and pipelined variants.

3.1.1 Tessellation Patterns

Testing a FUB array contained within a CM-LCV is accomplished by exploiting the bijective nature of the FUB functions. A test detects a fault if it satisfies the activation conditions for the fault and it propagates any resulting error to the circuit output(s). FUB bijectivity

guarantees not only propagation of errors to the array outputs, but also perfect controllability for any location in the array. Recall that a FUB function implements a mapping $F(x, y) \rightarrow (\hat{x}, \hat{y})$; let $p = (x, y)$ denote the input pattern to the FUB, and $\hat{p} = (\hat{x}, \hat{y})$ denote the FUB response, and similarly let $P = (X, Y)$ and $\hat{P} = (\hat{X}, \hat{Y})$ denote the input pattern and output response of the FUB array. A cycle of function F is defined as a sequence of input patterns $K = \{p^{(0)}, p^{(1)}, \ldots, p^{(|K|-1)}\}$ such that $F(p^{(i)}) = \hat{p}^{(i)} = p^{((i+1) mod\ |K|)}$ and $p^{(i)} \neq p^{(j)}$ for all $p^{(i)}, p^{(j)} \in K$. Given these definitions, Theorem 3.1 describes a method for constructing a test set for a FUB array from a cycle K of the FUB function F.

Theorem 3.1. *Given an $R \times C$ homogeneous array of FUBs which each implement bijective function F with cycle K of length k, there exists an array test set T of size k such that all input patterns in K are applied to all FUBs in the array when the tests of T are applied, regardless of the values of R and C.*

Proof. Assume a cycle K within F of length k, that is, $K = \{p^{(1)}, p^{(2)}, \ldots, p^{(k)}\}$, where $p^{(i)} = (x^{(i)}, y^{(i)})$. Let T be composed of k array input patterns, that is, $T = \{P_1, P_2, \ldots, P_k\}$, and let each $P_s \in T$ be constructed according to the following:

$$X_s = (x^{(s)}, x^{(s+1) mod\ k}, \ldots, x^{(s+R-1) mod\ k})$$
$$Y_s = (y^{(s)}, y^{(s+1) mod\ k}, \ldots, y^{(s+C-1) mod\ k})$$

Now consider all of the FUBs along a given diagonal of the array, that is, all $F_{(i,j)}$ where $i + j = \lambda$, as illustrated in Figure 3.1.

- For $\lambda = 2$, only FUB $F_{(1,1)}$ meets the constraint. Observe that for test P_s, $x_{(1,1)} = x^{(s)}$ (determined by X_s) and $y_{(1,1)} = y^{(s)}$ (determined by Y_s); thus $\hat{p}_{(1,1)} = F(p^{(s)}) = p^{(s+1) mod\ k}$.

- For $\lambda = 3$, FUBs $F_{(2,1)}$ and $F_{(1,2)}$ meet the constraint. Observe that for test P_s,

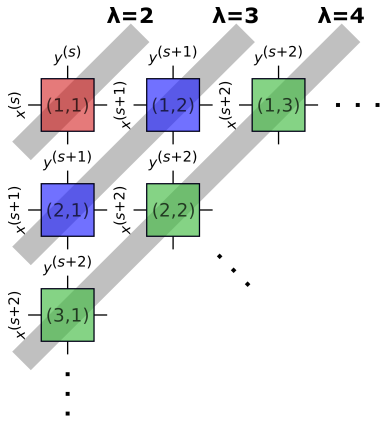


Figure 3.1: Visualization of the FUB input patterns applied to the FUBs along the diagonals of the array by a test derived from a cycle in the FUB function F. Note that there is a *mod k* (not shown due to space constraints) on all horizontal and vertical superscripts.

$p_{(2,1)} = p^{(s+1) mod\ k}$ (determined by X_s and $\hat{y}_{(1,1)}$); thus $\hat{p}_{(2,1)} = p^{(s+2) mod\ k}$. Similarly, for test P_s, $p_{(1,2)} = p^{(s+1) mod\ k}$ (determined by X_s and $\hat{x}_{(1,1)}$); thus $\hat{p}_{(1,2)} = p^{(s+2) mod\ k}$.

- For $\lambda > 3$, $x_{(i,j)}$ will be determined by either X_s or a FUB on diagonal $\lambda - 1$; in both cases $x_{(i,j)} = x^{(s+\lambda-2) mod\ k}$. Similarly $y_{(i,j)}$ will be determined by either Y_s or a FUB on diagonal $\lambda - 1$; in both cases $y_{(i,j)} = y^{(s+\lambda-2) mod\ k}$. Thus $p_{(i,j)} = p^{(s+\lambda-2) mod\ k}$, and therefore $\hat{p}_{(i,j)} = F(p^{(s+\lambda-2) mod\ k}) = p^{(s+\lambda-1) mod\ k}$.

Thus for any test P_s, all FUBs along the diagonal defined by λ have input pattern $p^{(s+\lambda-2) mod\ k}$ applied. Given that s ranges from 1 to k over the k tests in T, $p^{(s+\lambda-2) mod\ k}$ will cover all $p \in K$ irrespective of λ; thus, all FUBs along all diagonals will experience all input patterns in K over the application of the k tests in T. □

Figure 3.2 demonstrates how Theorem 3.1 can be used to construct a minimal test set that exhaustively applies all input patterns to each FUB in an array of arbitrary size. A 2-bit bijective FUB function $F(x, y)$ (Figure 3.2a) can be represented by a directed graph with a node for each input pattern (x, y), with transitions between the nodes determined by the function: $(x, y) \rightarrow F(x, y)$ (Figure 3.2b). Because the FUB function is bijective, all nodes in this graph are guaranteed to belong to a cycle. Test vectors for a FUB array of arbitrary size can be created using Theorem 3.1 for each cycle/input pattern combination.

In the example shown in Figure 3.2, the input pattern (11) is selected and applied to the FUB at array position (1, 1). The resulting test vector applies the 3-cycle (11, 10, 01) along the 45° diagonals of the array. Different test vectors are created by starting with different input patterns from the cycle (11, 10, 01) for the FUB at array position (1, 1), applying all three input patterns to every array FUBs with the minimal number of test vectors (equal to the length of the cycle). Repeating this process for all input patterns and cycles results in a set of N test vectors that apply all input patterns to all FUBs in the array, where N is the total number of FUB-level input patterns (in the example shown in Figure 3.2, $N = 4$ for the 2-bit function). These tests are collectively referred to as *tessellation* test patterns due to the repeating patterns that they induce in the array, and have been previously examined in the context of testing iterative logic arrays [71].

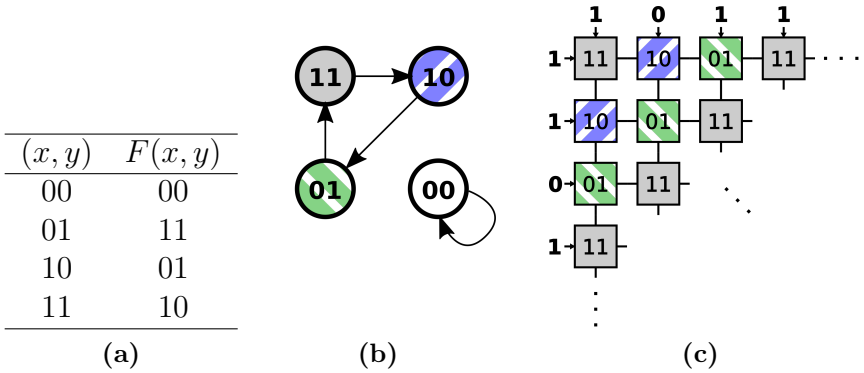


Figure 3.2: Method for creating test patterns for the FUB array. (a) The FUB function can be represented as (b) a directed cyclic graph on the possible input/output values for the FUB. (c) Cycles in the directed cyclic graph can be used to create test patterns for the FUB array.

3.1.2 Super-exhaustive Test

The construction methods for the tessellation test patterns in Section 3.1.1 are not limited to cycles from the function of a single FUB. In addition, sub-arrays of FUBs can be used to generate tessellation test patterns for a FUB array. Theorem 3.2 first establishes that any array (or sub-array) of bijective FUBs implements a bijective array function.

Theorem 3.2. *An $R \times C$ array of FUBs implementing bijective functions $F_1, F_2, \ldots, F_{R \times C}$ is equivalent to a single bijective function $G(X, Y) \rightarrow (\hat{X}, \hat{Y})$ on the array inputs and outputs.*

Proof. Suppose two array inputs $P_i \neq P_j$ both evaluate to the same array output $\hat{P}_{ij}$. The difference between P_i and P_j can be represented by a set of errors on the array inputs. Because every FUB function in the array is bijective, these errors are guaranteed to propagate through the array to some array output. Thus $G(P_i) \neq G(P_j)$ for all $P_i \neq P_j$, that is, G is one-to-one. Furthermore, because the array input space (X, Y) is the same size as the array output space $(\hat{X}, \hat{Y})$, G must be onto. Therefore G is bijective. □

In this way, any sub-array can be used to generate additional tessellation test patterns for the FUB array. The total number of such tessellation test patterns for a given sub-array scales up exponentially with the sub-array size; for example, given a 2-bit FUB function, a 1×2 sub-array implements a 3-bit bijective function, which will result in $N = 8$ tessellation test patterns. Figure 3.3 demonstrates this approach using a 1×2 sub-array for a 4×4 FUB array with the FUB function F of Figure 3.2. The 1×2 function $G(x, y)$ is presented in Figure 3.3a; note that the y input for $G(x, y)$ now consists of two bits (due to the two column inputs to the sub-array). This sub-array function has two cycles, of lengths one and seven, which are shown in Figure 3.3b. The cycle of length seven is used to create a tessellation pattern for the original 4×4 FUB array; Figure 3.3c shows the input patterns applied to each sub-array, while Figure 3.3d shows the input patterns applied to each FUB. The result is a tessellation pattern as before, but the repeating pattern now occurs on a different array diagonal (26.5° instead of 45° as illustrated in Figure 3.2).

The complete set of tessellation test patterns created in this way, from sub-arrays within the FUB array, are collectively referred to as a *super-exhaustive* test set. This is due to the fact that they apply every input pattern to each FUB in the array multiple times; and, furthermore, the neighboring values in the FUB array are different for each application of the same input pattern. This effect is demonstrated in Figure 3.4, which shows the input patterns applied to the FUBs in the 1×2 sub-array used in Figure 3.3 for all eight possible

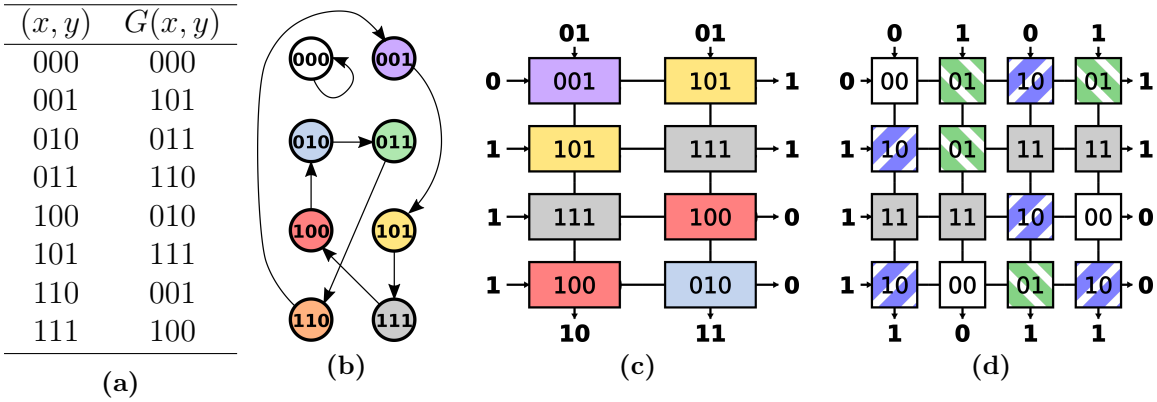


Figure 3.3: Method for creating super-exhaustive test patterns for a 4×4 FUB array with FUB function $F(x, y)$ from Figure 3.2. (a) The 1×2 sub-array function $G(x, y)$ can be represented as (b) a directed cyclic graph on the possible input/output values for the sub-array. Cycles in the directed cyclic graph can be used to create a test pattern for the FUB array. (c) The input patterns applied to each sub-array are constant along the diagonals as before. (d) The same is no longer true for input patterns applied to each FUB in the original FUB array for the same test pattern.

sub-array input patterns. Both FUBs in the 1×2 sub-array experience each FUB-level input pattern (00, 01, 10, 11) twice, and the input pattern applied to the neighbor FUB is different between the two; for example, the left FUB experiences the (00) input pattern in Figures 3.4a and 3.4b while the right FUB experiences input patterns (00, 01).

The result is that the super-exhaustive test set offers several advantages over the smaller test set derived from the single FUB function. The super-exhaustive test set is closely related to the concept of N-detect test [31] and physically-aware N-detect test [72,73], where test sets are constructed such that they detect faults multiple times. N-detect tests have improved defect detection rates compared to single-detect test sets [31, 74, 75], a property that the super-exhaustive test sets are expected to share. Additionally, the super-exhaustive test set provides more data for diagnosis; this effect is examined in greater detail in Section 4.3.

3.1.3 Heterogenous Array Test

A requirement for the construction of tessellation test patterns is that the FUB array is homogenous, that is, that every FUB implements the same function. Figure 3.5 shows how the tessellation test patterns break down for a heterogenous array. A 4×4 heterogenous

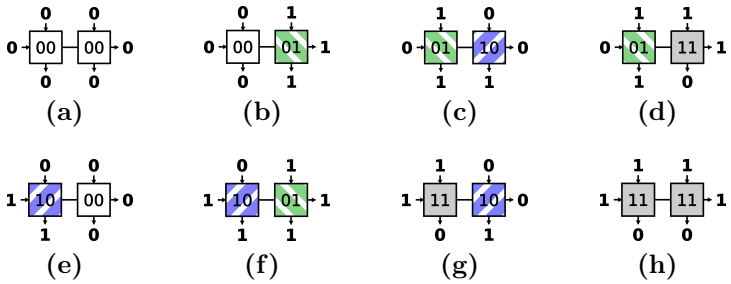


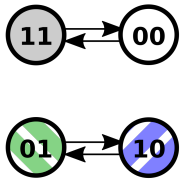
Figure 3.4: Demonstration of the eight (a-h) 1×2 sub-array input patterns applied to a 1×2 FUB array with FUB function $F(x, y)$ from Figure 3.2. Each FUB in the array experiences the four FUB-level input patterns twice.

FUB array is shown in Figure 3.5c, with FUBs implementing the function $F(x, y)$ (from Figure 3.2a) and a new 2-bit bijective function $H(x, y)$ (Figure 3.5a) indicated by the square and diamond symbols in the array, respectively. The cycles of $H(x, y)$ are represented as a directed cyclic graph in Figure 3.5b. Figure 3.5c shows the application of one tessellation test pattern constructed using the FUB function $F(x, y)$. The repeating pattern along the 45° diagonal of the array is disrupted by the presence of the second FUB function $H(x, y)$ at array position (2, 3). As a result, all FUBs downstream (i.e. array position (i, j), $i \geq 2$ and $j \geq 3$) of the FUB implementing $H(x, y)$ are no longer guaranteed to have all of the FUB-level input patterns applied from the cycles used to generate the tessellation test patterns. Note that any missing input patterns can still be applied (due to the controllability implicit in an array of bijective FUBs); the difference is that more than N test patterns may now be required.

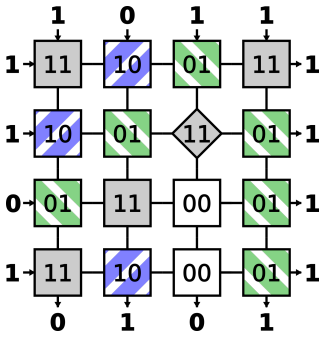
It is, however, possible to construct tessellation test patterns for a heterogenous FUB array if the placement of the different FUB functions is carefully controlled. More specifically, if every FUB along the diagonals created by the tessellation test patterns (e.g., the 45° diagonal in the Figures 3.2-3.5) implements the same bijective function, then all input patterns can be applied to every FUB in the array using N tessellation test patterns. The technique for constructing these patterns is identical to that outlined in Section 3.1.1 except that the FUB function appropriate for each diagonal (e.g., $F(x, y), H(x, y)$) is used in the second step. Figure 3.6 shows the $N = 4$ tests constructed for a 4×4 heterogenous array

(x, y)	$H(x, y)$
00	11
01	10
10	01
11	00

(a)



(b)



(c)

Figure 3.5: Breakdown of tessellation test patterns in a heterogenous FUB array. (a) A second FUB function $H(x, y)$ can be represented as (b) a directed cyclic graph on the possible input/output values for the FUB. Differences in the cycles defined by the two FUB functions $F(x, y), H(x, y)$ result in (c) a breakdown of the tessellation test patterns for the heterogenous FUB array (with the FUB implementing $H(x, y)$ indicated by the diamond symbol in the array).

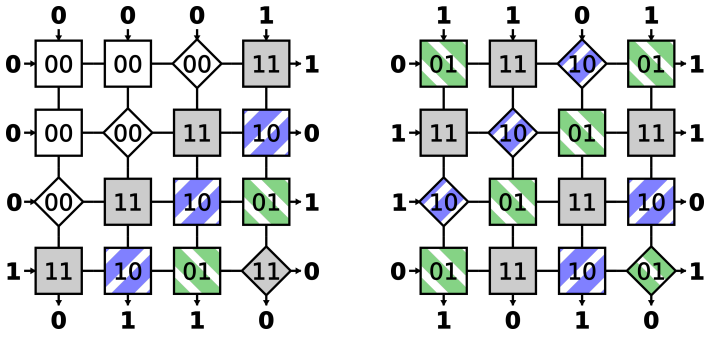


Figure 3.6: The four tessellation test patterns (a-d) constructed for a heterogenous array with consistent functions along the tessellation diagonals. The two FUB functions $F(x, y), H(x, y)$ are defined in Figures 3.2 and 3.5, respectively, with FUBs implementing $H(x, y)$ represented by the diamond symbol in the array.

constructed with consistent diagonal functions and using the same $F(x, y), H(x, y)$ functions described in Figures 3.2 and 3.5, respectively.

3.1.4 Sequential Array Test

Any tessellation test patterns created for a FUB array using the techniques of Section 3.1.1 can be applied to a pipelined version of the same array. Recall that a pipelined FUB array includes a D-type flip-flop (DFF) at every inter-FUB connection, with all DFFs in the FUB array driven by a single clock signal. Clocking this design while applying a tessellation test pattern to the array inputs will allow that tessellation test pattern to propagate through the array. Given enough clock cycles, the state of the pipelined FUB array will eventually match the state of an equivalent combinational FUB array with the same tessellation test pattern applied. Any static (i.e., timing independent) faults that are detected in the combinational FUB array by this tessellation test pattern will also be detected and propagated to the array outputs in the pipelined FUB array. The total number of clock cycles required to completely apply a tessellation test pattern (or any other test pattern) to the pipelined FUB array is determined by the array size; more specifically, $R + C$ clock cycles are necessary for test pattern application in a $R \times C$ pipelined FUB array.

The pipelined FUB array can also be used to detect delay faults with two-pattern tests. For simplicity, it is assumed in this section that every input (output) of the FUB array can be controlled (observed) at each clock cycle. First a parallel test scheme is considered, as shown in Figure 3.7 for a 4×4 pipelined FUB array with FUB function $F(x, y)$ as defined in Figure 3.2a. The FUBs and DFFs are represented by squares and circles, respectively. The values depicted correspond to the FUB inputs before a new clock edge arrives, with differences from the previous FUB inputs highlighted in red. The array begins with one tessellation test pattern applied, while a second tessellation test pattern is applied in parallel to all of the array inputs (Figure 3.7a). The succeeding clock cycles propagate this second tessellation test pattern through the array, until the final output is captured at the array outputs after a total of eight clock cycles (Figure 3.7i). The result of this test sequence is that every FUB in the array begins with one input pattern applied, in this case (00) for all FUBs, and ends with a different input pattern applied, in this case one of (01, 10, 11) depending on the FUB

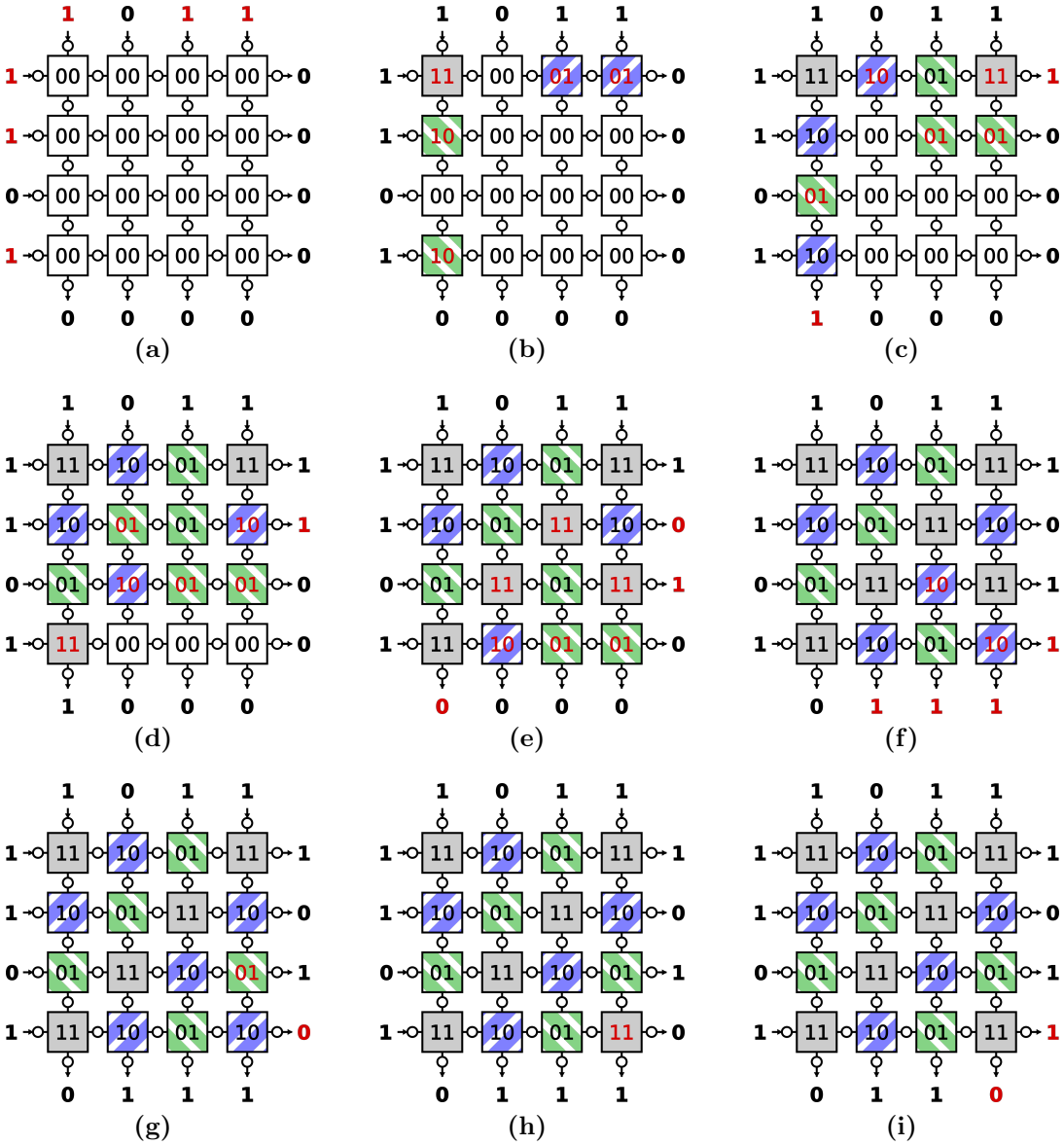


Figure 3.7: Parallel application of a tessellation test pattern to a pipelined 4×4 FUB array, with DFFs represented as circle symbols on all of the FUB ports and the values applied to the inputs of each FUB represented within the FUBs. Each clock cycle (a-i) represents the state of the FUB array, with differences from the previous clock cycle highlighted in red.

location within the array.

However, control over the exact input transitions applied to each FUB in the array is limited in this parallel test scheme: only the initial and final input patterns can be guaranteed for each FUB in the array. Consider the FUB at the top right corner (array position (1, 4)) of Figure 3.7: it begins with input pattern (00) and ends with input pattern (11) applied, however the actual sequence of input patterns applied over the eight cycles is (00 $\rightarrow$ 01 $\rightarrow$

11 → 11 → 11 → 11 → 11 → 11). It would be ideal if the sequence of input patterns applied to each FUB in the array could be controlled such that every pair of input patterns is applied to each FUB in the pipelined FUB array (equivalent to an exhaustive two-pattern test of all of the FUBs).

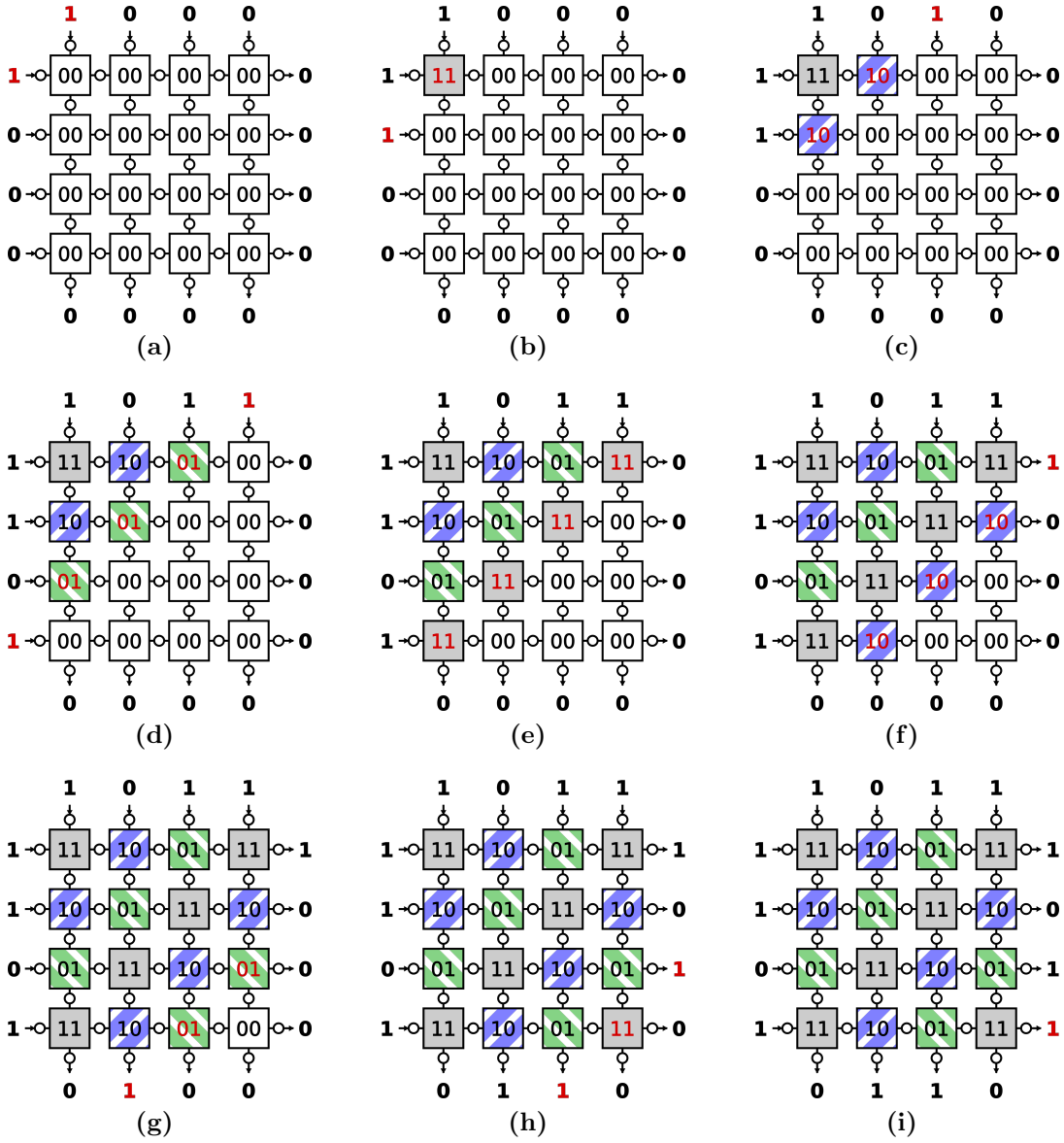


Figure 3.8: Serial application of a tessellation test pattern to a pipelined 4×4 FUB array, with DFFs represented as circle symbols on all of the FUB ports and the values applied to the inputs of each FUB represented within the FUBs. Each clock cycle (a-i) represents the state of the FUB array, with differences from the previous clock cycle highlighted in red.

This control problem can be resolved if a serial test scheme is used. Figure 3.8 demonstrates this serial test scheme for the same pipelined FUB array and test patterns of Figure

3.7. Again the array begins with one tessellation test pattern having been successfully applied (Figure 3.8a), but now the application (and observation) of the second tessellation test pattern proceeds at a rate of one array row and column per clock cycle. The process begins at the top left corner (Figure 3.8a) and ends at the bottom right array corner (Figure 3.8i) after eight clock cycles. The result is that the transitions applied to the FUBs are synchronized along the 45° array diagonals, every FUB in the array undergoes exactly one transition, and the array outputs are captured and observed without delay. In this way the uncontrolled FUB input transitions observed in the parallel test scheme of Figure 3.7 are resolved. Furthermore, a simple pairing of non-identical tessellation test patterns yields a two-pattern test set of size $N(N-1)$ with the property that it exhaustively applies every $N(N-1)$ pair of input patterns to each FUB in the pipelined FUB array.

3.2 Built-in Self Test

Built-in self test (BIST) is a design methodology that seeks to reduce the cost of test by implementing some or all of the test application and response collection in design itself. This section describes an efficient BIST scheme developed for the FUB array. The key property used in this section is that the output of the FUB array when a tessellation test pattern is applied is equivalent to another tessellation test pattern, given some constraints on the construction and composition of the FUB array. A simple BIST scheme, termed the circular FUB array BIST (CFA-BIST), is thus proposed, consisting of a feedback path for the FUB array that allows for the application of the tessellation test patterns, which have ideal fault detection properties (as established in Section 3.1). Section 3.2.1 describes in greater detail the constraints on the FUB array required. The design changes required for the CFA-BIST scheme is presented in Section 3.2.2. Finally, the CFA-BIST scheme is evaluated in Section 3.2.3.

3.2.1 Tessellation Test Theory

The key property exploited for the CFA-BIST is that the output of the FUB array when a tessellation test pattern is applied is equivalent to another tessellation test pattern given some constraints on the size of the FUB array. First, it is proved in Theorem 3.3 that this holds true for the tessellation test patterns of a square FUB array (i.e., the number of rows R is equivalent to the number of columns C).

Theorem 3.3. *Given an $R \times C$ (where $R = C$) array of FUBs implementing bijective function $F(x, y)$ with cycle K of length k and the test set T constructed from K according to Theorem 3.1, the array output $\hat{P}_s \in T$ for all test patterns $P_s \in T$.*

Proof. Consider test $P_s \in T$ derived from cycle K. The proof for Theorem 3.1 demonstrated that, for a fixed $\lambda = i + j$, $p_{(i,j)} = p^{(s+\lambda-2) mod\ k}$. Thus by definition $\hat{p}_{(i,j)} = p^{(s+\lambda-1) mod\ k}$. However, recall that $\hat{P}_s = (\hat{X}_s, \hat{Y}_s)$ where in this case:

$$\hat{X}_s = (\hat{x}_{(1,C)}, \hat{x}_{(2,C)}, \ldots, \hat{x}_{(R,C)})$$
$$\hat{Y}_s = (\hat{y}_{(R,1)}, \hat{y}_{(R,2)}, \ldots, \hat{y}_{(R,C)})$$

Substituting according to λ these become:

$$\hat{X}_s = (x^{(s+C) mod\ k}, x^{(s+C+1) mod\ k}, \ldots, x^{(s+2C-1) mod\ k})$$
$$\hat{Y}_s = (y^{(s+R) mod\ k}, y^{(s+R+1) mod\ k}, \ldots, y^{(s+2R-1) mod\ k})$$

This $\hat{P}_s$ is identical to $P_{s'} \in T$ where $P_{s'}$ is constructed from cycle K starting at $p^{(s+Q) mod\ k}$; thus $\hat{P}_s \in T$. □

Thus a set of tessellation test patterns constructed from a single cycle K according to Theorem 3.1 can be applied in its entirety to a square FUB array by iteratively feeding

the array output values back into the array inputs. Applying all of the tessellation test patterns in this manner requires loading a new starting pattern for each different cycle used to generate the tessellation patterns of interest. If there are many cycles in the tessellation patterns of interest, utilizing this test pattern feedback property offers little advantage over applying each tessellation test pattern individually.

To circumvent this limitation, Theorem 3.2 is again utilized. According to Theorem 3.2, any $r \times c$ sub-array within a FUB array can be represented as a single bijective function. Furthermore, if the dimensions of a FUB array are a scalar multiple of $r \times c$, the overall array can be considered a square array of FUBs implementing this sub-array bijective function. Thus a test set for this array can be derived using Theorem 3.1 and the cycles of the sub-array bijective function (effectively a super-exhaustive test set, as described in Section 3.1.2). Assuming some limited design freedom for the FUB array dimensions, it is expected that this process can be used to derive optimal test sets for the FUB array by examining the cycles of various sub-array sizes. Exploiting Theorem 3.2 for this purpose is further examined in Section 3.2.3.

3.2.2 BIST Architecture

The unique properties of the FUB array established in Section 3.2.1 lend themselves well to a feedback architecture. Previous BIST architectures with circular feedback include the Circular Self-Test Path (CTSP) [76] and the Circular Celluar BIST (C^2BIST) [77]. However, both of these methods require additional circuitry along the feedback path, which is not necessary for the FUB array in this case. Instead, implementation of the CFA-BIST is accomplished by a single scan chain around the FUB array periphery. Circular feedback is added to connect the array outputs to the normal-mode inputs of the scan chains that drive the array input. Thus, for normal-mode operation (i.e., when the scan enable signal is not asserted), the scan chains feeding the array are updated with the array output values.

Two variants of this CFA-BIST architecture based on the length of the scan chain are ex-

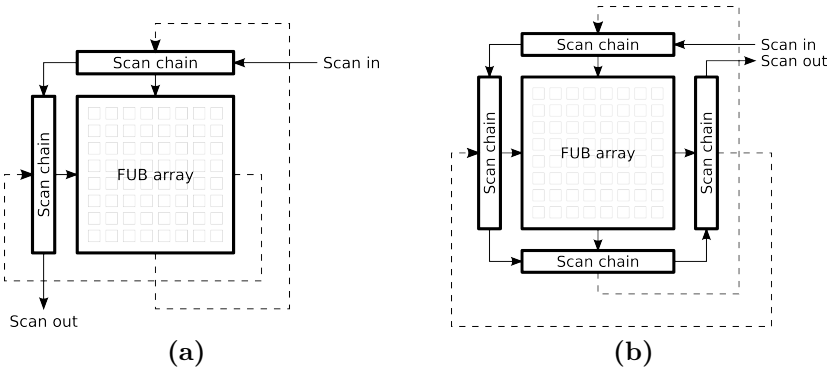


Figure 3.9: Diagrams for (a) the short-chain variant and (b) the long-chain variant of the CFA-BIST design. The feedback paths required are represented by the dashed lines.

plored: Figure 3.9a uses the minimum scan chain length required, while Figure 3.9b extends the scan chain to capture the outputs in a second set of flip-flops. While the short-chain variant requires less hardware overhead and has a shorter scan chain length, the long-chain variant has several advantages:

- Diagnosability - the additional flip-flops in the long-chain implementation allow the feedback connections to be tested independent of the logic array. This results in enhanced resolution because the short-chain implementation cannot distinguish between a failure in the FUBs located on the output edge of the array and a failure due to the feedback connections.

- Test independence - the long-chain implementation allows for two independent test patterns to be applied during BIST mode. Specifically, one pattern can be applied to the logic array by the input scan chain while a second pattern is simultaneously transferred from the output scan chain to the input scan chain via the feedback connections.

Execution of the CFA-BIST test cycle is achieved through three steps, as shown in the timing diagrams of Figure 3.10. First, a seed pattern is loaded into the scan chain (shown as "Load/Unload" in Figure 3.10). Second, a series of BIST-mode clocks are applied to execute the BIST cycle ("Run BIST" in Figure 3.10). Finally, the resulting values are unloaded from the scan chain and compared to the expected, fault-free signature, and a new seed pattern

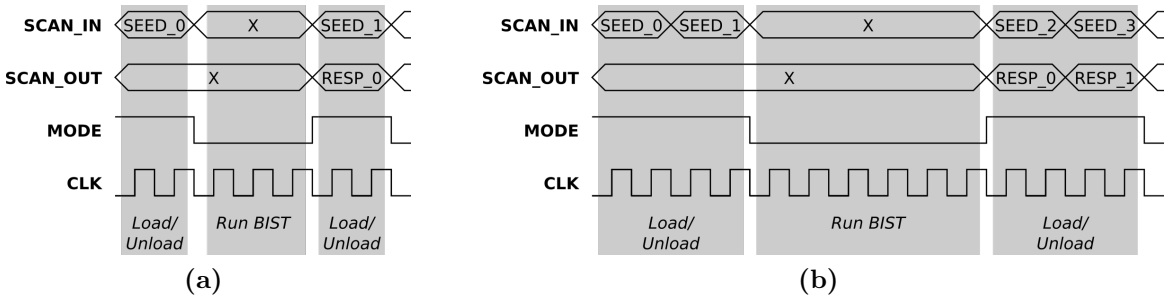


Figure 3.10: Timing diagrams for (a) the short-chain and (b) the long-chain CFA-BIST architectures.

is simultaneously loaded into the scan chain if applicable. The only difference between the timing for the short-chain (Figure 3.10a) and long-chain (Figure 3.10b) variants is that the long-chain architecture processes two tessellation patterns during the load/unload step, and requires more BIST-mode clock cycles to fully proceed through the cycles of the two loaded tessellation patterns.

3.2.3 BIST Evaluation

In this section a FUB array constructed from the 4-input VH-bijective FUB function described in Figure 2.6 is used to evaluate the CFA-BIST architecture. First, a high-quality test set is derived according to Theorem 3.1. Section 3.2.1 stated that an array can be considered a square array of FUBs implementing the sub-array function (assuming the array dimensions are a multiple of the sub-array dimensions). Thus, given freedom over the size of the FUB array, the cycles within the tessellation test sets derived from various sub-array sizes are all of interest as potential sources for a CFA-BIST test set. Table 3.1 lists these cycle lengths for various sub-array sizes; each entry under the column labeled "Cycle lengths" denotes a specific cycle length present as well as the number of such cycles (shown in parentheses if more than one). Note that the cycle length is equivalent to the number of BIST-mode clock cycles that can be applied using a tessellation pattern derived from that cycle before the tests begin to repeat (and a new seed pattern would need to be loaded).

The data shown in Table 3.1 support the previous assertion that BIST cycles of varying length can be found by considering various sub-array sizes. Of note are the single cycle of

Sub-array dimensions	Cycle lengths
1×1	1, 15
1×2	1, 7(×9)
2×1	1, 21(×3)
2×2	1, 15(×17)
2×3	1, 7, 127, 889
3×2	1, 31(×33)
3×3	1, 5(×3), 85(×48)
3×4	1, 127(×129)
4×3	1, 5461(×3)
4×4	1, 15(×4369)

Table 3.1: Cycle lengths for functions corresponding to sub-arrays of FUBs of various dimensions.

length 889 present in the 2×3 sub-array bijective function, and the three cycles of length 5,461 present in the 4×3 sub-array bijective function. Recalling Theorem 3.1, the existence of these cycles indicates that it is possible to create, for example, a design with a CFA-BIST test cycle of length 5,461 if the FUB array dimensions are a multiple of 4×3. The remainder of this section focuses on test sets derived from the 889 test cycle derived from the 2×3 sub-array bijective function for a 6×9 array.

Quantifying fault coverage for the proposed circular BIST architecture using fault simulation is computationally expensive since the errors produced may be masked when repeatedly fed back through the faulty array. To mitigate this issue, fault coverage can instead be equated to the combination of (a) fault activation, and (b) the probability of error masking.

First the fault activation achieved by a test set is examined. A single-pattern input pattern (IP) fault model [29] is assumed for each FUB in the array. Thus, in order to achieve 100% fault activation, the test set must apply all $2^4 = 16$ possible input patterns to each FUB in the array. Figure 3.11 shows how fault activation evolves as more tests are applied from the 889 tests derived from the 889-length cycle, and it indicates that the test set achieves 100% fault activation for all locations in the array by the 89th test pattern.

However, the fault activation results of Figure 3.11 is not indicative of all the capabilities of this test set. Activation for each 2×2 sub-array within the overall array translates to all

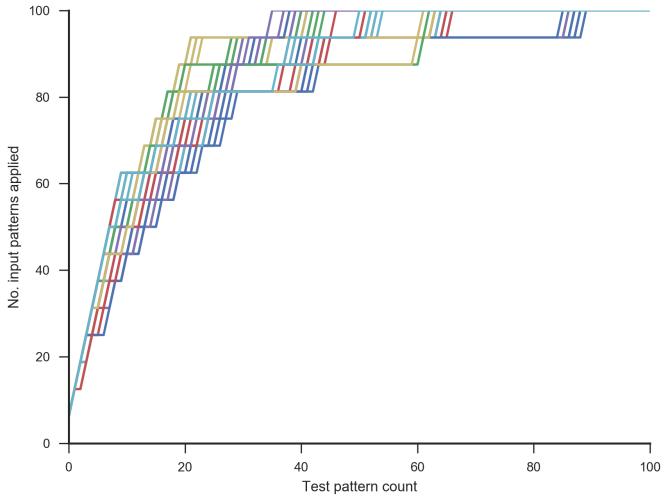


Figure 3.11: Evolution of IP fault activation over test pattern count for a 6×9 array using a test set derived from the 2×3 sub-array cycle of length 889. Each trace represents the IP fault activation for each FUB in the 6×9 array (note that there is significant overlap among the traces).

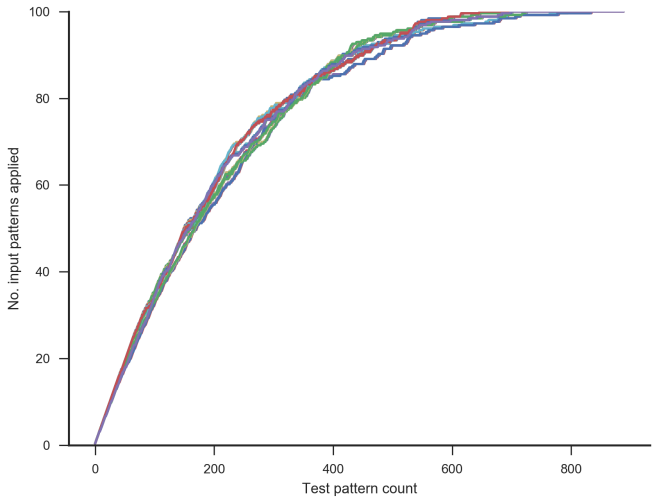


Figure 3.12: Evolution of IP fault activation for 2×2 sub-arrays over test pattern count for a 6×9 array using a test set derived from the 2×3 sub-array cycle of length 889. Each trace represents the IP fault activation for each unique 2×2 sub-array in the 6×9 array (note that there is significant overlap among the traces).

2^8 possible input patterns applied to the 8-input sub-array. From another perspective, fault activation for each 2×2 sub-array is equivalent to a 16-detect (or, rather, 16-activate) test set for each individual FUB. Figure 3.12 shows how fault activation evolves for all 2×2 sub-arrays for the same test set and 6×9 array. The test set achieves 100% IP fault activation

Metric	Scan test	Circular BIST
Scan chain length	30	30
Clock cycles	7,971	953
SSL fault coverage	100%	100%
Gate-level IP fault coverage	100%	100%
Simulation time (min.)	12.9	261.3

Table 3.2: Summary of simulation results for both a scan test set and CFA-BIST applied to a 6×9 FUB array.

for all 2×2 sub-arrays by the 835th test pattern.

Activation of all faults is necessary but not sufficient for 100% fault detection; one way to check sufficiency is to consider the probability of error masking within the BIST architecture. Recalling the definition of VH-bijectivity, it is guaranteed that an error will propagate to both the vertical and horizontal array outputs for some test that activates the fault. These erroneous signals will continue to be propagated by the non-faulty FUBs in the array during subsequent BIST cycles; the only way for an activated fault to escape detection is for these erroneous signals to converge on the faulty FUB and be masked before the final signature is observed. The probability of this form of masking is expected to be extremely low, and, moreover, it is expected to decrease exponentially as the size of the FUB array increases[1]. Thus the true fault coverage is expected to be equivalent to the level of fault activation achieved by the applied test set in the case of a single faulty FUB.

Empirical justification of the low likelihood for masking is investigated by simulating an implementation of a 6×9 FUB array using a commercial tool for both single-stuck line (SSL) and gate-level IP faults. The results are summarized in Table 3.2 for both the short-chain BIST architecture based on the 2×3 sub-array and a scan test set that achieves a similar level of fault detection. Note that 100% SSL and IP fault coverage is achieved by both the CFA-BIST and scan test for the given design.

[1]In a larger FUB array, a single defective FUB has less impact on the overall functionality of the array (which, being composed of bijective FUBs, will continue to propagate errors indefinitely). Furthermore, the state space for the FUB array increases exponentially with the number of array inputs/outputs; if it is assumed that an injected fault causes the FUB array to output random states, the probability of masking decreases exponentially with the size of the array state space. However, at this point the only means of verifying the probability of masking remains fault simulation of the design.

Furthermore, the CFA-BIST achieves this perfect fault coverage with reduced test time compared to the scan test. The number of test cycles for the scan test is determined by (a) the length of the scan chain, and (b) the number of test patterns that need to be applied to achieve the desired fault coverage on all FUB blocks in the array. The number of test cycles for scan test can be expressed as:

$$TC\_SCAN = (num\_patterns + 1) \times (sc\_length + 1) + k$$

where $num\_patterns$ is the number of test patterns (in this case $num\_patterns = 256$ for a super-exhaustive test set with activation properties equivalent to the CFA-BIST test cycle), sc_length is the length of the scan chain ($sc\_length = 30$), and k is a small constant number of test cycles added during scan-in of the first test pattern to ensure the scan chain is functioning properly ($k = 4$). However, the number of test cycles for CFA-BIST can be expressed as:

$$TC\_BIST = (seed\_patterns + 1) \times sc\_length + BIST\_cycles + k$$

where $seed\_patterns$ is the number of seed tessellation patterns that must be loaded/unloaded into the scan chain (in this case $seed\_patterns = 1$), sc_length is the length of the scan chain ($sc\_length = 30$), $BIST\_cycles$ is the number of BIST-mode clock cycles required to apply the test cycle (in this case $BIST\_cycles = 889$ to apply the complete test cycle), and k is a small constant number of test cycles added during the first BIST load/unload to ensure the scan chain is functioning properly ($k = 4$). Using these values, scan test requires a total of 7,971 test clock cycles while the CFA-BIST requires only 953 test clock cycles, representing a 88.0% reduction in the number of test clock cycles. Furthermore, this reduction is achieved without loss in test coverage for the SSL and gate-level IP fault models.

3.3 Summary

This chapter has presented strategies for testing the CM-LCV. Methods for the creation of tessellation test patterns have been described for the FUB array. These tessellation patterns exhaustively test every FUB in the FUB array (i.e., apply all FUB input patterns) with the minimum number of tests required, and can be constructed for FUB arrays of arbitrary size. These tessellation pattern creation methods can be extended using sub-arrays of FUBs to create super-exhaustive test sets of minimal size which exhaustively N-detect test every FUB in the FUB array. These methods are shown to be applicable to the heterogeneous FUB array if the FUB functions used are constrained to be the same along the tessellation diagonals of the array. Techniques for the application of exhaustive two-pattern tests to the pipelined FUB array are also demonstrated under both parallel and serial test application schemes. Finally, a BIST architecture that leverages the unique properties of the FUB array, namely the relationship between the inputs and outputs of the tessellation test patterns, is presented. Experiment results show a 88.0% reduction in the number of test cycles required with BIST for a reference design without loss in fault coverage.

Chapter 4

CM-LCV Diagnosis

This chapter describes a custom diagnosis methodology for the FUB array used in the Carnegie Mellon Logic Characterization Vehicle (CM-LCV). Various properties of the two-dimensional FUB array are exploited to improve diagnosis, particularly when multiple FUBs are defective. Section 4.1 examines in detail the phenomenon of error propagation in the FUB array. The proposed custom diagnosis methodology can be found in Section 4.2, while Section 4.3 reports the results from a series of experiments performed to evaluate its performance. Finally, Section 4.4 summarizes the work presented in this chapter.

4.1 Error Analysis

This section further describes the FUB array behavior relevant to diagnosis. Specifically, Section 4.1.1 elaborates on how VH-bijectivity can be used to localize a single defective FUB in a FUB array. Section 4.1.2 examines how the inverse of a VH-bijective function can be used to improve defective FUB localization. Section 4.1.3 extends these concepts to arrays with multiple defective FUBs, while Section 4.1.4 explores some of the complications that arise due to error masking. Finally, Section 4.1.5 examines the implications that the error bounds have on the accuracy of simulations of the FUB array.

4.1.1 Forward Bound

As noted at the end of Section 2.1, the location of the intersection of the errors observed on the array outputs for a failing pattern is closely related to the location(s) of defective FUBs in the array. Assuming there is only one defective FUB, Fig. 4.1 describes the three cases that can occur:

1. The defective FUB is horizontally adjacent to the error intersection (Fig. 4.1a).
2. The defective FUB is vertically adjacent to the error intersection (Fig. 4.1b).
3. The defective FUB is at the error intersection (Fig. 4.1c).

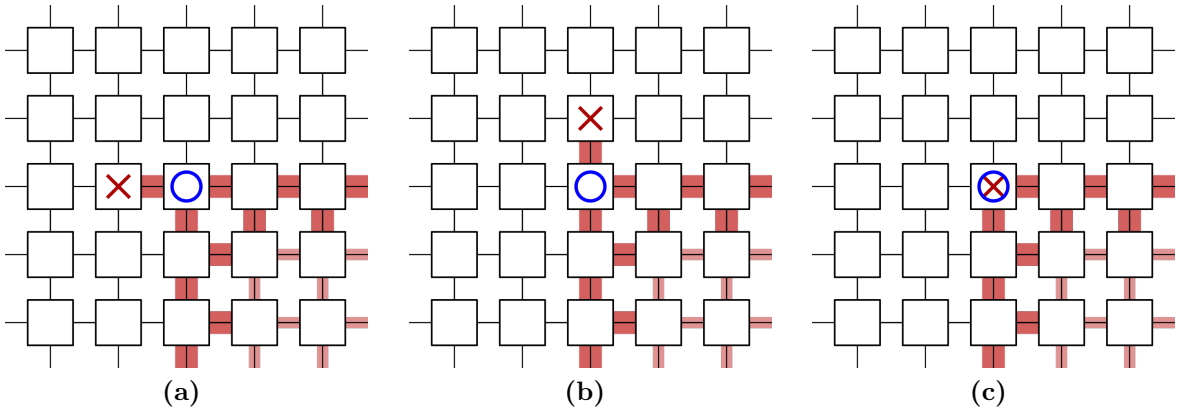


Figure 4.1: The three cases that exist for the relationship between the location of a single defective FUB (represented by the "×") and the location of the intersection of the errors observed on the array outputs (represented by the "○").

The array location derived from the errors observed at the array outputs is referred to as the *forward error intersection*. If only one defective FUB is present in the array, it must fall into one of the three cases described in Fig. 4.1; these three implicated FUB locations are referred to as the *forward bound*. Calculation of the forward bound is thus a very simple method that can be applied to any failing pattern to localize a single defect down to three candidate FUBs in a FUB array of arbitrary size.

4.1.2 Backward Bound

One property of bijective functions is that they are "reversible", that is, an inverse function is guaranteed to exist that maps the outputs of the bijective function to its inputs. The inverse of a VH-bijective function is also guaranteed to be VH-bijective, as proven in Theorem 4.1.

Theorem 4.1. *The inverse function* $F^{-1}(\hat{x}, \hat{y}) \rightarrow (x, y)$ *of a VH-bijective function* $F(x, y) \rightarrow (\hat{x}, \hat{y})$ *is itself VH-bijective.*

Proof. The property of bijectivity guarantees that the inverse function F^{-1} of bijective function F will be bijective. However, assume that the inverse function not VH-bijective, and let $F^{-1}(\hat{x_i}, \hat{y_i}) = (x_i, y_i)$ and $F^{-1}(\hat{x_j}, \hat{y_j}) = (x_j, y_j)$. This implies that the change in value on one of the inputs of F^{-1} does not result in a change in both of its output ports; thus, one of the following two cases must be true:

- There exists $\hat{x_i} = \hat{x_j}$ and $\hat{y_i} \neq \hat{y_j}$ such that either $x_i = x_j$ or $y_i = y_j$.
- There exists $\hat{x_i} \neq \hat{x_j}$ and $\hat{y_i} = \hat{y_j}$ such that either $x_i = x_j$ or $y_i = y_j$.

However, both of these cases violate the VH-bijectivity of the original function F. Consider the first case: if $x_i = x_j$, then $y_i \neq y_j$ (because F^{-1} is bijective), and thus by the VH-bijectivity of F both $\hat{x_i} \neq \hat{x_j}$ and $\hat{y_i} \neq \hat{y_j}$ must be true, resulting in a contradiction. Similar reasoning applies for the $y_i = y_j$ possibility in the first case, and all of the possibilities in the second case. Thus, the inverse function F^{-1} must be VH-bijective. □

This property allows for a failing test pattern and its response to be simulated *backwards* through the FUB array. This reverse simulation of a test pattern uses the observed response as the array input and uses the inverse FUB function to calculate the internal (fault-free) FUB array values and the array output. The internal FUB array values produced by this reverse simulation will be accurate until a defective FUB is reached, at which point differences will emerge between the reverse simulation and the actual FUB array values. Because the inverse FUB function is VH-bijective, these differences will propagate maximally along both

the columns and the rows, all the way to the FUB array inputs. Thus, the differences between the FUB array input values obtained from the reverse simulation and the applied test pattern can be used to generate a *backward bound* with the same properties as the forward bound.

This point is both subtle and powerful. Every failing test pattern and its observed response can be simulated both forwards and reverse, resulting in two bounds on the location of defective FUBs in the array. Fig. 4.2 is an example of how these bounds can be compared to localize a single defective FUB with even greater accuracy. In this example, the original test pattern and observed response can be used to construct the forward error intersection as shown in Fig. 4.2a. At this point the forward bound consists of three FUBs: the FUB at the forward error intersection (marked with the larger "○") and its upstream horizontal and vertical neighbors (marked with the smaller "○"). Performing the reverse simulation results in the backward error intersection as shown in Fig. 4.2b. The backward bound also consists of three FUBs: the FUB at the backward error intersection (marked with the larger "□") and its downstream horizontal and vertical neighbors (marked with the smaller "□"). Intersecting these two bounds results in only two possible locations for the defective FUB. In this way, the backward bound improves the resolution from three to two candidate FUBs.

4.1.3 Error Bounds and Multiple Defects

These error bounds are particularly useful in the case of multiple defects in the FUB array. There are two cases to consider: (i) multiple defects in a single FUB, and (ii) multiple defects in multiple FUBs. In the first case, the previous discussion on the forward and backward bounds remain valid as only a single FUB in the array is affected. For the second case, however, the bounds can be used to detect the presence of multiple defects. For a failing test pattern for an array with N defective FUBs, 1 to N FUBs may be active, where each active defective FUB produces an error. For these cases, the bounds can be used to differentiate between one active defective FUB and multiple active defective FUBs. Fig. 4.3 is an example

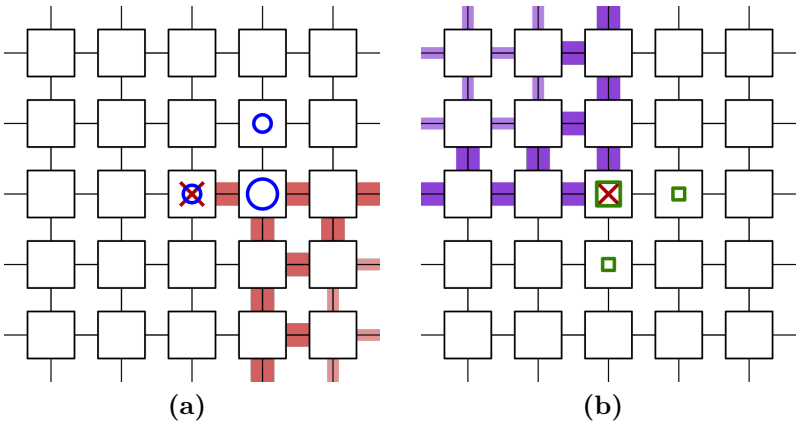


Figure 4.2: Forward (a) and backward (b) bounds can be used together to localize a single defective FUB in a FUB array with greater accuracy. The forward and backward error intersections (marked with the larger "□" and "○", respectively) and the appropriate adjacent FUBs (marked with the smaller "□" and "○", respectively) comprise the forward and backward bounds. In this example only two FUBs (the defecitve FUB marked with the "×" and its horizontal downstream neighbor) are implicated by both the forward and backward bounds.

of two different failing patterns for an array with two defective FUBs, each marked "×". In Fig. 4.3a, only one defective FUB is active, resulting in error intersections that are adjacent in the array. In contrast, Fig. 4.3b illustrates a test pattern with both defective FUBs active, resulting in bounds that implicate two non-overlapping regions of the array. A failing pattern caused by a single defective FUB can never result in an empty bounds intersection for simulations of the fault-free FUB array, as proven in Theorem 4.2.

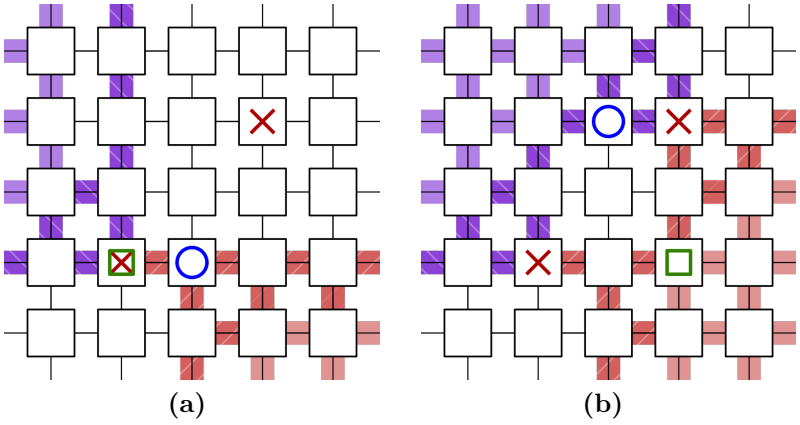


Figure 4.3: Demonstration of the forward and backward error intersections (marked "○" and "□", respectively) of two different failing test patterns for an array with two defective FUBs (marked "×"). The two examples shown represent (a) a test pattern with only one defective FUB active, and (b) a test pattern with both defective FUBs active.

Theorem 4.2. *A failing pattern caused by a single active defective FUB can never result in an empty error bounds intersection when compared to the simulation of the fault-free FUB array.*

Proof. Let a failing pattern be caused by a single active defective FUB at location (r, c) in the FUB array. This means that the defective FUB must produce an error on either $\hat{x}_{(r,c)}$, $\hat{y}_{(r,c)}$, or both for the applied test pattern, which will propagate through the remainder of the FUB array. The forward error intersection derived from the errors observed at the array outputs will thus be at one of the following locations: $(r, c), (r+1, c), (r, c+1)$ (see Theorem 2.2). In all three cases, the defective FUB location (r, c) will be included in the forward error bound (calculated as the forward error intersection and the two upstream adjacent FUB locations).

Next examine the backward error bound. During reverse simulation of the fault-free FUB array, the defective FUB at location (r, c) will result in a discrepancy at either $x_{(r,c)}$, $y_{(r,c)}$, or both. This discrepancy will propagate during reverse simulation through the remainder of the FUB array. The backward error intersection derived from the discrepancies observed at the array inputs will thus be at one of the following locations: $(r, c), (r-1, c), (r, c-1)$. In all three cases, the defective FUB location (r, c) will be included in the backward error bound (calculated as the backward error intersection and the two downstream adjacent FUB locations).

Thus, array location (r, c) will always be included in both the forward and error bound intersections derived from a failing pattern with a single active defective FUB and the simulation of the fault-free FUB array. □

4.1.4 Error Bounds and Error Masking

Uncertainties can be introduced into the error bounds if there is error masking, that is, when the effects of multiple defects interact in order to cancel out or otherwise obfuscate the propagation of errors through a design. There is a possibility of error masking in the FUB

array, though it is to some extent mitigated by the maximal propagation guaranteed by the VH-bijective FUBs. Figure 4.4 shows two examples of multiple defect interaction resulting in incorrect error bounds.

In Figure 4.4a, the errors from a single defective FUB are entirely masked by the presence of multiple defects downstream in the FUB array. The result is that the applied test pattern is incorrectly classified as a passing pattern. Because of the error propagation properties of the FUB array, the number of simultaneous defective FUBs required increases as the distance between the initial defective FUB and the masking defective FUBs increases.

In Figure 4.4b, the interaction between errors from three separate defective FUBs leads to partial error masking. The result is an incorrect error bound: the forward error intersection is displaced multiple rows and columns from the actual defective FUB locations.

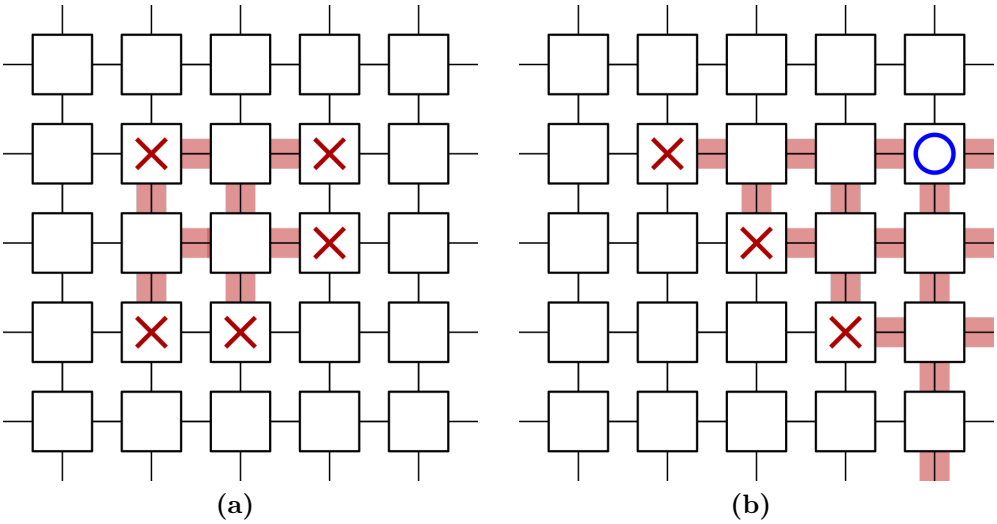


Figure 4.4: Examples of full (a) and partial (b) error masking due to the interaction of multiple defective FUBs (marked with "×") resulting in incorrect forward error intersection (marked with "○") in the FUB array. Error propagation within the FUB array is represented by the shaded FUB connections.

Note that these error masking cases are unavoidable in the FUB array, and indeed in any logic design. They can be mitigated to some extent in the FUB array by the use of additional test vectors. The probability that the masking behaviors exhibited in Figure 4.4 are consistent across a set of test vectors decreases as additional new test vectors are added. Of particular utility in this case are the super-exhaustive test sets generated using

sub-arrays, as discussed in Section 3.1.2, which apply the same input pattern to a given FUB multiple times with different neighborhood values, making it much less likely that a set of downstream defects can consistently mask the errors produced.

4.1.5 Error Bounds and Simulation Accuracy

A final aspect of the error bounds presented in this section regards their relationship with the accuracy of the FUB array state (i.e., the values at each FUB input and output port) obtained during simulation of the fault-free FUB array. Figure 4.5 shows the leading error propagation paths for the forward and reverse simulations with a single defective FUB. The crucial insight, originally stated in Section 4.1.2, is that the internal FUB array values produced by simulation of the fault-free FUB array will be accurate *until a defective FUB is reached*, at which point differences will emerge between the simulation and the actual values in the defective FUB array. This means that the input and output values for a targeted region in the array, in this case the single center FUB, can be extracted from the forward (input, Figure 4.5a) and reverse (output, Figure 4.5b) simulations. Furthermore, the error bounds can be used to determine which locations in the array can be accurately probed for a given simulation.

For multiple detect test patterns, the forward and reverse simulations no longer accurately provide these input and output values in all cases. Figure 4.6 shows two different arrangements of two defective FUBs for a multiple detect test pattern. For the first arrangement, with defective FUBs in an upstream/downstream relationship, the forward (Figure 4.6a) and reverse (Figure 4.6b) simulations can only provide the input values for the upstream defect and the output values of the downstream defect. In the second arrangement, with defective FUBs in a cross-stream relationship, the forward (Figure 4.6c) and reverse (Figure 4.6d) simulations can provide accurate input and output values for both defective FUB locations.

The key insight is that when defective FUBs are in an upstream/downstream relation-

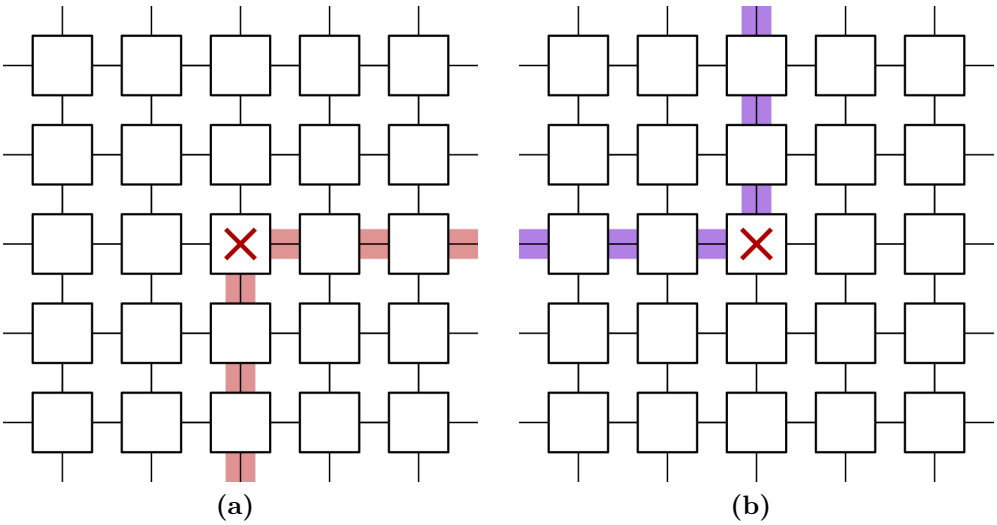


Figure 4.5: Demonstration of the forward (a) and reverse (b) simulation for a single defective FUB (marked with "×"). Only the leading error propagation paths are shown for each simulation direction (represented by the shaded FUB connections).

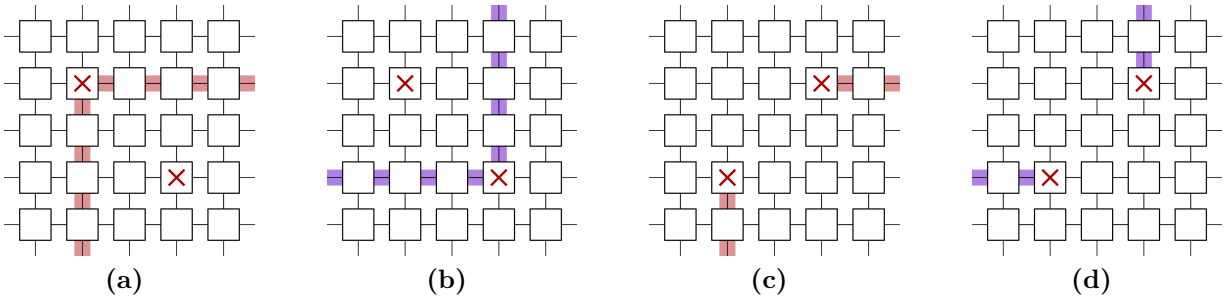


Figure 4.6: Demonstration of the forward (a, c) and reverse (b, d) simulation for a multiple detect test pattern caused by different arrangements of two defective FUBs (marked with "×"). Only the leading error propagation paths are shown for each simulation direction (represented by the shaded FUB connections).

ship with regards to the direction of simulation, the errors they produce compromise the accuracy of the simulation values at the locations of both defective FUBs. This issue could be circumvented if the FUB array could be simulated along the opposite diagonal directions. Adopting a cardinal direction notation (e.g., north, south, east, west), the forward and reverse simulation discussed up to this point represent the SE and NW simulation directions, and simulation in the SW and NE directions are being proposed as a solution for handling the troublesome defect interactions shown in Figures 4.6a and 4.6b. Not only is it possible to simulate the FUB array in these new directions, but also the corresponding FUB functions are VH-bijective for all of these directions. VH-bijectivity of the SE (original FUB function)

and NW (inverse FUB function) have already been established; Theorem 4.3 provides for the VH-bijectivity for the NE FUB function.

Theorem 4.3. *Given a VH-bijective function $F_{SE}(x, y) \to (\hat{x}, \hat{y})$, there exists a corresponding function $F_{NE}(x, \hat{y}) \to (\hat{x}, y)$ which is also VH-bijective.*

Proof. Let $F_{NE}(x_i, \hat{y}_i) = (\hat{x}_i, y_i)$ and $F_{NE}(x_j, \hat{y}_j) = (\hat{x}_j, y_j)$. First suppose that $F_{NE}(x, \hat{y}) \to (\hat{x}, y)$ is not bijective. Given that the domain and range of F_{NE} are identical ($F_{NE} : \mathbb{Z}_n^2 \to \mathbb{Z}_n^2$), this must imply that two different input values for F_{NE} result in the same output value: there exists $x_i \neq x_j$ or $\hat{y}_i \neq \hat{y}_j$ such that $\hat{x}_i = \hat{x}_j$ and $y_i = y_j$. Three relevant cases thus exist:

- If $x_i \neq x_j$, the VH-bijectivity of F_{SE} implies that both $\hat{x}_i \neq \hat{x}_j$ and $\hat{y}_i \neq \hat{y}_j$ (as $y_i = y_j$). However this violates the assumption that $\hat{x}_i = \hat{x}_j$.
- If $\hat{y}_i \neq \hat{y}_j$, the VH-bijectivity of $F_{NW} = F_{SE}^{-1}$ (which must exist and be VH-bijective by Theorem 4.1) implies that both $x_i \neq x_j$ and $y_i \neq y_j$. However this violates the assumption that $x_i = x_j$.
- If $x_i \neq x_j$ and $\hat{y}_i \neq \hat{y}_j$, both of the previous cases apply.

Thus, the VH-bijectivity of F_{SE} and F_{NW} require that F_{NE} be bijective.

Next suppose that F_{NE} is not VH-bijective. This implies that the change in value on one of the inputs of F_{NE} does not result in a change in both of its output ports; thus one of the following two cases must be true:

- There exists $x_i = x_j$ and $\hat{y}_i \neq \hat{y}_j$ such that either $\hat{x}_i = \hat{x}_j$ or $y_i = y_j$.
- There exists $x_i \neq x_j$ and $\hat{y}_i = \hat{y}_j$ such that either $\hat{x}_i = \hat{x}_j$ or $y_i = y_j$.

However, all of these options contradict the either the bijectivity or VH-bijectivity of F_{SE} or F_{NW}:

- If $x_i = x_j$, $\hat{y}_i \neq \hat{y}_j$ and $y_i = y_j$, F_{SE} cannot be bijective.

- If $x_i = x_j$, $\hat{y}_i \neq \hat{y}_j$ and $\hat{x}_i = \hat{x}_j$, F_{NW} cannot be VH-bijective.
- If $x_i \neq x_j$, $\hat{y}_i = \hat{y}_j$ and $\hat{x}_i = \hat{x}_j$, F_{NW} cannot be bijective.
- If $x_i \neq x_j$, $\hat{y}_i = \hat{y}_j$ and $y_i = y_j$, F_{SE} cannot be VH-bijective.

Thus the function F_{NE} is also VH-bijective.

□

Taken together, Theorems 4.1 and 4.3 guarantee that the FUB function is VH-bijective for all of the primary intercardinal directions (SE, NE, SW, NW). Thus, the FUB array can be simulated in any of these directions, and error bounds with all of the properties discussed in Section 4.1 can be obtained for each direction. However, the original SE direction remains unique: it alone reflects the reality of signal propagation in any physical implementation of the FUB array.

4.2 Hierarchical FUB Array Diagnosis

The FUB array properties discussed in Section 4.1 can be exploited to implement a hierarchical FUB array diagnosis (HFAD). Figure 4.7 shows the two hierarchical levels: In the first (Figure 4.7a), the FUB array is modeled at a purely logical level (i.e., no gate-level netlist or physical design; each FUB is a black box implementing the corresponding VH-bijective function), and the observed test response is mapped to a set of defective regions in the FUB array. In the second level (Figure 4.7b), the netlist for each defective region is extracted and diagnosed using conventional logic diagnosis techniques. Both levels of the proposed hierarchical FUB array diagnosis procedure are described in greater detail in the remainder of this section.

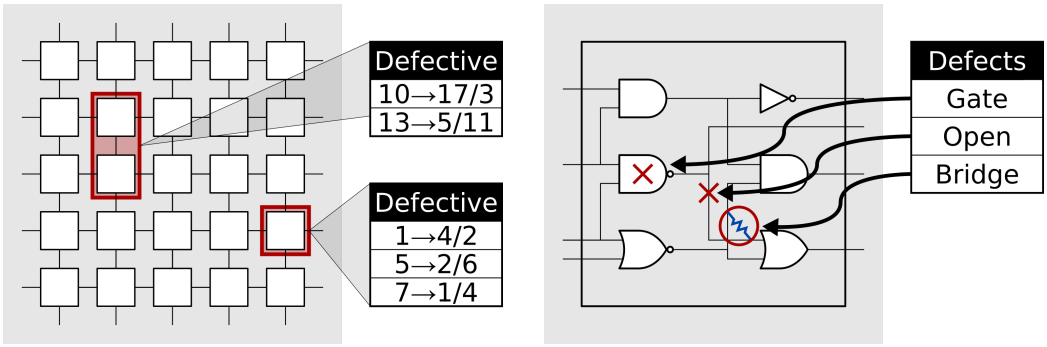


Figure 4.7: Hierarchical FUB array diagnosis consists of (a) an array level where the array behavior is mapped to smaller defective array regions, followed by (b) a netlist level where each defective array region behavior is analyzed with the corresponding netlist to derive the defect candidates.

4.2.1 Array Diagnosis

The array-level diagnosis begins with a fault-free logical model of the FUB array, with each FUB represented by a black box that implements the corresponding VH-bijective function. The observed test response is used to iteratively modify the functions used in this black-box representation, with the goal of creating a logical model of the array (termed the candidate array model) that perfectly matches the observed test response. This iterative modification of the candidate array model is guided by the forward and backward error bounds described in Section 4.1. The relationship between the two bounds is used to classify each test pattern into one of five categories:

1. **Match** - There is no difference between the simulation and the observed array response.

2. **Strong Single Detect** - The intersection of the forward and backward bounds results in exactly one FUB location in the array.

3. **Weak Single Detect** - The intersection of the forward and backward bounds results in exactly two FUBs.

4. **Multiple Detect** - The intersection of the forward and backward bounds is empty, and the forward error intersection is upstream of the backward error intersection.

5. **Inconsistent** - Any other relationship between the forward and backward bounds is classified as inconsistent. An example of this would be an empty bounds intersection with the forward error intersection downstream of the backward error intersection.

The test patterns that exhibit simpler defect behavior (i.e., the "strong single detect" and "weak single detect") are then preferentially used to determine the modifications to the candidate array model. The reason for this is that they are likely caused by a single defective FUB in the array, and can be effectively localized using the error bounds.

Once a change has been made to the candidate array model, the test set is re-simulated (in both forward and reverse directions), and the error bounds are re-calculated by comparing the observed test response to the *candidate array model simulation*. This represents a significant paradigm shift from the discussion of Section 4.1, which defined the error bounds as based on the comparison between the observed test response and the fault-free FUB array. These new error bounds no longer represent errors caused solely by defective FUBs; instead, they represent discrepancies between the candidate array model and the observed test response, whose source may be defective FUBs in the array or inaccurate changes to the candidate array model. In fact, it is no longer guaranteed that every FUB in the candidate array model is VH-bijective after any modifications have been made, which violates the conditions on Theorem 4.2, and may lead to inaccurate error bounds.

Despite these concerns, this approach is powerful in that it allows the iterative diagnosis to tease apart complex defect interactions. Figure 4.8 shows a simple example of how this can work with a test pattern that has three active defective FUBs (marked with "$\times$") in a small 5×5 FUB array. Initially (Figure 4.8a), the fault-free candidate array model differs at all three defective FUB locations, resulting in discrepancies during both forward (represented by the red shaded connections, only leading row/column shown) and reverse (represented by the purple shaded connections, only leading row/column shown) simulations. These discrepancies result in the forward (marked with "$\circ$") and backward (marked with "$\square$") error intersection. In this case the error intersections shown lead this test pattern to be

classified as a "multiple detect".

Now, suppose some other test is used to update the candidate array model in such a way that the errors produced by the lower left defective FUB (array location (4, 2)) are accurately modeled (represented by the additional red box), as shown in Figure 4.8b. The forward and reverse simulations of the candidate array model now result in different discrepancies (again indicated with the red and purple shading, respectively, with only the leading row/column shown) and a change in the error intersections. In this case the test pattern is again classified as a "multiple detect".

Next, again suppose some other test is used to update the candidate array model, this time in such a way that the errors produced by the top center defective FUB (array location (2, 3)) are accurately modeled (represented by the additional red box), as shown in Figure 4.8c. The forward and reverse simulations of the candidate array model again result in different discrepancies and a change in the error intersections. Now the test pattern is classified as a "strong single detect"; in fact, it can now be used to update the candidate array model for the final defective FUB (at array location (3, 4)). The end result, shown in Figure 4.8d, is a candidate array model with three modified FUBs that now matches exactly the observed test response for this test pattern.

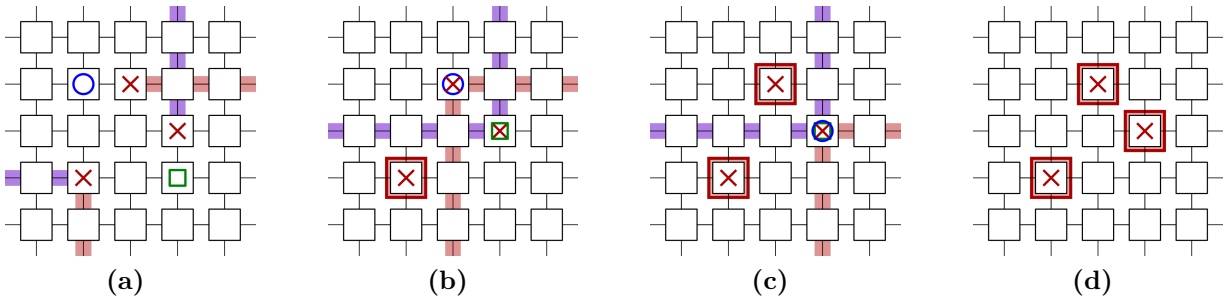


Figure 4.8: Demonstration of iterative updates (a-d) to the candidate array model for a test pattern with three active defective FUBs (marked with "×"). Modified FUBs in the candidate array model are indicated by the bold red outline. Additionally indicated are the forward and backward error intersections (marked "○" and "□", respectively) and the discrepancy propagation paths (shaded FUB connections, red and purple for forward and reverse, respectively) that determine the error intersections.

Several caveats should be noted at this point. First, there are cases where location ambiguity is unavoidable; for example, a "weak single detect" test pattern implicates two

FUB locations within the array. In these cases Theorem 3.2 can be leveraged to model multiple FUB locations with a single, larger black-box function in the candidate array model. Second, even "multiple detect" and "inconsistent" test patterns can be used to update the candidate array model, as will be discussed later in this section. Third, a new terminology of *defect region* is adopted for each black-box function that has been changed in the candidate array model.

An additional benefit of this iterative approach to diagnosis is that the progress of the diagnosis can be monitored in the changes in the test pattern classifications as the candidate array model is updated. Figure 4.9 depicts the expected changes in test pattern classification during a successful diagnosis process. In the ideal case, the candidate array model should move towards greater agreement with the observed array response; for example, "multiple detect" test patterns should progress to "single detect" or "match", and "single detect" test patterns should progress to "match". These ideal transitions are represented by the solid lines in Figure 4.9. However, in addition to these ideal transitions there are several other valid transitions in classification that can occur during a successful diagnosis process, represented by the dashed lines in Figure 4.9. Any other change is indicative of an error in diagnosis; an example of such a change would be a "match" pattern classified as a "multiple detect" after a change to the candidate array model.

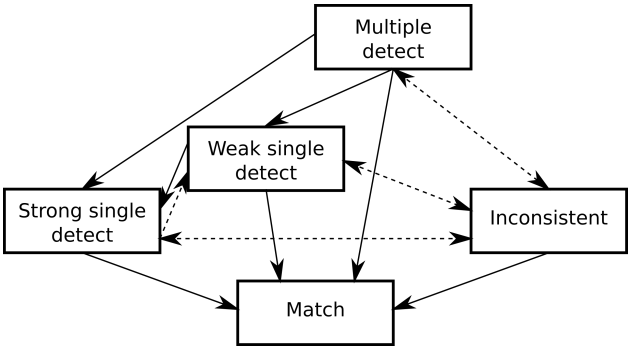


Figure 4.9: Representation of the ideal (solid) and valid (dashed) changes in test pattern classification during a successful diagnosis process.

Figure 4.10 presents the pseudocode for the iterative array diagnosis flow. Test patterns are classified according to their error bounds and then used to update a the candidate array model, with preference given to less complicated behaviors (e.g., "single detect" test patterns before "multiple detect" test patterns). The updated candidate array model is then simulated, and changes in test pattern classification are used to determine whether the changes should be accepted or rejected. Additionally, an ignore list of problematic test patterns is maintained and updated with any test patterns that lead to rejected changes to the candidate array model. Diagnosis ends when every test pattern is either classified as a "match" or has been added to the ignore list.

```
1   array_diagnosis(P, M)
2   // P = List of patterns
3   // M = Defect-free array model
4   begin
5
6     final_candidates = NULL // list of final diagnosis candidates
7     CM = copy of defect-free array model M
8     ignore_list = NULL // list of ignored patterns
9     current_candidates = list((CM, ignore_list))
10
11    // diagnosis loop
12    while(current_candidates is not empty)
13
14      CM, ignore_list = pop(current_candidates) // get current candidate model
15
16      // classify all non-ignored patterns
17      for each p in P - ignore_list
18        current_class(p) = classify_pattern(p, CM)
19
20      // select pattern to use for updating candidate model
21      if("Strong Single Detect" in current_class)
22        select p where current_class(p) == "Strong Single Detect"
23      else if("Weak Single Detect" in current_class)
24        select p where current_class(p) == "Weak Single Detect"
25      else if("Multiple Detect" in current_class)
26        select p where current_class(p) == "Multiple Detect"
27      else if ("Inconsistent" in current_class)
28        select p where current_class(p) == "Inconsistent"
29      endif
30
31      // update model based on selected pattern
32      update_model(CM, p)
33
34      // reclassify all non-ignored patterns
35      for each p in P - ignore_list
36        updated_class(p) = classify_pattern(p, CM)
37
38      // check for changes in classification
39      if(updated_class is all "Match")
40        add (CM, ignore_list) to final_candidates
41      else if(check_transitions(current_class, updated_class) indicates error)
42        undo changes to CM
43        add p used to update model to ignore_list
44        add (CM, ignore_list) to current_candidates
45      else
46        add (CM, ignore_list) to current_candidates
47      endif
48
49    end while
50
51    return final_candidates
52  end
```

Figure 4.10: Pseudocode for array-level diagnosis.

The key components of this diagnosis flow are the procedures used to update the candidate FUB array model based on the selected test pattern, which are explored in the remainder of this section.

Strong Single Detect

Test patterns in this category are the most straightforward, and thus are prioritized the highest in the array diagnosis flow. The input and output values for the implicated array location can be obtained using the forward and reverse simulations of the candidate array model. The FUB function at that array location is then updated to reflect the observed input and output values.

Weak Single Detect

Test patterns in this category strongly implicate two adjacent FUBs in the array. While it is possible to immediately group both of these implicated FUBs into a single defect region in the candidate array model, this may adversely impact the diagnostic resolution (defined as the number of FUBs included in all of the defect regions in the candidate array model). Instead, if the implicated FUBs intersect with any existing defect regions, it is assumed that those existing defect regions are responsible for the observed behavior. This is essentially a conservative heuristic that minimizes the number (and size) of defect regions in the candidate array model, resulting in improved diagnostic resolution. Based on this heuristic three straightforward cases now exist:

1. If only one of the implicated array locations belongs to a defect region, isolate the observed behavior from the forward and reverse simulations and update the functionality of that defect region.

2. If both of the implicated array locations belong to a single defect region, isolate the observed behavior from the forward and reverse simulations and update the functionality of that defect region.

3. If neither of the implicated array locations belongs to a defect region, treat both together a new defect region in the candidate array model. Isolate the observed behavior from the forward and reverse simulations and update the functionality for the new defect region.

A fourth case exists when both implicated array locations belong to different defect regions. In this case it is impossible to determine which defect region is responsible for the observed behavior. One appropriate course of action in this case would be to branch the array diagnosis process and create two new candidate array models, with each one updating a different defect region based on the current pattern. There are several downsides to this approach; it slows down array diagnosis considerably (as each new candidate array model must be simulated and diagnosed separately), and it creates ambiguity if multiple different candidate array models match the observed response. These issues can be avoided without loss of accuracy if adjacent defect regions in the candidate array model are always combined to create a single, larger defect region.

Multiple Detect

To begin there are two separate multiple detect cases: when the bounds share a row or column (termed a "1D" multiple detect) and when the bounds are in different rows and columns (termed a "2D" multiple detect). The 2D multiple detect will be examined first. Figure 4.11 shows examples of how two defects (Figure 4.11a-4.11b), three defects (Figure 4.11c-4.11f), and four defects (Figure 4.11g) can produce a 2D multiple detect signature.

Taken together, all of these options define eight separate regions in the FUB array, of which various combinations can produce the 2D multiple detect signature. Figure 4.12 depicts these eight regions for the 5×5 FUB array example used in Figure 4.11. The regions have been labeled with the cardinal directions (north, south, east, west); any combination of regions that, together, includes all four cardinal directions is capable of producing the 2D multiple detect signature observed in Figure 4.11.

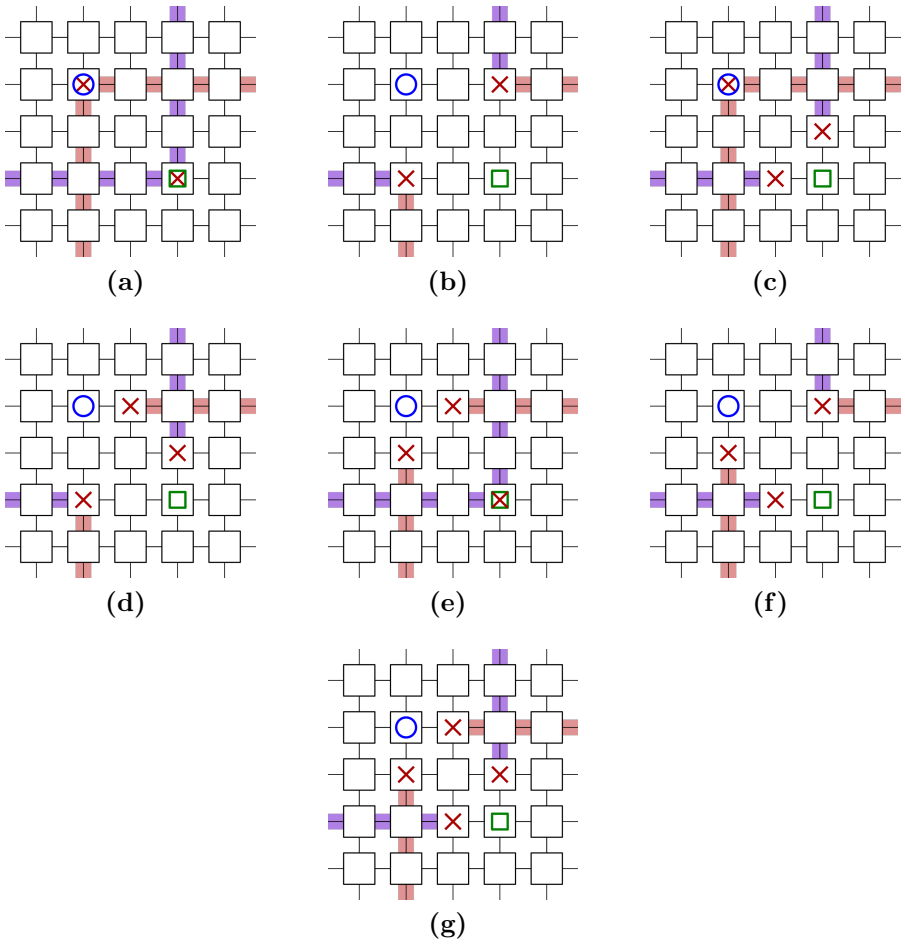


Figure 4.11: Demonstration of the ways in which two (a-b), three (c-f), and four (g) defective FUBs can result in a 2D multiple detect signature. The defective FUBs are marked with "×"; forward and backward error intersections are marked "○" and "□", respectively; and the discrepancy propagation paths that determine the bounds are shaded in the array.

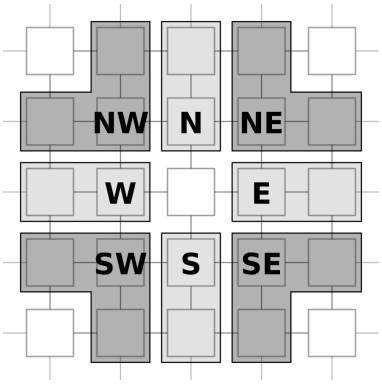


Figure 4.12: The eight distinct regions defined in the FUB array by a 2D multiple detect.

Exhaustively handling all of these possible cases in the array diagnosis would significantly add to its complexity without necessarily improving diagnosis performance. Instead, several

simplifying assumptions are made. The first involves a simple observation: only the corner regions of Figure 4.12 (i.e., NW, NE, SE, SW) can be accurately probed using the appropriate array simulations as discussed in Section 4.1.5. Particularly illustrative of this point is Figure 4.11g; the location and interaction of the four defective FUBs shown guarantees that the simulated array state will not accurately reflect the values in the very center of the FUB array regardless of the direction of simulation; furthermore, knowledge of this center array state is required in order to appropriately model the behavior of the four defect locations.

In addition to this insight, the previously-described conservative heuristic is again employed to filter down the number of options. However, there are now three separate cases that must be examined:

1. The first and most preferable case is if an existing defect region intersects with any one of the corner regions; in this case each such intersection can be used to update the candidate array model and diagnosis can proceed as usual.

2. If none of the first case exist, a second case is if an existing defect region intersects with one of the complementary regions of the array; for example, a known faulty region in the W region of Figure 4.12 provides indirect support for a defect in the complementary SE and NE corner regions. This results in two new candidate array models, corresponding to each corner region, which must be separately diagnosed moving forward.

3. If none of the previous two cases exist, the fallback case is to attempt to diagnose the current pattern to each of the four corner regions, resulting in four new candidate array models that must be separately diagnosed moving forward.

The combination of these three cases are sufficient to handle all 2D multiple detect test patterns. However, the 1D multiple detect still remains problematic. Figure 4.13 is an example of how two defective FUBs can result in a 1D multiple detect, in this case with a shared row in the FUB array. This figure hints at a key property of the 1D multiple detect: barring cases of error masking, there must be defective FUBs in the regions adjacent to each

of the two ends of the 1D multiple detect region. However, properly modeling these two end regions requires knowledge of the array state in the center of the 1D multiple detect region, which again cannot be determined using simulations in the four diagonal directions.

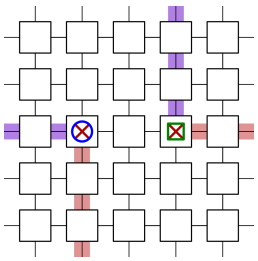


Figure 4.13: Demonstration of how two defective FUBs can result in a 1D multiple detect signature. The defective FUBs are marked with "×"; forward and backward bounds are marked "○" and "□", respectively; and the error propagation paths that determine the error bounds are shaded in the array.

This issue can be overcome by again leveraging the properties of the VH-bijective FUB function. In this case, it is possible to evaluate the FUB function in as NS or EW direction, that is, to input only the horizontal (vertical) values and determine the vertical (horizontal) values. Not only do these NS and EW functions exist for the VH-bijective FUB function, they are themselves VH-bijective[1]. In this way the forward (SE) and reverse (NW) simulations can be used to determine the array state up to the edges of the center of the 1D multiple detect region, and then the appropriate NS or EW function can be used to "squeeze" this region to determine the values needed to update the defect regions at the two ends of the 1D multiple detect region.

The only case where this approach is not viable is when the two ends of the 1D multiple detect region are adjacent. However, it has been previously established in the discussion on "weak single detect" test patterns that an appropriate course of action for adjacent defect regions is to combine them; this approach can also be applied here. In this way all 1D multiple detect test patterns can be handled by the array diagnosis.

[1]This property can be easily proven by adapting Theorem 4.3 with the desired input and output port pairs.

Inconsistent

The final classification that requires analysis is the "inconsistent" test patterns. This pattern classification can be caused by error masking (as noted in Section 4.1.4) or by simulation ambiguity caused by defect regions in the candidate array model. In the latter case, a change in the functionality of any defect region in the candidate array model can result in the loss of bijectivity. This means that the defect region function may no longer be reversible; in some cases, it may not have an input value that corresponds to the output value obtained from reverse simulation. In other cases, there may be multiple possible input values may correspond to the output value obtained from reverse simulation. The latter case is potentially problematic, as multiple choices at some defect region during reverse simulation can result in different discrepancies at the array inputs. Figure 4.14 shows the possible cases for a defect region consisting of a single defective FUB (marked "×") in a FUB array. Different options during reverse simulation can result in discrepancies at either both input ports (Figure 4.14a), only the horizontal input port (Figure 4.14b), only the vertical input port (Figure 4.14c), or neither input port (e.g., both values correct) (Figure 4.14d). Note that the backward error intersection (marked "□") is accurate in the first three cases; it remains on or adjacent to the site of the defective FUB. In the fourth case the backward error intersection is non-existent because the defective FUB behavior has been accurately captured by the FUB array model. Based on this analysis, the actual choice of input value during reverse simulation does not affect the accuracy of the backward bounds, though the existence of a choice that results in no errors at the array inputs indicates that the defect has been successfully modeled by the defect region.

Regardless of the cause for the "inconsistent" classification, the forward and backward error bounds are still often in close proximity in the array to the locations that are causing the error masking or simulation ambiguity. Thus, a simple heuristic is proposed: for any "inconsistent" test pattern, find a nearby defect region in the array model and attempt to update it with the behavior observed for that pattern. While this is an inexact heuristic, in

many cases it allows the array diagnosis to converge to a good solution despite the presence of these "inconsistent" test patterns.

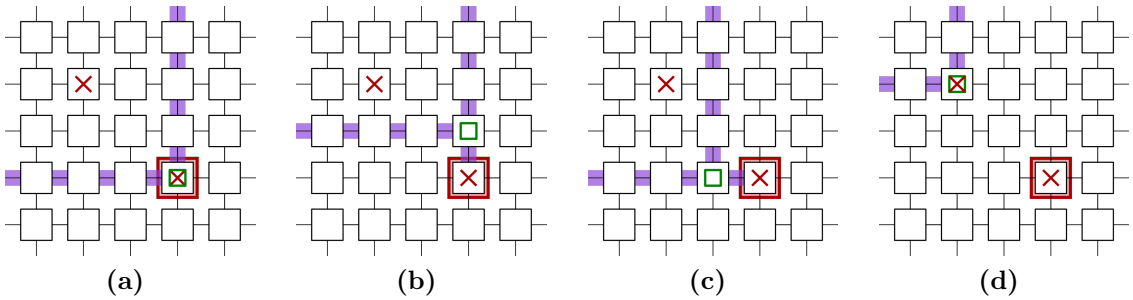


Figure 4.14: Demonstration of the four possible backward error intersections (marked with "□") during reverse simulation of a 5×5 candidate array model with a single defect region (represented by the additional red box) corresponding to a single defective FUB (marked "×"). During reverse simulation the defect region can emit discrepancies (represented by the shaded FUB connections) on both (a), only one (b-c), or neither (d) of its input ports.

4.2.2 Netlist Diagnosis

The candidate model produced by the array-level diagnosis described in Section 4.2.1 is effectively a black-box model of the FUB array with one or more regions (termed defect regions) that do not match the functionality of the original FUB array. For each of these defect regions, a netlist is formed by identifying the corresponding subcircuit from the original FUB array netlist. The truth table for each defect region (i.e., the black box functionality from the candidate model) is then used with the extracted netlist to perform conventional logic diagnosis. This logic diagnosis can be either one of the following:

- **Cause-Effect** - The first approach for netlist-level diagnosis is to use fault dictionaries for each for each FUB in the array. These fault dictionaries are constructed by simulating faults for each FUB netlist and recording the observed fault signatures. The fault signatures for the appropriate FUB can then be compared to the defect region truth table from the array-level diagnosis to derive the diagnosis candidates. Fault dictionaries are fast and simple to use, but typically incur a large memory overhead and have difficulties when multiple defects are present. The memory overhead is mitigated in

this case, due to the small size of the individual FUBs and the minimal number of tests for each FUB. An additional complication, however, is that the defect regions produced by array-level diagnosis may be larger than a single FUB, requiring the construction of additional multi-FUB dictionaries.

- **Effect-Cause** - The second approach is to use an effect-cause analysis on each implicated FUB. The faulty behaviors produced by array-level diagnosis can be interpreted as input and output values for each defect region netlist, which can then be processed by a diagnosis procedure to determine the final defect candidates. Because these defect regions are smaller than the overall array (typically encompassing on the order of one to four FUBs), and because the individual FUBs themselves are relatively small, this method allows for the employment of more sophisticated approaches without significant runtime overhead, resulting in improved diagnosis outcomes.

4.3 Experiment

This section presents the results of several experiments conducted to determine the efficacy of the proposed hierarchical FUB array diagnosis (HFAD). The array-level diagnosis described in Section 4.2.1 was implemented using the Python programming language [78]. Section 4.3.1 describes a simulation experiment where virtual failures in the FUB array are used to evaluate the performance of array-level diagnosis on its own. The Python implementation of the array-level diagnosis is paired with a commercial diagnosis tool to handle the netlist-level diagnosis, comprising a complete HFAD flow. Section 4.3.2 describes a simulated fault injection experiment used to compare the performance of the HFAD flow to the commercial tool alone. Finally, Section 4.3.3 presents the diagnosis results for data collected from FUB arrays manufactured in a 7nm technology.

4.3.1 Array-Level

The first results presented in this section are derived from a simulated array fault experiment designed to evaluate the diagnosis of the array-level only. A purely logical model of a 10×10 FUB array composed of 6-bit FUBs is created (i.e., no gate-level netlist or physical design; each FUB is a black box implementing the VH-bijective FUB function). Faults are injected into this FUB array model by introducing random changes to the FUB functions used at randomly selected array locations. Different combinations of the number of faulted FUB(s) (up to four) and the number of FUB function input patterns faulted (up to 32 out of the 64 input patterns for the 6-bit FUB functions) are simulated to create virtual fail logs, with 50 runs performed for each combination, and the virtual fail logs produced are analyzed using the array-level diagnosis implementation.

The results from the array-level diagnosis are evaluated in this context according to two metrics: perfect accuracy and FUB resolution. Perfect accuracy is achieved if the array-level diagnosis returns a candidate model that both matches the virtual fail log for every test pattern and encloses all of the FUBs with changed functionality in the defect regions. The FUB resolution is simply the number of FUBs included in all of the defect regions of the candidate model. Note that array-level diagnosis may return multiple candidate models; in such cases only the candidate model that matches the largest number of test patterns in the virtual fail log is examined.

Figure 4.15 shows the results of the array-level diagnosis when a tessellation test set (consisting of 64 test patterns) is applied. Figure 4.15a presents the percentage of simulations for which the diagnosis is perfectly accurate. The array-level diagnosis returns perfectly accurate results 100% of the time when only a single faulted FUB is present in the array. For two or more faulted FUBs, increasing the complexity, either through additional faulted FUBs or increased fault severity (i.e., number of input patterns affected), results in a loss of diagnosis accuracy. The one exception is that additional changed input patterns can actually result in improved accuracy when the number of affected input patterns is very

low, as observed in Figure 4.15a in the range of one to four faulty input patterns. This can be attributed to the fact that additional faulty input patterns give the array-level diagnosis more opportunities to precisely pin down the location of the faulty FUBs. To give an example, a known issue with the current array diagnosis implementation is in handling adjacent faulty FUBs. If only one adjacent FUB is precisely observed (i.e., a "strong single detect" pattern exists for only one of the injected faulty FUBs), the array-level diagnosis will create a defect region containing that FUB and, because of the conservative heuristic of attributing defective behavior to known defect regions whenever possible, will attempt to attribute all of the observed behavior to that single defect region. Depending on the type of interaction between the adjacent faulted FUB functions, this can cause the array diagnosis routine to fail to converge on a solution. This specific issue is expected to be addressed by continued improvements to the array-level diagnosis implementation.

Figure 4.15b presents the average FUB resolution for all of the perfectly accurate diagnosis results. The average FUB resolution is closer to the actual number of faulted FUBs when fewer faulted FUBs are injected into the array. An interesting effect is again observed when there are less than four faulted input patterns. In this case the average FUB resolution is higher (worse) when there are relatively few faulted input patterns. This is attributed to the higher likelihood that a faulted FUB is only detected by "weak single detect" test patterns when only a few input patterns in the FUB function have been faulted. If the location of the faulted FUB is never precisely pinned down by a "strong single detect" test pattern, the array-level diagnosis resorts to creating a larger defect region (in this case two adjacent FUBs), resulting in an increase in the FUB resolution. As the number of faulted input patterns increases, the likelihood that all of the randomly-selected faulted input patterns result in only "weak single detect" test patterns in the FUB array decreases, leading to improved FUB resolution.

Figure 4.16 shows the effect that a larger test set, in this case a super-exhaustive test set consisting of 512 test patterns, has on the array-level diagnosis performance. The same

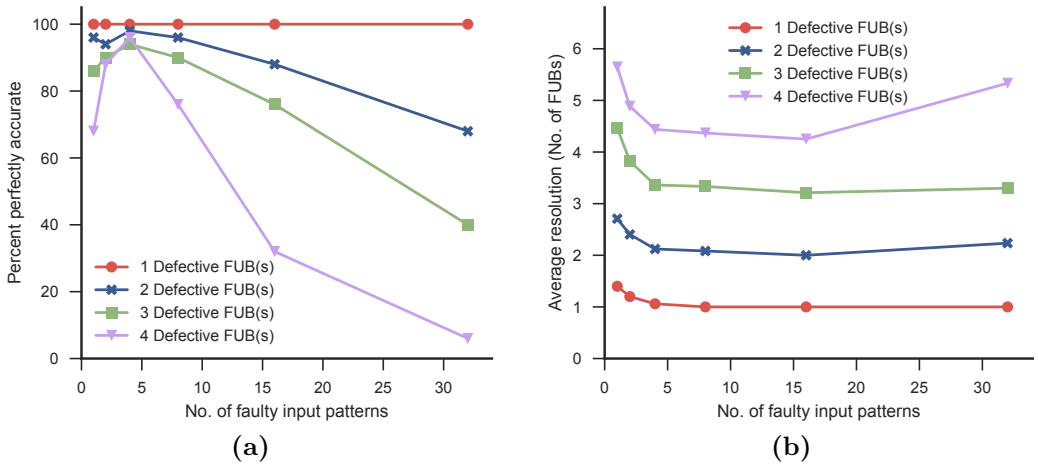


Figure 4.15: (a) Percentage of perfectly accurate diagnoses and (b) the average FUB resolution for perfectly accurate diagnoses for a simulated 10×10 FUB array. Faulted FUBs were injected in varying numbers and fault severity, expressed as the number of faulted input patterns per faulted FUB. A total of 50 simulations were performed at each data point, and the FUB arrays were tested with a 64 pattern tessellation test set.

10×10 FUB array and faulted FUBs used for Figure 4.15 are simulated and diagnosed using this expanded test set. The percentage of perfectly accurate diagnosis runs, shown in Figure 4.16a, shows significant improvement with the expanded test set for all of the multiple faulty FUB injections. The average FUB resolution, shown in Figure 4.16b, shows less of a change; the main exception is the improvement in resolution for four faulted FUBs, though it is worth noting that number of perfect diagnoses using the smaller test set for these cases is very small in the original test set data.

Taken together, these experiments strongly support the assertion that the array-level diagnosis is capable of handling multiple defects in the FUB array. This is particularly true when the larger test set is applied, with perfect diagnostic accuracy in over 76% of the runs with up to four faulted FUBs. Furthermore, perfect diagnostic accuracy is a very conservative metric; even in cases where the diagnosis output is not perfect, it may still accurately capture the behavior of several (but not all) of the faulted FUBs. It is conjectured that it is possible to diagnose up to two faulted FUBs in the array with perfect accuracy every time; the slight losses for two faulty FUBs demonstrated in Figures 4.15 and 4.16 are thus attributed to issues in implementation, and are a target for improvement in future work.

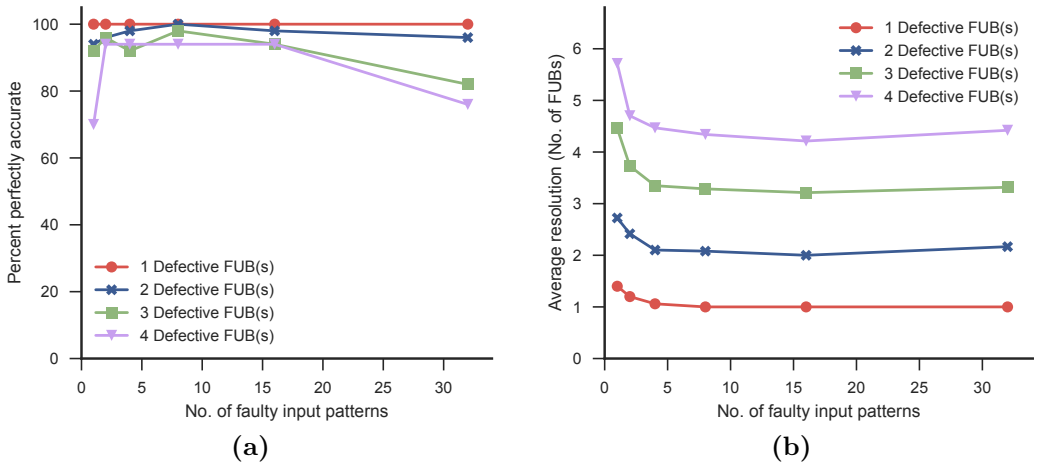


Figure 4.16: (a) Percentage of perfectly accurate diagnoses and (b) the average FUB resolution for perfectly accurate diagnoses for a simulated 10×10 FUB array. Faulted FUBs were injected in varying numbers and fault severity, expressed as the number of faulted input patterns per faulted FUB. A total of 50 simulations were performed at each data point, and the FUB arrays were tested with a 512 pattern super-exhaustive test set.

4.3.2 Netlist Level

The second experiment is a simulated fault-injection experiment using a FUB array created for verifying a commercial standard-cell library. The design contains a total of 8,308 standard cells, organized into 144 individual FUBs that are arranged into a 12×12 FUB array, where each FUB implements the same 6-bit VH-bijective function. A super-exhaustive test set consisting of 512 test patterns is constructed for this FUB array. Two faults are simultaneously injected into this array and simulated to create a virtual fail log for diagnosis. The injected faults are randomly-selected input pattern faults [29] on two different standard cells in the design. A total of 2,999 virtual fail logs are generated and analyzed using both the HFAD flow (i.e., the Python implementation of the array-level diagnosis combined with the commercial tool for the netlist-level diagnosis) and the standalone commercial diagnosis tool. For both the HFAD flow and commercial diagnosis, only the diagnosis candidates with the top score (as reported by the commercial diagnosis tool) are kept for each suspected defect.

Before presenting the results of the simulated fault-injection experiment, it is helpful to examine what an ideal diagnosis result would be in this context. Diagnosis is again evaluated

based on two criteria: resolution, which is defined here as the number of defect candidates reported, and accuracy, which is defined here as whether those defect candidates subsume the actual injected fault(s). Because two faults are injected into the array for each virtual fail log, an ideal diagnosis result should have exactly two defect candidates, with each defect candidate corresponding to one of the injected faults. A resolution lower than two cannot be perfectly accurate for both injected faults; a resolution greater than two indicates a loss of precision.

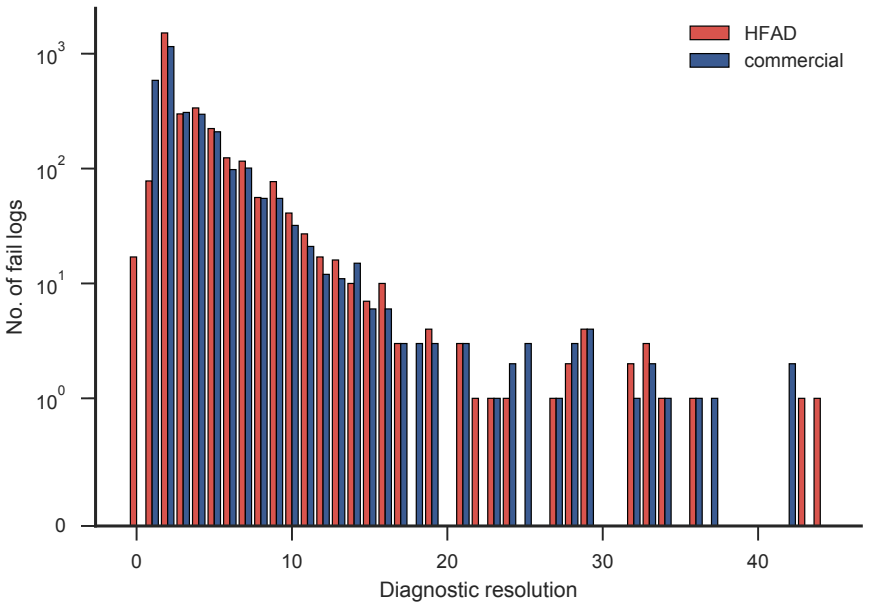


Figure 4.17: Diagnostic resolution for both the HFAD flow and custom diagnosis tool (vertical axis is logarithmic).

Fig. 4.17 is a histogram of the diagnostic resolution for the HFAD flow and commercial diagnosis. Note that the vertical axis is logarithmic to better represent the range of the two distributions. Commercial diagnosis has a minimum diagnostic resolution of 1, a maximum of 42, and an average of 3.48. The HFAD flow has a minimum diagnostic resolution of 0, a maximum of 44, and an average of 3.82. While on average the diagnostic resolution of the commercial tool is better, the HFAD flow does produce more outcomes with the expected ideal diagnostic resolution of 2.

However, viewing the distribution of diagnostic resolutions in this way may be misleading

HFAD accuracy	Commercial accuracy	Count	HFAD resolution			Commercial resolution		
			minimum	mean	maximum	minimum	mean	maximum
100%	100%	2,058	2	3.93	36	2	4.01	37
100%	50%	530	2	3.65	44	1	2.21	42
100%	0%	204	2	4.92	28	1	2.16	25
50%	100%	84	1	2.48	10	2	3.46	25
50%	50%	66	1	2.44	28	1	1.79	28
50%	0%	13	1	2.00	5	1	1.46	4
0%	100%	8	0	0.50	2	2	3.50	7
0%	50%	13	0	0.31	1	1	3.08	10
0%	0%	23	0	1.30	4	1	1.65	4

Table 4.1: Diagnostic outcome comparison for HFAD and commercial diagnosis.

without taking accuracy into account. Table 4.1 presents a comparison of the diagnosis outcomes broken down by the accuracy of the results. Given that the faults are injected into individual standard cells in the array, a diagnosis candidate is defined to be accurate if it includes (a) the standard cell or (b) a net that is connected to the standard cell used for fault injection. Diagnostic accuracy can thus be either 100% (candidates include the two injected sites), 50% (candidates include one of the two injected sites), or 0% (candidates include neither injected site) for each fault-injection simulation. The first two columns of Table 4.1 report this accuracy for the HFAD and commercial diagnosis, with the third column showing the number of simulated fail logs for each accuracy combination. Columns four through six report the minimum, mean, and maximum number of diagnosis candidates (i.e., the diagnosis resolution) for the HFAD flow, while the remaining columns report the same for the commercial diagnosis tool.

Table 4.1 indicates that the HFAD flow achieves ideal accuracy for 93.1% of the simulated fail logs (2,792 out of 2,999). Commercial diagnosis, on the other hand, achieves perfect accuracy for only 71.7% of the simulated fail logs (2,150 out of 2,999). Furthermore, for the 2,058 simulated fail logs where both tools are perfectly accurate, the HFAD produces an improved average diagnostic resolution (3.93 compared to 4.01). However, the HFAD does not offer a strict accuracy improvement over the commercial tool; there remain a total of 105 virtual fail logs where the commercial tool improves upon the HFAD accuracy, with perfect commercial accuracy for 92 of those virtual fail logs. Examination of these 105 virtual fail logs reveals that at least one of the following three conditions are present in each one:

- Faults are injected into FUBs in the same row or column of the array in 46 simulated fail logs (43.8%).
- One or more faults are injected into FUBs on the boundary of the array in 69 simulated fail logs (65.7%).
- Excluding the previous two cases, faults are injected into FUBs in adjacent rows or columns of the array in the remaining 10 simulated fail logs (9.1%).

All of these conditions represent edge cases for the array-level diagnosis implementation. Faults injected into FUBs in the same/adjacent row or column in the FUB array can lead to error masking along the error propagation paths used to calculate the error bounds (as described in Section 4.1). This potential for incorrect error bounds and the increased complexity of handling '1D' multiple detects (Section 4.2.1) are likely contributors to the poor performance of the HFAD flow. The remaining cases with faults injected into FUBs on the boundary of the array require special handling in the array-level diagnosis implementation; the poor accuracy of the HFAD flow in this case may be attributable to programming errors. Overall, while these cases remain a target for improvement, the significant increase in diagnosed fail logs with perfect accuracy (from 71.7% to 93.1%) demonstrates the value of the HFAD flow.

4.3.3 Silicon

In addition to the simulated experiments presented in Sections 4.3.1 and 4.3.2, real silicon test data is obtained for a test chip manufactured in a 14nm process under development by an industry partner. This test chip included 64 different 5×5 FUB arrays with 6-bit FUBs, which were implemented as macro blocks in a larger test chip architecture. The CMU team supplied the logical descriptions of the 64 arrays as well as a 64-pattern exhaustive test set. The industry partner then integrated these arrays into the larger test chip architecture,

Diagnosis	Total runtime (sec)	Average runtime (sec)
HFAD array-level	19,910	14.48
HFAD netlist-level	19.15	0.1
HFAD overall	19,929.15	14.49
Commercial	1,535.15	1.12

Table 4.2: Runtime for HFAD and commercial diagnosis.

created the physical design, and manufactured the test chip. A total of 1,503 logs of failing FUB arrays are provided to the CMU team for diagnosis.

Of these 1,503 fail logs, 128 exhibit a defect that impacted the test architecture used to access the FUB arrays. The remaining 1,375 are diagnosed using both the HFAD flow and a standard commercial diagnosis tool. The results of both are filtered to include only the top diagnosis candidates for each suspected defect. Figure 4.18 presents a histogram of the diagnostic resolutions, defined as the number of diagnosis candidates, obtained for each fail log. Overall the diagnostic resolution distributions are very similar; differences include a number of fail logs for which the HFAD flow failed to produce any candidates, and more outliers with high diagnostic resolution for the commercial tool.

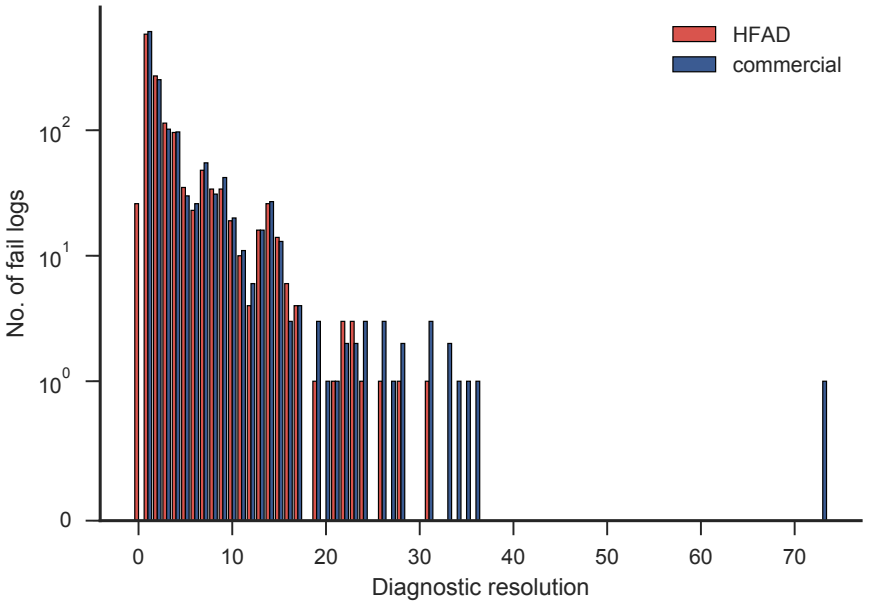


Figure 4.18: Diagnostic resolution for both HFAD flow and the commercial diagnosis (vertical axis is logarithmic).

The runtimes for the HFAD and commercial diagnosis are presented in Table 4.2. Both the HFAD and commercial diagnosis were performed on a machine with 64 CPU cores running at 2.2GHz with 1TB of RAM, and a timeout of five minutes was applied to each diagnosis run. The HFAD array-level diagnosis alone is an order of magnitude slower than the commercial tool, but the HFAD netlist-level diagnosis is very fast (less than a second per fail log), resulting in an overall average runtime of 14.48 seconds per fail log compared to 1.12 seconds per fail log for the commercial tool. This difference is unsurprising given the complexity of the array-level diagnosis (with repeated simulations of the FUB array in multiple directions). While an effective slowdown of 12.9x is manageable for this data set (total runtime remains under six hours), improvements to the runtime of the HFAD flow represents a high-priority target for future optimization.

Unlike in the previous two experiments, the ground truth is not known for this data set (i.e., the nature and location of the defect(s) present for each fail log), making it impossible to directly evaluate the diagnosis accuracy. Instead, two techniques are utilized: first, the forward and backward error bounds (as described in Section 4.1) can be calculated for each test pattern and compared to the array location of the diagnosis candidates. If diagnosis candidates exist in FUBs at array locations that are never implicated by the error bounds they are likely to be incorrect. Alternatively, if the error bounds implicate an array location that has no diagnosis candidates, it is likely that the diagnosis missed a defect. A second approach is to compare the defect candidates produced by the HFAD flow and the commercial tool. Consensus between the two tools indicates higher confidence in the diagnosis accuracy, while disagreements in the number of suspected defects or their locations are indicative of inaccuracy by one (or both) of the tools. The latter approach is utilized in Table 4.3, which shows the diagnosis outcomes for the two tools by the number of suspected defects reported by each. The first two columns report the number of defects reported for each HFAD and commercial diagnosis, with the third column showing the number of fail logs with each combination of reported defects. Columns four through six report the minimum, mean, and

No. HFAD defects	No. commercial defects	Count	HFAD resolution			Commercial resolution		
			minimum	mean	maximum	minimum	mean	maximum
0	2	3	0	0.00	0	7	7.67	9
0	3	6	0	0.00	0	10	17.8	36
0	4	7	0	0.00	0	9	20.2	35
0	5	5	0	0.00	0	19	27.5	33
0	6	2	0	0.00	0	17	25.5	34
0	7	3	0	0.00	0	24	42.7	73
1	1	1,113	1	2.98	31	1	2.97	31
1	3	1	1	1.00	1	7	7.00	7
2	1	44	2	5.09	17	1	2.91	13
2	2	126	2	5.02	24	2	5.14	24
2	3	7	2	4.57	15	3	6.86	15
3	1	2	3	3.00	3	1	1.00	1
3	2	5	3	6.60	16	2	6.20	15
3	3	25	3	6.32	23	3	6.20	23
3	4	4	3	5.00	7	7	7.75	9
3	7	1	3	3.00	3	11	11.0	11
4	1	1	12	12.0	12	9	9.00	9
4	3	5	5	9.20	16	4	7.60	12
4	4	5	4	10.8	28	4	11.8	28
5	2	1	9	9.00	9	13	13.0	13
5	3	2	13	14.0	15	9	17.5	26
5	4	2	5	6.00	7	4	11.5	19
6	2	2	13	13.0	13	3	6.00	9
6	4	1	9	9.00	9	6	6.00	6
7	3	1	16	16.0	16	9	9.00	9
9	4	1	21	21.0	21	14	14.0	14

Table 4.3: Diagnostic outcome comparison for HFAD and commercial diagnosis.

maximum number of diagnosis candidates (i.e., the diagnosis resolution) for the HFAD flow, while the remaining columns report the same for the commercial diagnosis tool.

Both the HFAD flow and the commercial diagnosis report similar results for a majority of the fail logs: a total of 1,269 fail logs result in agreement in the number of suspected defects, with minimal differences in the mean diagnostic resolution. The remaining 106 fail logs have been divided into three categories for further analysis: cases where HFAD flow produces no candidates, cases where commercial diagnosis produces more suspected defects, and cases where the HFAD flow produces more suspected defects. The remainder of this section will discuss each of these categories individually.

HFAD Zero Candidates

The HFAD flow failed to produce any diagnosis candidates for 26 fail logs (represented in the first section of Table 4.3). In each of these cases the lack of diagnosis results is due to a lack of candidates produced during the array-level of the HFAD flow. Closer examination of

these 26 fail logs reveals that 18 (69.2%) have one or more inconsistent test pattern on first classification, indicating the presence of multiple defects with error masking. In contrast, only 75 of the 1,375 fail logs diagnosed (5.5%) have one or more inconsistent test pattern. This suggests that error masking remains problematic for the proposed array-level diagnosis. Of the remaining eight fail logs with no inconsistent patterns, commercial diagnosis indicates between three and seven suspected defects for each, with relatively poor resolution (nine diagnosis candidates or more for each). Overall, these 26 fail logs support the conclusion that the HFAD flow fails only when the diagnosis problem is complex (multiple defects and/or error masking), and represent a target for improvements to the implementation of the array-level diagnosis used in the HFAD flow.

HFAD Fewer Suspected Defects

The HFAD flow produces fewer suspected defects than the commercial tool for 13 fail logs. The diagnosis problems posed by these fail logs are again relatively difficult: both tools agree on the existence two or more suspected defects for 12 of the 13 fail logs, and four fail logs include one or more inconsistent patterns (indicative of error masking). Nevertheless, both tools are fairly consistent in the diagnosis candidates produced, with 11 fail logs having at least one common diagnosis candidate.

As discussed earlier in this section, the array locations implicated by error bounds can be compared to the locations of the diagnosis candidates produced. Mismatches between these implicated locations and the diagnosis candidates can indicate either false diagnosis candidates (which cannot be present due to the FUB array error propagation properties) or defects that have been missed by diagnosis. Performing this analysis on the seven fail logs without error masking finds that, for one fail log, the commercial tool includes a suspected defect at a location that is never implicated by the error bounds. This strongly suggests that the commercial tool produced a false candidate in this case. Analysis of the remaining six fail logs is inconclusive; the error bounds do not exclude the additional suspected defects

that the commercial tool found relative to the HFAD flow. While the twelve fail logs with error masking or inconclusive error bounds represent potentially inaccurate diagnosis (missed defects) for the HFAD flow, they comprise a small fraction (0.9%) of the total population of fail logs.

HFAD More Suspected Defects

The HFAD flow produces more suspected defects than the commercial tool for 67 fail logs. However, it is unclear if these additional suspected defects represent are true defects that commercial tool missed, or false diagnosis candidates created by the HFAD flow. The error bound analysis is again performed to help answer this question. Of the 67 fail logs, 16 include one or more inconsistent pattern (indicative of error masking), and are thus excluded. Of the remaining 41 fail logs, for 40 fail logs the error bounds implicate array locations that are unaccounted for by the commercial diagnosis result. The HFAD flow, on the other hand, includes additional suspected defects that fully explain all implicated regions for all fail logs. This analysis suggests that the commercial tool failed to produce any diagnosis candidates for an existing defect in 40 of the 1,375 diagnosed fail logs (2.9%), and that in each of these cases the HFAD flow did produce one or more diagnosis candidates for this missing defect. Several cases from the latter category have been examined and will be described in greater detail.

Case Study I. This first case study is a fail log where the commercial and custom diagnosis tools disagree on the number of suspected defects, with the commercial tool returning only one while HFAD indicated the presence of two. Tables 4.4 and 4.5 describe the abridged diagnosis results for the commercial diagnosis and HFAD, respectively. Each row in the table corresponds to a single diagnosis candidate. The first and second columns provide the defect and candidate identifier; multiple diagnosis candidates can correspond to the same defect, as indicated by a shared defect ID. The third column indicates the candidate's score as provided

by the commercial diagnosis tool, with a higher score representing a better match between the candidate and the fail log behavior. The fourth column indicates the type of the candidate, which may match a fault model (e.g., stuck-at, bridge, etc.) or some other behavior specified by the diagnosis tool. The sixth column provides additional information on the behavior of the candidate (for example, a "STUCK" candidate with value "0" corresponds to a stuck-at-0 fault). Finally, the seventh column provides the netlist location, expressed as a pin path, for each candidate.

Defect	Candidate	Diagnosis score	Type	Value	Pin
1	1	100	INDETERMINATE	0	U_0_0_U218:o1
1	2	100	EQUIVALENT	1	U_0_0_U218:c
1	3	100	EQUIVALENT	1	U_0_0_U165:o1
1	4	100	EQUIVALENT	1	U_0_0_U218:a
1	5	100	EQUIVALENT	1	U_0_0_U218:b
1	6	100	EQUIVALENT	1	U_0_0_U211:o1
1	7	100	EQUIVALENT	0	U_0_0_U211:b
1	8	100	EQUIVALENT	0	U_0_0_U211:a
1	9	100	EQUIVALENT	0	U_0_0_U210:o1
1	10	100	EQUIVALENT	0	U_0_0_U207:o1

Table 4.4: Abridged commercial diagnosis result for Case Study I.

Suspected defect	Candidate	Diagnosis score	Type	Value	Pin
1	1	100	STUCK	1	U_0_0_U207:a
1	2	100	EQUIVALENT	1	U_0_0_U164:o1
1	3	100	STUCK	1	U_0_0_U207:b
1	4	100	EQUIVALENT	1	U_0_0_U206:o1
1	5	100	STUCK	0	U_0_0_U218:o1
1	6	100	EQUIVALENT	1	U_0_0_U218:c
1	7	100	EQUIVALENT	1	U_0_0_U165:o1
1	8	100	EQUIVALENT	1	U_0_0_U218:a
1	9	100	EQUIVALENT	1	U_0_0_U218:b
1	10	100	EQUIVALENT	1	U_0_0_U211:o1
1	11	100	EQUIVALENT	0	U_0_0_U211:b
1	12	100	EQUIVALENT	0	U_0_0_U211:a
1	13	100	EQUIVALENT	0	U_0_0_U210:o1
1	14	100	EQUIVALENT	0	U_0_0_U207:o1
2	1	100	CELL	0	U_2_2_U90:out0

Table 4.5: Abridged HFAD result for Case Study I.

The commercial diagnosis callout consists of a single defect candidate of type "INDETERMINATE", along with nine equivalent candidates. All of these candidates have a score of 100, indicating a high degree of confidence in the result. Additionally, all of these candidates are in the top left FUB in the FUB array (indicated by the "U_0_0" prefix on the pin pathnames). By itself, this would be an acceptable diagnosis outcome: high confidence in a single defect with moderate resolution.

The HFAD callout contradicts the commercial result. While it too includes a defect in the top left FUB, a second defect is found in the center FUB in the FUB array (indicated by the "U_2_2" prefix on the pin pathname for the second suspected defect candidate). Again the diagnostic scores are all 100, indicating a high degree of confidence in all of these results. Comparison of the candidates in the top left FUB shows that the custom result includes all of the same locations ("U_0_0_U218:c" and its nine equivalents), however the type of the candidate is changed from "INDETERMINATE" to stuck-at-0, and two additional stuck-at candidates (along with their equivalents) are added. This change in candidate type is significant; whereas stuck-at candidates have a simple and well-defined behavior, indeterminate candidates are much more flexible, making them more likely to match the observed fails but less helpful in characterizing the defect.

Most significant, however, is the fact that the commercial diagnosis missed the defect in the center of the FUB array. Analysis of the fail log using the array-level diagnosis shows that ten test patterns are classified as "Multiple detect", and a further two test patterns (classified as "Weak single detect") only detected the defect in the center FUB. These last two tests are especially troubling; given the error propagation properties of the FUB array, it is impossible for a defect in the top left FUB to match the errors produced by a defect in the center of the FUB array. This means that, not only did commercial diagnosis miss a high-confidence and meaningful defect candidate that HFAD found, it overstates confidence in the defect candidate that it did find.

Case Study II. This second case study is another fail log where the commercial and custom diagnosis tools disagree on the number of suspected defects, with the commercial tool returning two while custom indicated the presence of three. Tables 4.6 and 4.7 present the abridged diagnosis results for the commercial and custom tools, respectively. The column format remains the same as in Tables 4.4 and 4.5.

The commercial diagnosis callout consists of two suspected defects: one of type stuck-

Suspected defect	Candidate	Diagnosis score	Type	Value	Pin
1	1	98	STUCK	1	Yout[6]
1	2	98	EQUIVALENT	1	U_4_2_U89:o
2	1	96	STUCK	1	U_0_0_U109:o1
2	2	96	EQUIVALENT	0	U_0_0_U109:d
2	3	96	EQUIVALENT	0	U_0_0_U109:c
2	4	96	EQUIVALENT	0	U_0_0_U108:o1
2	5	96	EQUIVALENT	0	U_0_0_U109:a
2	6	96	EQUIVALENT	0	U_0_0_U109:b
2	7	96	EQUIVALENT	0	U_0_0_U105:o1
2	8	96	EQUIVALENT	0	U_0_0_U97:o1
2	9	96	EQUIVALENT	0	U_0_0_U99:o1
2	10	96	EQUIVALENT	1	U_0_0_U105:a
2	11	96	EQUIVALENT	1	U_0_0_U105:b
2	12	96	EQUIVALENT	1	U_0_0_U101:o1
2	13	96	EQUIVALENT	1	U_0_0_U104:o1

Table 4.6: Abridged commercial diagnosis result for Case Study II.

Suspected defect	Candidate	Diagnosis score	Type	Value	Pin
1	1	100	CELL	1	U_2_4_U71:out0
2	1	100	STUCK	1	U_0_0_U109:o1
2	2	100	EQUIVALENT	0	U_0_0_U109:d
2	3	100	EQUIVALENT	0	U_0_0_U109:c
2	4	100	EQUIVALENT	0	U_0_0_U108:o1
2	5	100	EQUIVALENT	0	U_0_0_U109:a
2	6	100	EQUIVALENT	0	U_0_0_U109:b
2	7	100	EQUIVALENT	0	U_0_0_U105:o1
2	8	100	EQUIVALENT	0	U_0_0_U97:o1
2	9	100	EQUIVALENT	0	U_0_0_U99:o1
2	10	100	EQUIVALENT	1	U_0_0_U105:a
2	11	100	EQUIVALENT	1	U_0_0_U105:b
2	12	100	EQUIVALENT	1	U_0_0_U101:o1
2	13	100	EQUIVALENT	1	U_0_0_U104:o1
3	1	100	STUCK	1	Yout[6]
3	2	100	EQUIVALENT	1	U_4_2_U89:o

Table 4.7: Abridged HFAD result for Case Study II.

at-1 (along with a single equivalent candidate) at the center of the vertical output edge of the FUB array (the "U_2_4" prefix indicates the center bottom FUB), and a second of type stuck-at-1 (along with its 12 equivalent candidates) in the top left corner of the FUB array (again indicated by the "U_0_0" prefix). While the scores for these candidates are high, at 98 and 96 respectively, they are not perfect. Overall this is again an acceptable diagnosis outcome, with two isolated suspected defects with good confidence and moderate resolution.

The custom diagnosis callout improves upon the commercial result by including an additional suspected defect at the horizontal output of the FUB array (the "U_2_4" prefix corresponds to the center far-right FUB). Including this third suspected defect raises the score of all of the candidates to 100, including the score for the defect candidates that the

commercial tool discovered. The result is a significantly better diagnosis outcome, with three isolated suspected defects with very high confidence, of which two have resolution of two or less.

4.4 Summary

This chapter has discussed the diagnosis properties of the CM-LCV. The unique error propagation properties achieved by using VH-bijective functions in the FUB array are leveraged to define bounds on the locations of defects. A hierarchical FUB array diagnosis (HFAD) flow, consisting of array-level and netlist-level diagnosis stages, is designed to take advantage of these error bounds. An implementation of this HFAD flow was developed and several experiments were performed to evaluate its performance. Fault simulations of a 10×10 FUB array indicate that the array-level HFAD diagnosis is well-suited for multiple defect diagnosis, with perfect diagnostic accuracy achieved in up to 76% of the runs when four FUBs were simultaneously faulted. The full HFAD flow implementation achieves perfect accuracy for 93.1% of the simulations of two injected faults performed on a 12×12 FUB array design, representing a significant improvement over state-of-the-art commercial diagnosis. This value of the custom diagnosis is further demonstrated in the analysis of fail logs from a series of 5×5 FUB arrays fabricated in a 14nm process, where the custom diagnosis flow was able to detect and characterize defects that the commercial tool missed for 40 out of the 1,375 diagnosed fail logs.

Chapter 5

Conclusions

Achieving acceptable yield as the semiconductor manufacturing process continues to increase in complexity presents a significant technical challenge. This dissertation has presented a new logic test chip, the Carnegie Mellon Logic Characterization Vehicle, to help meet this challenge. The CM-LCV is intended for use in the later stages of the technology development, and is designed to aid in the detection and mitigation of product-relevant defects before they can adversely impact product yield. The CM-LCV can be tested with high efficiency; methods for constructing minimal test sets with 100% fault coverage for a variety of fault models have been demonstrated, as well as a BIST architecture that reduces the number of test cycles by a factor of 88.0% in a reference design. The CM-LCV also offers improved diagnosis properties: included in this dissertation is a hierarchical FUB array diagnosis (HFAD) methodology capable of accurately diagnosing 93.1% of simulations of two injected defects, an improvement on the 71.7% accurate diagnosis rate achieved by a commercial state-of-the-art diagnosis tool. Furthermore, a CM-LCV design has been incorporated into a larger test chip and fabricated by an industry partner in a 14nm process. Diagnosis of the failing test chip data provided by the industry partner demonstrates the value of the HFAD methodology, with multiple cases of HFAD detecting defects that the commercial diagnosis tool missed. The remainder of this chapter describes the contributions of this dissertation

and the directions for future work.

5.1 Contributions

This dissertation has presented advancements to the state-of-the-art in the fields of logic test chip design, test, and diagnosis. The main contributions for each of these aspects are summarized below:

Logic Test Chip Design

- The definition of a new logic test chip based on two-dimensional arrays of VH-bijective FUB modules. Two variant FUB arrays are also proposed: a heterogenous FUB array with multiple different FUB functions, and a pipelined FUB array with sequential elements inserted at every array connection.

- The definition of VH-bijectivity, a special property of four-port functions, and an examination of its relationship to the extant concept of orthogonal Latin squares. Proofs were supplied concerning the VH-bijectivity of various derivatives of a VH-bijective function, including its inverse.

- The description of the composability of various properties within the FUB array. Composability is shown for varying degrees of fault coverage, logical design features, physical design features, and fault distinguishability.

- The description of a synthesis flow to translate the logical design of a two-dimensional FUB array into a physical design that leverages the composability of the FUB array to meet various design requirements (fault coverage, standard cell composition, etc.).

Test

- Methods for constructing tessellation test sets for the FUB array. The tessellation test patterns apply every input pattern to each FUB within the FUB array, in a constant number of test patterns regardless of the size of the array.
- Methods for constructing super exhaustive test sets for the FUB array. These test sets are shown to apply every input pattern multiple times to each FUB within the FUB array.
- A built-in self-test (BIST) scheme for the FUB array that utilizes the properties of the tessellation patterns to achieve high fault coverage with a simple circular feedback architecture. Conditions on the size of the FUB array for proper implementation of this BIST scheme are presented.

5.1.1 Diagnosis

- A description of the special properties of the forward and reverse error bounds obtained from the FUB array. A proof is provided regarding the necessity for more than one defective FUB to be present in order for the error bounds intersection to be empty.
- A hierarchical FUB array diagnosis flow tailored for the FUB array, with improved performance for multiple defects.

5.2 Future Work

This dissertation has presented a new logic test chip, the Carnegie Mellon Logic Characterization Vehicle, along with methodologies for its test and diagnosis. Listed below are some possible avenues for future work.

- *Refinements to the HFAD diagnosis flow:* It is posited that two defects in the FUB array can always be accurately diagnosed. The current implementation of the HFAD

flow achieves less than 100% accuracy for simulations of two injected faults, representing a significant target for future work. Performance improvements would also be beneficial in allowing for more extensive simulation experiments to better characterize the tool's capabilities.

- *Exploration of additional FUB functions:* The large number of viable FUB functions presents a significant opportunity. Both a simple search strategy across possible FUB functions and developing techniques to create application-specific FUB functions are promising avenues for improving the performance of the CM-LCV.

- *Higher dimensional FUB arrays:* The current restriction to two-dimensional FUB arrays is arbitrary; the concept of VH-bijectivity (or, equivalently, pairs of orthogonal latin squares) can easily be extended to higher dimensions [79]. All of the test and diagnosis concepts discussed in this work should also be adaptable to higher dimensions as well. Furthermore, it is posited that up to n defects can always be accurately diagnosed in an n-dimensional extension of the FUB array presented in this dissertation.

- *On-Chip diagnosis:* the error bound behavior observed in the FUB array can be leveraged for on-chip diagnosis. If coupled with the BIST architecture included in this dissertation it should be possible to implement a fully self-contained logic test chip that can test and diagnose itself.

Bibliography

[1] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[2] C. Weber, "Yield learning and the sources of profitability in semiconductor manufacturing and process development," in *13th Annual IEEE/SEMI Advanced Semiconductor Manufacturing Conference. Advancing the Science and Technology of Semiconductor Manufacturing. ASMC 2002 (Cat. No.02CH37259)*, pp. 324–329, 2002.

[3] D. H. C. Du, S. H. C. Yen, and S. Ghanta, "On the general false path problem in timing analysis," in *26th ACM/IEEE Design Automation Conference*, pp. 555–560, June 1989.

[4] G. A. Klutke, P. C. Kiessler, and M. A. Wortman, "A critical look at the bathtub curve," *IEEE Transactions on Reliability*, vol. 52, pp. 125–129, March 2003.

[5] T. Kim, W. Kuo, and W.-T. K. Chien, "Burn-in effect on yield," *IEEE Transactions on Electronics Packaging Manufacturing*, vol. 23, pp. 293–299, Oct 2000.

[6] T. Karnik, S. Borkar, and V. De, "Sub-90 nm technologies-challenges and opportunities for cad," in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, pp. 203–206, Nov 2002.

[7] "Process integration, devices and structures," tech. rep., International Technology Roadmap for Semiconductors (ITRS), 2013.

[8] P. Solutions, "Personal communication."

[9] R. F. Pease and S. Y. Chou, "Lithography and other patterning techniques for future electronics," *Proceedings of the IEEE*, vol. 96, pp. 248–270, Feb 2008.

[10] D. Hisamoto, W.-C. Lee, J. Kedzierski, H. Takeuchi, K. Asano, C. Kuo, E. Anderson, T.-J. King, J. Bokor, and C. Hu, "Finfet-a self-aligned double-gate mosfet scalable to 20 nm," *IEEE Transactions on Electron Devices*, vol. 47, pp. 2320–2325, Dec 2000.

[11] R. Chau, S. Datta, M. Doczy, B. Doyle, J. Kavalieros, and M. Metz, "High-κ /metal-gate stack and its mosfet characteristics," *IEEE Electron Device Letters*, vol. 25, pp. 408–410, June 2004.

[12] NXP, "Personal communication."

[13] P. Solutions, "Personal communication."

[14] M. B. K. M. Bhushan, *Microelectronic Test Structures for CMOS Technology*. Springer, 2011.

[15] S. Saxena, C. Dolainsky, M. Lunenborg, J. Cheng, B. Yu, R. Vallishayee, and D. Ciplickas, "Impact of layout at advanced technology nodes on the performance and variation of digital and analog figures of merit," in *2013 IEEE International Electron Devices Meeting*, pp. 17.2.1–17.2.4, Dec 2013.

[16] A. J. Strojwas, "Conquering process variability: A key enabler for profitable manufacturing in advanced technology nodes," in *2006 IEEE International Symposium on Semiconductor Manufacturing*, pp. xxiii–xxxii, Sept 2006.

[17] P. K. Nag, A. Gattiker, S. Wei, R. D. Blanton, and W. Maly, "Modeling the economics of testing: a dft perspective," *IEEE Design Test of Computers*, vol. 19, pp. 29–41, Jan 2002.

[18] A. D. Friedman, "Fault detection in redundant circuits," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 99–100, Feb 1967.

[19] J. E. Smith, "On the existence of combinational logic circuits exhibiting multiple redundancy," *IEEE Transactions on Computers*, vol. C-27, pp. 1221–1225, Dec 1978.

[20] C. Hora and S. Eichenberger, "Towards high accuracy fault diagnosis of digital circuits," pp. 47–51, 01 2004.

[21] C. Hess, B. Stine, L. Weiland, and K. Sawada, "Logic characterization vehicle to determine process variation impact on yield and performance of digital circuits," in *International Conference on Microelectronic Test Structures*, pp. 189–196, April 2002.

[22] M. Fujii, K. Nii, H. Makino, S. Ohbayashi, M. Igarashi, T. Kawamura, M. Yokota, N. Tsuda, T. Yoshizawa, T. Tsutsui, N. Takeshita, N. Murata, T. Tanaka, T. Fujiwara, K. Asahina, M. Okada, K. Tomita, M. Takeuchi, and H. Shinohara, "A large scale, flip-flop ram imitating a logic lsi for fast development of process technology," in *Microelectronic Test Structures, 2007. ICMTS '07. IEEE International Conference on*, pp. 131–134, March 2007.

[23] A. Pancholy, J. Rajski, and L. J. McNaughton, "Empirical failure analysis and validation of fault models in cmos vlsi," in *Proceedings. International Test Conference 1990*, pp. 938–947, Sep 1990.

[24] S. Eichenberger, J. Geuzebroek, C. Hora, B. Kruseman, and A. Majhi, "Towards a world without test escapes: The use of volume diagnosis to improve test quality," in *Test Conference, 2008. ITC 2008. IEEE International*, pp. 1–10, Oct 2008.

[25] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu, "Process defect trends and strategic test gaps," in *IEEE International Test Conference*, pp. 1–8, Oct 2014.

[26] G. R. Case, "Analysis of actual fault mechanisms in cmos logic gates," in *Proceedings of the 13th Design Automation Conference*, DAC '76, (New York, NY, USA), pp. 265–270, ACM, 1976.

[27] M. A. Breuer, "The effects of races, delays, and delay faults on test generation," *IEEE Transactions on Computers*, vol. C-23, pp. 1078–1092, Oct 1974.

[28] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition fault simulation," *IEEE Design Test of Computers*, vol. 4, pp. 32–38, April 1987.

[29] R. D. Blanton and J. P. Hayes, "Properties of the input pattern fault model," *International Conference on Computer Design*, pp. 372–380, Oct 1997.

[30] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schloeffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson, "Defect-oriented cell-aware atpg and fault simulation for industrial cell libraries and designs," in *2009 International Test Conference*, pp. 1–10, Nov 2009.

[31] S. C. Ma, P. Franco, and E. J. McCluskey, "An experimental chip to evaluate test techniques experiment results," in *International Test Conference*, pp. 663–672, Oct 1995.

[32] K. Y. Cho, S. Mitra, and E. J. McCluskey, "Gate exhaustive testing," in *IEEE International Conference on Test, 2005.*, pp. 7 pp.–777, Nov 2005.

[33] K. Y. Cho and E. J. McCluskey, "Test set reordering using the gate exhaustive test metric," in *25th IEEE VLSI Test Symposium (VTS'07)*, pp. 199–204, May 2007.

[34] W. Qiu, J. Wang, D. M. H. Walker, D. Reddy, X. Lu, Z. Li, W. Shi, and H. Balachandran, "K longest paths per gate (klpg) test generation for scan-based sequential circuits," in *2004 International Conferce on Test*, pp. 223–231, Oct 2004.

[35] M. Abramovici and M. A. Breuer, "Fault diagnosis based on effect-cause analysis: An introduction," in *17th Design Automation Conference*, pp. 69–76, June 1980.

[36] S. J. Sangwine, "Deductive fault diagnosis in digital circuits: a survey," *IEE Proceedings E - Computers and Digital Techniques*, vol. 136, pp. 496–504, Nov 1989.

[37] J. A. Waicukauski and E. Lindbloom, "Failure diagnosis of structured vlsi," *IEEE Design Test of Computers*, vol. 6, pp. 49–60, Aug 1989.

[38] I. Pomeranz and S. M. Reddy, "On dictionary-based fault location in digital logic circuits," *IEEE Transactions on Computers*, vol. 46, pp. 48–59, Jan 1997.

[39] B. Chess and T. Larrabee, "Creating small fault dictionaries [logic circuit fault diagnosis]," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 346–356, Mar 1999.

[40] I. Pomeranz and S. M. Reddy, "On the generation of small dictionaries for fault location," in *1992 IEEE/ACM International Conference on Computer-Aided Design*, pp. 272–279, Nov 1992.

[41] C. Hora, R. Segers, S. Eichenberger, and M. Lousberg, "An effective diagnosis method to support yield improvement," in *Proceedings. International Test Conference*, pp. 260–269, 2002.

[42] X. Yu and R. D. Blanton, "Estimating defect-type distributions through volume diagnosis and defect behavior attribution," in *2010 IEEE International Test Conference*, pp. 1–10, Nov 2010.

[43] B. Benware, C. Schuermyer, M. Sharma, and T. Herrmann, "Determining a failure root cause distribution from a population of layout-aware scan diagnosis results," *IEEE Design Test of Computers*, vol. 29, pp. 8–18, Feb 2012.

[44] V. Muthumalai, D. Iverson, A. Sinnott, R. Desineni, R. Turakhia, T. Berndt, and N. Bell, "Successful yield ramp using product test, scan and memory diagnosis," in *25th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC 2014)*, pp. 1–4, May 2014.

[45] B. Kruseman, A. Majhi, C. Hora, S. Eichenberger, and J. Meirlevede, "Systematic defects in deep sub-micron technologies," in *Test Conference, 2004. Proceedings. ITC 2004. International*, pp. 290–299, Oct 2004.

[46] A. D. Friedman, "Easily testable iterative systems," *IEEE Transactions on Computers*, vol. C-22, pp. 1061–1064, Dec 1973.

[47] Z. Liu, P. Fynan, and R. D. Blanton, "Front-end layout reflection for test chip design," in *IEEE International Test Conference*, Nov 2017.

[48] S. Mittal, Z. Liu, B. Niewenhuis, and R. D. Blanton, "Test chip design for optimal cell-aware diagnosis," in *IEEE International Test Conference*, Nov 2016.

[49] R. D. Blanton, B. Niewenhuis, and Z. D. Liu, "Design reflection for optimal test-chip implementation," in *IEEE International Test Conference*, pp. 1–10, Oct 2015.

[50] Z. Liu, B. Niewenhuis, S. Mittal, and R. D. Blanton, "Achieving 100% cell-aware coverage by design," in *Design, Automation Test in Europe Conference*, pp. 109–114, March 2016.

[51] Y. Zhang and V. D. Agrawal, "A diagnostic test generation system," in *IEEE International Test Conference*, pp. 1–9, Nov 2010.

[52] P. Fynan, Z. Liu, B. Niewenhuis, S. Mittal, M. Strojwas, and R. D. Blanton, "Logic characterization vehicle design reflection via layout rewiring," in *IEEE International Test Conference*, Nov 2016.

[53] L. Euler, "Recherches sur une nouvelle espéce de quarrés magiques," in *Verhandelingen uitgegeven door het zeeuwsch Genootschap der Wetenschappen te Vlissingen*, vol. 9, pp. 85–239, 1782.

[54] E. T. Parker, "Construction of some sets of mutually orthogonal latin squares," *Proceedings of the American Mathematical Society*, vol. 10, no. 6, pp. 946–949, 1959.

[55] R. M. Wilson, "Concerning the number of mutually orthogonal latin squares," *Discrete Mathematics*, vol. 9, no. 2, pp. 181 – 198, 1974.

[56] M. G. Tarray, "Le probléme des 36 officiers," in *Comptes Rendus Assoc. France Av. Sci. 29*, vol. 2, pp. 170–203, 1900.

[57] R. C. Bose and S. S. Shrikhande, "On the falsity of euler's conjecture about the non-existence of two orthogonal latin squares of order 4t + 2," in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 45, pp. 734–737, March 1959.

[58] E. T. Parker, "Orthogonal latin squares," in *National Academy of Sciences of the United States of America*, vol. 45, pp. 859–862, May 1959.

[59] D. Klyve and L. Stemkoski, "Graeco-latin squares and a mistaken conjecture of euler," vol. 37, p. 2, 01 2006.

[60] G. E. P. Box, W. G. Hunter, and S. J. Hunter, *Statistics for Experimenters*. John Wiley & Sons, Inc., 1978.

[61] R. O Kuehl, *Design of experiments : statistical principles of research design and analysis*. Duxbury/Thomson Learning, 2000.

[62] M. Y. Hsiao, D. C. Bossen, and R. T. Chien, "Orthogonal latin square codes," *IBM Journal of Research and Development*, vol. 14, pp. 390–394, July 1970.

[63] M. Y. Hsiao and D. C. Bossen, "Orthogonal latin square configuration for lsi memory yield and reliability enhancement," *IEEE Transactions on Computers*, vol. C-24, pp. 512–516, May 1975.

[64] A. R. Alameldeen, Z. Chishti, C. Wilkerson, W. Wu, and S. L. Lu, "Adaptive cache design to enable reliable low-voltage operation," *IEEE Transactions on Computers*, vol. 60, pp. 50–63, Jan 2011.

[65] R. Datta and N. A. Touba, "Generating burst-error correcting codes from orthogonal latin square codes – a graph theoretic approach," in *2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 367–373, Oct 2011.

[66] S. E. Lee, Y. S. Yang, G. S. Choi, W. Wu, and R. Iyer, "Low-power, resilient interconnection with orthogonal latin squares," *IEEE Design Test of Computers*, vol. 28, pp. 30–39, March 2011.

[67] A. D. Keedwell and J. Dénes, *Latin Squares and Their Applications.*, vol. Second edition. North Holland, 2015.

[68] J. D'Nes, A. Keedwell, and G. Beliavskaia, *Latin Squares: New Developments in the Theory and Applications*. Advances in Psychology, North-Holland, 1991.

[69] J. Egan and I. M. Wanless, "Enumeration of MOLS of small order," *ArXiv e-prints*, June 2014.

[70] R. L. Rudell, "Multiple-valued logic minimization for pla synthesis," tech. rep., DTIC Document, 1986.

[71] A. Chatterjee and J. Abraham, "Ncube: an automatic test generation program for iterative logic arrays," in *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 428–431, Nov 1988.

[72] J. E. Nelson, J. G. Brown, R. Desineni, and R. D. Blanton, "Multiple-detect atpg based on physical neighborhoods," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 1099–1102, July 2006.

[73] Y. T. Lin, O. Poku, N. K. Bhatti, and R. D. Blanton, "Physically-aware n-detect test pattern selection," in *2008 Design, Automation and Test in Europe*, pp. 634–639, March 2008.

[74] M. E. Amyeen, S. Venkataraman, A. Ojha, and S. Lee, "Evaluation of the quality of n-detect scan atpg patterns on a processor," in *2004 International Conferce on Test*, pp. 669–678, Oct 2004.

[75] Y. T. Lin, O. Poku, R. D. Blanton, P. Nigh, P. Lloyd, and V. Iyengar, "Evaluating the effectiveness of physically-aware n-detect test using real silicon," in *2008 IEEE International Test Conference*, pp. 1–9, Oct 2008.

[76] A. Krasniewski and S. Pilarski, "Circular self-test path: a low-cost bist technique for vlsi circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 46–55, Jan 1989.

[77] F. Corno, N. Gaudenzi, P. Prinetto, and M. Reorda, "On the identification of optimal cellular automata for built-in self-test of sequential circuits," in *IEEE VLSI Test Symposium*, pp. 424–429, Apr 1998.

[78] G. Rossum, "Python reference manual," tech. rep., Amsterdam, The Netherlands, The Netherlands, 1995.

[79] J. Arkin and E. G. Straus, "Latin k-cubes," *Fibonacci Quarterly*, vol. 12, no. 0974, pp. 288–292, 1974.