Test Automation for Software Quality Assurance:

A comparative study of hardware and software test automation

Zhiyang Ong *

University of Southern California
Department of Electrical Engineering
Summer Internship Report

Commencement Date: May 18, 2006
Submission Date: August 8, 2006

Supervisor: Professor Sandeep K. Gupta (sandeep@poisson.usc.edu)

*Email: zhiyang@ieee.org; Contact number: +1 213 284 1562

Acknowledgments

The author would like to thank his supervisor Nicholas Uchida and manager William Allen for advising him on test automation for software quality assurance during his internship with Symantec's Macintosh Products Group. They and fellow colleagues in the Macintosh team provided the author lots of advice and insight into various design for test techniques in software architectural design and use of assertions for software verification. They also provided opportunities to develop test automation scripts for graphical user interface applications, and scripts for automated analysis of test results.

He would also like to thank Professor Sandeep K. Gupta for supervising his internship class, and enlightening him in discussions on how testing methodologies for can be used for testing VLSI circuits. In particular, he helped reined in the author from working on a myriad range of topics related to software test automation by getting him to focus on specific research topics such as test coverage and software metrics, which are mutually interesting to both.

Finally, the author wishes to express his gratitude to friends whom he had discussions about this research project. In particular, he would like to thank Kelli Mays-Aboagye for her encouragement and support when the author was having difficulties with doing this research project. She also offered to provide the author with resources on immunology and virology when the author was attempting to develop software to mimic immune systems for automated test pattern generation.

Executive Summary

Test automation, which is often misconstrued as a cost-cutting measure to reduce the size of quality assurance staff, allows more test cases to be tested effectively and efficiently. Test automation can be used to improve test coverage, and shift focus on verification, validation, and testing to developing appropriate test cases, as opposed to spending unnecessary effort in entering test input for designed test cases, and ensuring that the test output are correct.

Various techniques associated with test automation in VLSI circuit design and software engineering were discussed. Some selected topics for discussion included design for test techniques, including built-in self-test, automated test pattern generation, design of experiments, and test coverage. A comparison an analogy between the application of such techniques in VLSI circuit design and software development is provided.

Finally, the techniques for test automation in VLSI design or software development are not mutually exclusive, since they share a lot of similarities under the umbrella of systems architecting and engineering. Hence, software developers and testers should learn from VLSI circuit designers and testers, and vice versa to learn about each other's concepts and techniques in test automation. This will enable them to incorporate other people's innovations in test automation into their software and/or hardware products.

"Dont be encumbered by past history, go out and do something wonderful."
- Robert Noyce, Intel founder and chair

Contents

1 Introduction

1.1 Difficulties of Software Testing

Software testing is the process of verifying a software system according to its design specifications through its execution, and validating its intended use by the user(s) in the desired environment(s) [1]. It is used to exercise the computer program to test certain blocks of code in the structure of the software design, and the validity of its specifications. It is also used to detect faults in the computer program, and in parts of the computer program that is more likely to fail [2]. The act of software execution differentiates it from static software verification, during which the source code is read and analyzed. Since software specifications define the appropriate functions of the software, incorrect functions that are carried out by the software indicate its failure. These failures, defects, or bugs, are due to errors in its source code [1].

There are various reasons for the evasion of software bugs from the scrutiny of the quality assurance team, which end up being reported by users. By empowering software testers with different testing techniques, and educating them about the application domain of the software usage, testers would be able to source software defects. Subsequently, they would also be able to utilize the skills that they had acquired to determine what caused the errors to occur, and the locality of the bugs [1].

To understand how will the software be used by clients to perform certain tasks, and its operating environment, testers have to seek the users' perspectives on how will the software be used so that they can innovate effective and efficient test cases. This requires the testers to have adequate experience and skills in software development, such as programming, data structures, design patterns, formal methodologies, and algorithms. Modeling the operating environment can include interaction with users, software interfaces in the operating system or runtime library, communication interfaces between physical devices and computer networks, and interfaces for file manipulation and modification [1].

Once the software's interaction with its operating environment can be modeled, test cases can be selected for evaluating the software to verify and validate its correctness. In addition, test input for unit testing, integration testing, and system testing to be designed; regression testing should be carried out whenever modifications are made to the source code. Test scenarios should also be selected based on the specifications and the operating environment to test the functions and structure of the software. Subsequently, software metrics, such as the number of errors found per million lines of code, can be used to estimate how far has testing progressed [1].

While testing software, a tester should consider valid, invalid, and unexpected inputs. Also, memory usage and functional failure of system processes should also be evaluated for

their impact. In addition, changes made to files, directories, and/or disks by another program must be reflected in other programs that are concurrently accessing the shared resources; it is important to track these modifications as contemporary operating systems may only allow simultaneous read access, and one exclusive write access, to the same resources. Consideration of these scenarios lead to many test cases, which result in significant investments in verification, validation, and testing to achieve adequate test coverage [1].

By considering each state in the execution of the program as a boolean condition that may or may not occur, these states can be tabulated in a truth table to determine each possible combination of these states. If a state is considered to be too complex compared to others in the truth table, it can be modeled as several states. This leads to a possible combination of $(2^n - 1)$ states to be evaluated as test cases, where n is the number states; these test cases can also be modeled as a binary decision diagram. Other methods based on statistical analysis (see Section 7 on Design of Experiments), finite state machines, or graphs, such as Petri nets, can also be used to select a suitable subset of these combinations for testing. Methods for generating test patterns include evolutionary algorithms and formal methodologies [1].

It is evident from above that software tests are unable to exhaustively cover each possible test case, as that would require enormous amount of resources in terms of man-hours to do so. This leads to a need for testing methodologies that provide adequate and effective test coverage, without having redundancy in evaluating test cases. In addition, the range of input values that can be entered as input can be large, and the boundary conditions on these valid ranges should be tested to avoid exhaustive testing of all possible values. Testers must also consider how the sequence in which the input was entered into the software, or its modules/sub-modules, will influence its behavior. This is important for concurrently processed input data. Moreover, the software can be intentionally inserted with software defects so that test cases can be designed to locate them, and, possibly, other errors that were unintentionally entered into the source code [1].

1.2 Software Test Plans

Test plans and test system development can be based on the available software requirements and design specifications. They can also be used to determine the amount of test coverage using a numeric approach, or a cross-reference approach. In the former approach, benchmarks are created for determining the progress and coverage of testing. The benchmarks indicate the percentage of test cases that have been tested, the percentage of test cases that have passed the tests, and the risk involved in not having ran this test case. In the latter approach, techniques such as functional tracing and function-test matrix use the functions that are required to be provided by the software to check if they have all be satisfied by the software

during testing [3].

1.3 Why Use Test Automation for Software Engineering (and VLSI Circuit Design?)

1.3.1 Software Test Automation and Formal Verification

Software test automation is using software to automate any part of the software testing process. It may involve generating test inputs and expected test results, test execution, and evaluation of test results by comparing them to expected test results. Assertions, pre-conditions, and post-conditions can be used to facilitate the automated testing environment in determining and locating errors, which will be isolated or corrected [1]. In addition, it can be used to help make software testing more effective, utilize less human resources, and be repeatable [4]. That is, it can be used to reduce the need for the labor-intensive task of manually entering input to the software, based on different test scenarios. This allow test results under specific conditions to be duplicated as there won't be any human errors in entering the input data on different executions of the software under the same test scenario [1].

It can also provide the coverage of the executed tests, and support regression testing. Its advantages include the following: verifying fixes of software bugs are faster and more efficient as the fixed code is evaluated with the same test inputs, facilitate debugging for the aforementioned reason, and facilitate regression testing. In addition, it is used in combination with manual testing in cases where there exists significant user interaction with the graphical user interface (GUI), and in embedded systems that require a lot of domain knowledge and environmental conditions. This is because developing and/or maintaining software testing tools for test automation can be more costly than manual testing [4].

Test code can be also embedded into the source code as built-in self-tests, where the test code may have various procedures to introduce redundancy into software testing. Thus, these testing procedures should test the same set of test cases, and be independent of or lowly correlated with each other. After executing each of these redundant procedures, their output test results are collected for comparison with each other. If these test results are the same for a set of test cases, it indicates that the software probably implements the specifications for these test cases correctly. It is unlikely that a set of test cases is improperly verified after taking different approaches to testing that set of test cases. Alternatively, the test code can undo the function of the segment of the source code, and compared the output of the undo operation with the given inputs for any discrepancies [1].

The software architecture and source code should be designed and developed with the objective of making software testing easier [1]. This is important since testing usually takes

up more time than scheduled for [5], and the successful execution of these tests can only indicate that no software errors exist for the executing the software under these test scenarios. That is, there is no guarantee that this software has no errors; reliability models can then be used to predict how the software will perform correctly before failure when it receives poor data input [1].

Automated black-box testing of software components, which include modules and classes, can provide effective and efficient modular testing. Test programs can be automatically generated for software modules, along with their input programs. In addition, software module wrappers that serve as test mechanisms can be produced automatically or semi-automatically. That is, extra code for each software module is provided to serve as a compatible interface between that software module and the test programs/scripts. This makes it easier for the test program to call functions/methods for testing [6].

The tools for automated testing shall be developed to implement selected test cases to provide meaningful test results that can be collated and analyzed. Using such automated testing tools to execute the software for the verification of its behavior is meaningful only when the software fails to perform as expected. Hence, determining what shall be the test inputs, and what shall the expected output and behavior be, is necessary to prior to the execution of the software. This also enables software testers to write scripts for filtering the large amount of generated test results to obtain information that indicates unexpected software performance; unexpected behavior can be determined by comparing the expected output with the test results [7].

The organization of the test cases dictate which test case scenarios can be selected for execution, and how to interpret its test results. It should allows subsets of selected test cases or part of test scripts to be executed during debugging so that software errors can be determined, as these selected portions of the test scripts act as breakpoints in the software. However, the use of graphical user interfaces (GUIs) complicate things further. In addition, there exist problems in comparing expected outputs with test results, as they may raise false positives. Such problems include the rounding off values due to the chosen numerical representation of variables in the program, and incorrect format of output strings. Consequently, the test automation architecture must be flexible to various input formats and a considerable range of selected test input values, and be adaptable to changes in the requirements and test case scenarios [7].

Finally, by formally describing the intended functions of the software using a model-based specification language, software requirements that describe the functions of each module can be generated automatically. These software can be enhanced by semi-formal and informal functional descriptions. When informal descriptions are used, user interaction is required to generate software requirements describing the behavior of components. This is due to

the difficulties of deriving functional requirements from informal descriptions. For formal specifications, preconditions and postconditions can be used to determine how the software components deal with erroneous input, and only assert acceptable output. Subsequently, input test data, automated testing facilities, and expected test results for these test suites and input test data can be automatically generated [6].

1.3.2 VLSI Test Automation and Formal Verification

To ensure the correctness of an IC, simulation can be used to determine if a circuit has correct values in the output parameters of the IC when given a set of input signals. Hence, the effective use of simulation depends on the selection of test inputs, and acute knowledge of the circuit to select test inputs with significant test coverage. However, when verification is used to determine if a circuit is functioning correctly, it obtains information about the parameters of the circuit from its description, and affirm that these extracted circuit parameters are correct. That is, verification does not depend on the selection of test input patterns. Since not all functional behavior of the circuit can be simulated, its non-functional specifications and structure can be adequately and succinctly specified without ambiguity for verification. Due to this need for additional specifications, verification is not used widely in hardware or software validation, verification, and testing. Although the scope of using verification tools is limited, they provide an infallible means of establishing that the circuit performs correctly to meet certain specifications [8].

Examples of verification tools used in VLSI test automation include electrical verification and timing verification. The former ascertains the adherence of design rules and electrical rules, which are specific to each design style. It also requires keen knowledge of the circuit structure at different hierarchical levels so that the verification tools can determine if the aforementioned rules are consistently followed at various hierarchical levels. The latter is used to establish if the critical paths (or worst-case path) in the circuit have meet their (its) timing requirements. Since the delay on a path depends on the inputs applied to the circuit, the critical paths that are identified for observation of circuit parameters during circuit simulation may not be the worst-case path. Thus, a timing verification tool, or timing verifier, is used to enumerate each path in the electrical network, and sort these paths according to their delays. That is, the sorting procedure will determine the path with the longest delay [8].

Next, the behavior of a circuit's component, such as a transistor, gate, or module, can be mathematically described as a function of its inputs and internal state. The description of all components in the circuit can be synthesized into a register transfer level (RTL) or netlist description of the circuit. Formal verification can subsequently be used to compare this derived behavioral description of the circuit with its specifications to determine if they

are mathematically equivalent, so that it can be ascertained that this circuit will perform as specified. Since formal verification is computationally intensive to enumerate all possible input patterns and sequences, and compare the outputs of two different circuit descriptions, it can only be applied for certain categories of circuits where the search space is not unreasonably large [8].

For formal verification to work effectively, the circuit must be unambiguously specified with formal methods that are mathematical description models [9]. Following this, either of the following two types of formal verification techniques are used to determine if the circuit will perform as specified: equivalence proving and property proving. In equivalence proving, a low-level description of the circuit is compared with a high-level description of the circuit to determine if they are mathematically equivalent or only perform the same functions. On the other hand, property proving determines if a circuit description possess characteristics. If the circuit is found not to own the required attributes, the property proving tool would examine the conditions that lead to this error [8].

During the fabrication and testing process of IC chips, faults can be introduced in these chips. Hence, there is a need to determine if they are functioning correctly with different input and under various conditions. Consequently, when some of these faulty products are released to consumers, an expensive recall, repair, or replacement of products would be needed. However, it is cumbersome and time-consuming to hook up expensive automated test equipment (ATE) to the IC chip for adequate testing. Thus, the design-for-testability (DFT) approach is used to facilitate IC design and testing so that it can avoid some of the aforementioned problems [8]. By implementing different DFT techniques, testing of VLSI circuits would require less human resources, time, and monetary costs as it can be automated.

Testing procedures in VLSI fall under the following categories: diagnostic test, functional test, and parametric test. Diagnostic tests are used to identify the fault and determine its location, while functional tests are used to determine if the IC chip functions correctly. Parametric tests are used to observe continuous parameters in static and dynamic tests to determine if they exhibit erroneous values [8].

2 Software (and Hardware) Test Coverage and Metrics

2.1 Software Test Coverage, Metrics, and Adequacy

The improvement of test coverage and analysis of test results facilitates debugging. It results in allowing the behavior of the system to be examined for a greater range of test cases to determine locality of the software errors, and to eliminate or isolate them [6].

The adequacy of criteria for objective measurement of the quality in software testing will be discussed as follows. To discuss how adequate existing or proposed criteria are, various

concepts regarding software testing metrics need to be defined. A test case is an input that the program will receive and process during test execution, and a test set refers to a collection of test cases. Statement coverage is the extent to which each statement in the program is executed at least once during testing, and complete statement coverage (100% coverage) is an adequate criterion; an adequate test set achieves complete code coverage [2, 10].

Similarly, branch coverage is the extent to which all branch statements in the computer program are exercised at least once during testing, and its adequacy is determined by complete branch coverage (100% coverage). Also, path coverage is the extent to which all paths in the computer program are executed at least once during testing, and its adequacy is determined by complete path coverage (100% coverage) [10]. Software bugs can also be intentionally embedded into the computer program; such programs are known as mutants. Test sets that produce different outputs for mutants and nonmutants are known to have killed the mutants. Therefore, mutating adequacy/score refers to the extent in which mutants are dead [2].

The adequacy for software testing criteria indicates what aspects of the computer program need to be tested before it is void of certain types of software errors. However, there is a need to verify that the software has been adequately tested according to existing criteria, so that it can be assured that the software has been adequately tested. This implies that test coverage would be a percentage ranging from zero to one, inclusive. Also, test sets have to meet a certain extent of test coverage for them to be adequate for software testing; this extent is the test data adequacy criterion. Having a tighter test data adequacy criterion improves the ability to detect software bugs during testing. Therefore, this criterion is often used to compare software testing methodologies. Unfortunately, there is no strong correlation between the the test data criterion and the quality of the software, which includes its correctness and reliability [2].

Test data adequacy can be used as a terminating condition to determine when the software shall stop processing various test cases and execution. Else, if test coverage is not improved during software testing, different types of tests may be required for execution to improve test coverage. Also, it can be used to determine if execution of test cases have been repeated. Testing at this low-level requires valid test sets that indicate the types and sources of errors in their testing results. For testing at higher levels, the test cases must be reliable so that the test sets will consistently and repeatedly produce test results under the same test conditions [2].

The adequacy criteria for software testing can be based on specifications or program-based. With regards to the former, the test set is adequate if the computer program has been exercised with all software specifications. As pertinent to the latter, the test set is adequate if the computer program is adequately exercised. Note that specification-based or

program-based adequacy criteria does not depend on test results, since they depend whether the program has been exercised or executed with certain specifications. They are part of black-box testing [2].

These code coverage metrics give an inaccurate estimation of how software testing is progressing. If the testing process indicate high scores on a metric for a set of test cases or certain types of software bugs, it does not mean that other test cases are tested at high code coverage, or other types of software bugs are nonexistent. For example, software testing that has full statement coverage does not guarantee that it is tested with various values that include boundary values, and values lying inside or outside of the boundaries. Consequently, software testing should use code coverage to determine if there exists any block of code that is left untested, and should not use it to determine its progress [10].

Software test coverage can be facilitated easily with the UNIX system command "gcov", which is used with another UNIX system command "gcc" to discover segments of the software that are untested. It can also be used with the UNIX system command "gprof" to profile the computer program, and ascertain the blocks of the program that consume more processing time than others. By executing this command after compiling the computer program with "gcc" and its profiling option, the standard output displays a listing of the source code for the computer program with a number on the left-hand side to indicate the number of times each line of code is executed. See its help pages in its UNIX manual for further information. Lastly, if a line of code is not executed at all, and should have been, a series of hash characters will be placed by its left side to indicate that test coverage is not 100%.

2.2 Hardware Test Coverage and Metrics

Metrics to help determine the quality of the VLSI chip include testability, yield and manufacturability, reliability, and ease of enhancing its technology (modifiability). To ensure that ICs are seamlessly integrated into embedded systems, they need to be verified, validated, and tested for functional correctness and having high non-functional qualities. These verification and testing of ICs should be carried out at reasonable speeds with available resources so that the time to market for the product is not considerably delayed. Also, to enhance the processes of verification and testing, the design and hierarchical architecture of the IC shall facilitate testing and verification [11].

If the yield of the manufactured ICs is low due to a high proportion of defective ICs, it can be attributed to the variations in the manufacturing processes and poor handling of delicate ICs by machinery. Hence, the design of ICs must facilitate manufacturing using the design for manufacturability (DFM) paradigm by considering statistical variations in its manufacturing processes, and environmental and operating conditions. ICs are tested for their functionality, voltage values, and timing specifications and parasitic impedance (process-dependent

parametric values). Consequently, the stringent criteria of the aforementioned testing metrics will determine the number of discarded ICs due to design errors, manufacturing errors, and careless handling of ICs. For example, such errors may include defective materials and chemicals, misalignment of lithography equipment, and process variations [11].

The aforementioned statistical variations affect the power supply voltages, operating temperatures, mobility of electrons, gate-oxide capacitance per unit area, and threshold voltages of devices in the IC. These variations will affect noise margins and signal integrity. Hence, if the design of ICs are made less responsive to statistical variations during manufacturing processes and operate the devices with larger design margins, it will improve the yield and the performance of the ICs. Some important DFM techniques include parametric yield estimation and maximization, worst-case analysis, and minimization of variability in the circuit's performance [11].

Fabrication process variations and random noise due to the environment affect the device parameters and circuit characteristics. These also affect the performance of the devices and circuits. Consequently, design of experiments (DOE) techniques can be used to model the performance of the circuit, and determine how much variability in the fabrication process will lead to a drastic decrease in the performance of the circuit [11].

The reliability of the IC can be attributed to poor design, manufacturing, and processing of chemicals and materials. They can be affected by low chip yield, and result in process-dependent errors such as electromigration, electrostatic discharge (ESD), and latch-up in CMOS I/O and internal circuits. In addition, they can be affected by single event upset, on-chip noise and crosstalk, and power and ground bounce as process-independent errors [11].

Since the design of electronic circuits and systems can last several generations of process technologies, the IC design must facilitate various scaling techniques to move to a new process technology. Such scaling techniques include constant-field scaling, constant-voltage scaling, and interconnect scaling. The possible decreasing time to market for the IC design with each subsequent process technology will complicate the problems of completing the timely upgrade to a new process technology [11].

3 Fault and Test Models

3.1 Hardware Fault Models

As the risks of inadequately testing ICs increases, different types of fault models need to be included in testing schemes so that more types of physical defects, and electrical and logical faults can be covered. Examples of some fault models include the stuck-at fault model and delay fault model. The former fault model deals with short circuits between the source and drain of a MOSFET, and open circuits at the drain or source of a MOSFET [8]. The latter

model deals with failures to meet timing requirements at the desired speed. Other fault models include the bridging fault model, which occur when a net is used to replace two other nets with an implementation of a elementary logic function (AND or OR the previous nets), open and break faults [8], and I_{DDQ} faults due to leakage currents [8, 12].

3.2 Software Test Models

The complexity of contemporary software requires the usage of models to facilitate the development of tests for the software, and comprehension of the software implementation of existing software architectural design. Test models or its revised versions are used to improve such comprehension of software implementation, and enhance test design to produce consistent and repeatable tests. They allow software testing to be systematic, focused, and automated to avoid repetitive work, to narrow the region where the source of the software bugs can be, and to reduce resource consumption. The use of test models allow methodical enumeration of inputs and states that affect the software. While such test models are developed empirically or from some heuristics, they result in more effective and efficient testing than the random or biased application of certain test inputs and coverage of selected states [4].

A test-ready model is comprised of adequate information to automatically produce and implement test cases. These models are used with testable Object-Oriented Analysis/Design (OOA/D) techniques to assure that test cases can be developed, so that software tests which implement these test cases can be executed to produce test results. Models for software analysis and design can be used as test models if they provide enough information to automatically produce and implement test cases. Else, test models need to be developed to facilitate software testing. Software designs that are formally specified can be easily and automatically be used as sources for generating test cases, test inputs, and expected test results. However, not all software designs can easily and quickly be formally specified. Hence, a mixture of unambiguous and complete test models that are problem-oriented need to be combined with formal test models. Together, they form a set of test models that are consistent, feasible, modifiable, traceable, usable, and verifiable [4].

4 Software Test Automation Architecture

Recent trends towards component-based software development improves software modularity while decreasing inter-module dependencies. This improves software development efficiency, as it is easier to develop software components without worrying about its dependencies with other components and how these software interfaces can be tested. When these software components are adequately tested before their integration, software testers can confidently

test the interaction between software modules without worrying about errors that may occur within a specific component. This also allows them to verify the concurrent execution of complex modules with greater ease [6].

Software test environments facilitate the automation of software testing with the use of an integrated set of tools. Such environments are used to help set up test plans, manage the test automation process, develop the testing tools for the environment, execute the automated tests, and garner test results for analysis. The architecture for these software test environments are used to specify how it uses its components and features to perform certain functions [13].

The analytical evaluation of the architecture for the software test environment is to determine if a framework is capable of achieving accurate, repeatable, and efficient and cost-effective testing. Consequently, this architecture will be able to facilitate the choices of making trade-offs to obtain performance gains during design. However, the development of such architectures occurs under constraints such as the performance of automated tests, and their ability to be transferred between platforms and modified to test future versions of the software. A proposed reference architecture for these environments employs the Software Architectural Analysis Method (SAAM) to evaluate software on various test environments under the aforementioned evaluation criteria and constraints [13].

Traditional methods of evaluating software test environment architectures use taxonomy to determine if they perform certain test functions, or possess certain features to support testing. In contrast to the elaborate written descriptions of taxonomies, SAAM uses a graphical description of rules and guidelines to help software test engineers develop and analyze their architectures. The rules and guidelines dictate what can or cannot be done, they do not merely categorize what has to be done. Subsequently, SAAM uses these guidelines and macroscopic perspectives on the system's structure to classify the functional sub-domains of the system. Next, it develops a graphical description of the architecture for each sub-domain, and describes a set of tasks needed to be completed to accomplish the desired objectives. Finally, it determines how well the architecture supports each task, which indicates how well the architecture facilitates testing [13].

To develop and analyze architectures, a canonical partition of the application domain is performed by describing its distinctive features. This must entail the characterization of specific tasks that must be carried out during testing, and those that will be used to automate this process, which is evolving with testing technology. Contemporary technology dictates the early use of software specifications and requirements in the software development lifecycle to concurrently develop, and verify and test software. This will prevent software errors that occur early in the lifecycle to propagate through different lifecycle phases [13].

Next, SAAM is used to obtain a graphical description of the structure of its components

such as its processes, modules, files, and database. They are used to individually specify the flow of information and control in the execution of the program. The resultant graphical descriptions provide a uniform abstraction of the system so that different architectures can be compared. Finally, each of the components is then allocated to the different sub-domains. At this juncture, it is noted that this is similar to integrated circuit design, as there are different hierarchical levels of design, each of which has its own styles for modeling the design [13].

To plan the test automation architecture, the interaction between its components and features with the environment must be identified and specified. When these are formally described, the tasks for test automation can be specified so that resources can be optimally spent only on incorporating components and features that are intended for, and can be included in, automated testing. This also helps to reduce the complexity of the test automation architecture, which already has significant difficulty to enable tools and utilities to be transferred between different operating environments [7].

Next, testing tools must be selected or modified from existing resources; available resources can also be used to purchase testing tools and additional features, or to develop them. Subsequently, a plan for incorporating these tools into test automation can be planned. If necessary, a unique test automation architecture may be developed for each environment to improve cost effectiveness and efficiency, and to reduce the complexity of the architectures. For example, automated tests in such architectures can be concurrently executed on different environments, or simultaneously evaluated the software's performance on different sets of test case scenarios. Finally, the structure indicating the components and their interaction must be provided along with the sequence describing how the tests shall be automated. This will help clarify issues regarding what tests can and shall be automated, their sequence of execution, and what are the expected and acceptable test results [7].

The analysis and interpretation of test automation results involves filtering poor test results from the collection of results. In addition, various memory and data components, and communication channels that may be compromised in the execution of the software under test must be checked. Moreover, the interaction between the software with others during concurrent execution must be tested and monitored for resource consumption, and the state of the operating environment and software. This is difficult to perform since many components in the operating environment may be affected, and the means to predict acceptable and expected test outcomes also need to be developed. Thus, when expected and acceptable test results are not obtained, warning flags need to be raised to indicate software malfunction and determine the locality of discovered software errors. Also, the use of assertions, preconditions, and postconditions can facilitate test automation in verifying the states, behavior, and outputs of the software [7].

Mechanisms for automatically generating expected and acceptable test results for com-

parison with test results is faster than those used by human testers, and do not make mistakes in pattern matching. Also, computers do not get distracted, and consequently fail to spot software errors [7].

The architecture for test automation can be simplified using divide and conquer on components that are not highly coupled [14]. This will facilitate concurrent testing of software modules, and allow simplify the mechanisms required to generate test cases. That is, the modularity of the software architecture facilitates concurrent unit and module testing. Lastly, the complexity of tools for analyzing the test results can also be reduced [7].

System architecting heuristics, and the divide and conquer framework can be used along with design patterns to design the software's architecture [15–19]. It would enable software developers and testers reuse common solutions to various problems in software engineering or programming. In addition, the application of design patterns, heuristics, and software frameworks as general tools for refactoring can facilitate the customization of each software product to enable software developers improve its properties or quality. Such properties required for developing good quality software include maintainability, modularity, portability, reconfigurability, reusability, simplicity, and testability [14,15,18,20,21]. Those properties in the software that are required for good operational quality include performance, reliability, and survivability [18, 22].

These design patterns can help reduce the complexity of the software so that it is more manageable for software developers and testers [23–25]. The gradual improvement of the software's architecture and its modifiability can come from analysis and verification of the non-functional requirements, which covers more than the behavioral and structural specifications of the software. This can help improve the knowledge and application of design patterns into the software architecture [21]. Finally, these software techniques can be combined with software architectural styles and object-oriented design to improve the design of software architectures [20].

The structure of such design patterns can be formerly modeled with the Unified Modeling Language (UML). This allows software architects to analyze the application of design patterns and document software architecture, without having to deal with unnecessary details such as assertions, association, and method definitions in the UML diagrams [19,26]. In addition, pattern-based tools to facilitate semi-automated UML modeling of design patterns can ensure that the behavioral and structural constraints of these design patterns are satisfied using model-checking [19], generate methods definitions in the source code, and determine which design patterns are correctly implemented in the source code. Moreover, the use of UML and Object Constraint Language (OCL) specification enhanced with temporal logic operators can be used to specify the pre-conditions, post-conditions, and assertions of the software architecture's refactoring [26].

5 Design for Test (DFT)

5.1 Software DFT

The design for testability (DFT) paradigm in software comes down to using system architecting heuristics, software frameworks, and design patterns to improve the high-level architectural design of the software so that it is more testable. Assertions, preconditions, and post-conditions can also be used to facilitate DFT techniques. See section 4 on "Software Test Automation Architecture" for more details.

5.2 Hardware DFT

The advancement of the semiconductor industry into the nanometer scale (ultra deep submicron - UDSM technology) raises daunting challenges in reducing time-to-market. The rationale for bringing out the products faster is that firms can undercut competition by having considerably earlier release of semiconductor products with cutting-edge technologies to gain a larger market share [27].

One of these challenges is to limit the rising costs of testing VLSI circuits. There exist contemporary tools for producing input test patterns that are scanned into automated test equipment (ATE) from compiling and synthesizing the functional specifications. Furthermore, using the process of cyclization for cycle-based ATE systems, existing tools for automated test pattern generation can produce combinations of input test sequences to verify the IC's integrity against timing issues. However, the tools face increasing amounts of difficulty in synthesizing timing requirements from the timing specifications in a hardware specification language (HDL), such as Verilog and VHDL. To address this issue, iterative and incremental development, which involves design and test, is used to establish such timing specifications [27, 28].

Also, contemporary techniques in testing the application-specific ICs (ASICs) at the system and interface levels using ad hoc techniques and heuristics are inadequate. The design for testability approach, which improves the controllability and observability of signals in the ASIC, can be used to deal with the challenges that current techniques have yet to overcome. It can also help obtain ASIC solutions that are more cost-effective in a reasonable time to market [29].

Since the quality of the ASIC solution depends on the range of testing being carried out, fault coverage is often used as a metric to determine if a system has been adequately tested. Hence, input test vectors and sequences produced from automated test pattern generators (ATPGs) are evaluated with such metric to help determine the quality of the ASIC design. Without giving due consideration to how the ASIC design can be tested, personnel in quality assurance may have significant difficulties quantifying their progress in

testing, and the quality of the product [29].

The input test pattern generation tools for design-for-test (DFT) software need to handle the aforementioned issues, and deal with faster buses, several clocks, and different manufacturing processes. They also have to mitigate VLSI testing costs by supporting test languages, such as Standard Test Interface Language (STIL) [30, 31], Core Test Language (CTL) [32, 33], and Digital Test Interchange Format (DTIF) [34]. In addition, they must support scan-based and functional testing. Other DFT software tools should deal with analysis of test results on a myriad of ATE platforms during various stages of the manufacturing and post-manufacturing process [27].

Improvements of the current state-of-the-art DFT software tools can ameliorate the exorbitant costs, long duration, and test coverage of existing VLSI testing methodologies. By supporting more than one ATE platform, automated test pattern generators can support different approaches to VLSI testing. This includes the outsourcing of the testing team, strategical testing using a team of specialists, and/or the utilization of a myriad range of ATE for testing various requirements. In addition, as the number of processor cores increase, it is a daunting task to simulate the functions of the System-on-Chip with many input test patterns and verify that the results for these tests are correct. It is also necessary to optimize the duration of such lengthy tests, and compress the enormous size of the input test patterns and test results for complex multi-core systems or asynchronous VLSI circuits [27].

Hence, to avoid problems with misalignment of the clock signals with the loading of data from buses or the storing data into buses, the multi-core IC can be transformed into a single-core problem with one clock domain. This involves cyclization and ATE targeting. Timing characteristics and patterns of devices are obtained for ATEs by processing simulation log files. These timing parameters are subsequently entered into ATEs with the use of scripts to automate the process of data entry. The timing waveforms from the tests performed by the ATEs can be fully or partially reproduced to help visualize differences between waveforms of actual test results and expected test results. Also, the data has to be entered into ATEs using a particular format so that these ATEs can read and process them [27].

Electronics for telecommunication devices have operating frequencies higher than 3 GHz, which most ATEs can effectively test ICs. The use of BIST allows logical functions of digital VLSI circuits to be tested at such high operating frequencies, and the functions of analog and mixed-signal devices in communication circuitry to be tested internally without the use of expensive ATE for the digital, analog, and mixed-signal domains. Next, DFT techniques can provide access to numerous internal points in the ICs [27].

Advances of high-level synthesis in electronic design automation allow Design-for-Test (DFT) techniques to be used in system and high-level design of VLSI circuits. This implies that it will be easier to implement low-level circuitry for selected DFT methods that fulfills

the system's specifications, and is consistent with its high-level design. Consequently, the hierarchical design and its appropriate test structures can be optimized within an iteration of the design cycle. This circumvents the need for an iterative and long approach to develop an IC that meets its timing requirements [35].

Implementing a DFT synthesis system early in the design process, which occurs in between high-level and logic syntheses, also allows the organization to synthesize the circuit design using different technologies from a design source. Thus, the choice of technology (or technologies) that is (are) used to map the design does not require the team to produce a separate design source for each technology. Such designs are specified with register-transfer level descriptions. In contrast, DFT techniques that are synthesized for implementation after physical synthesis are dependent on the process technology. This is a result of selecting a process technology to create the physical layout of the IC and its netlist for simulation, verification, and manufacturing. It requires test structures to be embedded into the ICs as harnesses without affecting the logical functions of the IC. Thus, if certain requirements of the IC design for a particular process technology are not met at the end of each design cycle, another iteration would be required to attempt to meet these specifications [35].

Prior to the insertion of test structures into the IC early in the design flow, an analysis is carried out to determine if the inclusion of such test harnesses will facilitate testing, and seek out the faults and errors of the design. This shall prevent embedment of test circuitry for implementing selected DFT techniques without increasing the IC's ability to yield design and implementation errors. That is, if testability analysis is not carried out, the additional circuitry may take up extra chip volume, decrease the yield of the IC design, and not improve internal test effectiveness and coverage [35].

During the early phase in the design flow, the hierarchical design of the IC using system architecting heuristics can be used to improve its modularity, which decreases coupling between subsystems, survivability, and testability amongst other design objectives [22]. Also, techniques such as "divide and conquer" can be used to split the system design into appropriate partitions at each level [36,37]. These can be carried out before and after the insertion of test structures in the ICs [35].

Several structured DFT techniques for testing non-functional logic include boundary scan, and memory and logic built-in self-test (BIST). By inserting test structures for boundary scan or BIST during high-level synthesis, the high-level design of the IC can be made to facilitate testing at different hierarchical levels by improving controllability and observability. Consequently, test logic structures can be minimized, test programs can be simplified, and autonomous testing can be carried out. Also, it allows the design team to deal with register and I/O transformation so that the IC designed with a DFT approach can be manufactured. While register transformation may include clock gating, verification of implicit initialization,

and multi-bit group affiliations of the inferred registers, I/O transformation must facilitate wrap testing, reduced pin count testing, and sharing I/O resources between functional and test circuitry. Moreover, transformations made to clock gating early in the IC design cycle with DFT can optimized functional and test logic simultaneously. This is better than sequentially performing gating transformations, timing analysis, and clock skew adjustment to consider the extra gate delay on the clock path late in the VLSI design cycle. Also, late in the VLSI design cycle, functional logic has to be optimized for area/volume coverage prior to area minimization of test logic under timing constraints [35].

To seemlessly integrate logic and test synthesis, technology mapping is used to implement the logic circuit with a library of elementary standard cells. However, technology mapping brings about the matching and covering problems. With the matching problem, the computer-aided design tool must match the desired function of a VLSI circuit module and any feasible combination of elementary standard cells. This leads to the covering problem, which is about determining the optimum combination of cells from all feasible combinations to minimize area, delay, and/or power. Finally, it is empirically found that VLSI design with DFT considerations early in the design cycle is expected to lead to smaller area/volume consumption and delay in the IC than considering DFT late in the VLSI design cycle. It also utilizes compact and modular VHDL modules to facilitate reuse of components [35].

A model to facilitate designers and test engineers make the trade-offs involved in deciding whether Design-for-Test (DFT) techniques shall be used is discussed as follows. If DFT methods are utilized, this model also facilitates these engineers to make reasonable trade-offs in determining the final design of the IC and its test strategy. These design engineers strive for IC designs that are simple in architecture, have a capability to process information rapidly, and minimizes power consumption and heat dissipation. On the other hand, test engineers endeavor to obtain an IC architecture that facilitates the identification of design errors, facilitates the monitoring of signal values throughout the IC, and can deterministically produce certain outputs given a known set of inputs [38–41].

For the system engineers and project managers working with these engineers, both sets of perspectives are needed to ensure a profitable IC design. The profitability of this IC design depends on the difference between the revenue generated from sales of this IC design, and the accrued and opportunity costs. Thus, such project managers have to account for costs of expensive design and test software and equipment, labor costs in design and testing, and the duration of design, design verification against product specifications, test input generation, testing, and analysis of test results; they should also take note of the expected yield of various design options, and the test coverage provided by the considered DFT techniques. In addition, they have to account for the absence of cost savings, had a more optimum solution been passed [41].

To make difficult decisions on the trade-offs for the adoption of DFT and the selection of DFT techniques, project managers need to comprehend the relationships between the domains of IC design, manufacturing, and testing. The model described provides an analysis of the benefits and costs involved in the adoption or rejection of DFT. This model's input parameters are the volume of production for the IC design, yield of the manufactured chips, chip area/volume, and testability of the design. Subsequently, a description is provided to indicate how the model can be used to determine if DFT should be adopted, and if DFT is adopted, which DFT technique should be used [41].

In the development of the model for the cost-benefit analysis, significant financial factors influencing the benefits and costs involved in VLSI circuit testing have to be accounted for. Subsequently, the subset of these factors that are affected by the decision to adopt DFT, and any DFT technique or combination thereof, are identified to determine their significance in the interaction of these factors. The author notes that the cost determination for IC testing a linear summation of costs involved in generating test input and setting up the test environment, and test execution time and power consumption. It also includes cost of expensive automated test equipment and their depreciatory costs, costs of the excess material and yield of the ICs in DFT implementation, and costs associated with inefficient and ineffective testing, and poor test coverage. Hence, the correlation between costs associated with anomalies in manufacturing, such as wafer production, and poor quality assurance is not accounted for. For example, poorly manufactured wafers not discarded early in the IC manufacturing process may lead to defective dies being tested, and defective wafers and dies that are not promptly disposed will lead to double counting of costs in this model. Based on this, the author suggests considering the interaction of costs associated with test-related silicon, test execution, and imperfect test quality with the usage of multiple regression in statistical analysis. An allusion to such interaction of the model's parameters is provided in the paper, and needs further investigation [41].

Moreover, the costs of storage space for test input generators and input test sequences should have also been considered in the model for test cost generation. This is because an enormous amount of input test sequences is going to be generated and stored in the IC in DFT implementation, and externally for non-DFT IC designs. Also, the relationship between the capital cost of testing is proportional to the average number of pins on the tested ICs. This implies that the capital cost of testing is also a function of the average area of the tested ICs. The author believes that the square root of the area is used to model the decreasing rate of cost increase as the size of the ICs increase. This is because the large costs of test equipment used for testing large ICs is amortized for testing ICs for various sizes. Whereas, if test equipment is only purchased for testing small ICs, this equipment will have to be undergo a considerable amount of upgrade before testing of larger ICs can

commence [41].

The model is enhanced for circuits that implement DFT technology; it considers the modifications in benefits and costs for employing DFT techniques, and for modifications to the yield of the ICs. For the former category of costs, it is decomposed into extra additive costs associated with DFT chips in terms of excess silicon and manufacturing resources, and multiplicative increase in costs pertinent to input test generation, test execution time, and the costs of test equipment. For the latter category, a new approach is required for benefits and costs modeling, since the gains from using DFT in VLSI circuit design encompasses the domains of design, testing, and manufacturing. Consequently, the benefits from using DFT is modeled as faster time to market, and enhancement of the manufacturing and testing processes. This process improvement occurs when test analyses are used to eliminate defects earlier in the process; unfortunately, no known metric is available for determining the gains in process improvement, or yield learning [41].

This empirical study indicates that DFT should not be applied for VLSI circuits of large die volume or planar size, and a considerable region of uncertainty bounds between enormous ICs that should not implement and small ICs that should implement DFT technology. This is due to the inadequacies of the benefits-costs model in terms of capturing the interaction between its parameters, and the uncertainty related to the development of certain metrics. By increasing the risk incurred in the acceptance of defective ICs, it is more beneficial for ICs of larger volume to implement DFT technology [41].

Finally, the primary message that the authors are trying to convey is that an economic model is necessary to determine the benefits and costs involved in the trade-offs in deciding if the DFT paradigm should be adopted, and if so, which DFT technique or combination thereof shall be used. That is, the model in discussion is incomplete, and needs further improvement as it is based only on empirical study. Also, the temporal relationship between the design, testing, and manufacturing domains also need to be investigated. Similarly, the impact of implement various combinations of DFT techniques on yield learning process, and the effectiveness and efficiency of the testing process need to be further researched [41].

Design for Testability (DFT) in IC design has becoming increasingly more important due to the growing costs and complexity of testing VLSI circuits at different hierarchical levels, such as chip, board, and system level [8]. Also, as the risks of inadequately testing ICs increases, different types of fault models need to be included in testing schemes so that more types of physical defects, and electrical and logical faults can be covered. Examples of some fault models include the stuck-at fault model and delay fault model. The former fault model deals with short circuits between the source and drain of a MOSFET, and open circuits at the drain or source of a MOSFET [8]. The latter model deals with failures to meet timing requirements at the desired speed. Other fault models include the bridging fault

model, which occur when a net is used to replace two other nets with an implementation of a elementary logic function (AND or OR the previous nets), open and break faults [8], and I_{DDQ} faults due to leakage currents [8, 12]. Together, such fault models are used to generate input test patterns, build fault dictionaries [12, 42–45], and determine the presence of faults in circuit analysis [11].

The controllability of a circuit indicates the amount of ease that specific signal values can be set up at certain nodes by assigning certain input patterns to the circuit, while its observability describes how well can a signal at a given node be ascertained by feeding certain inputs to the circuit and observing its outputs [8]. Some algorithms to generate input test patterns for combination circuits include D-algorithm, Path-Oriented DEcision Making (PODEM), and FAN-out-oriented test generation (FAN). For sequential circuits and circuits with reconvergent fan-out, input test pattern generation is considerably more difficult, and requires DFT techniques to facilitate input test pattern generation. Such DFT techniques may fall under the following classes: scan-based techniques, boundary-scan tests [8], built-in self test (BIST) techniques, and current monitoring I_{DDQ} testing [11].

Traditional test techniques involve using test pattern generators to produce sequences of input test combinations, and loading these input test patterns into an automated tester for loading. Subsequently, the expected test outputs are compared with the responses of the automated tester to determine if there is any discrepancy; such discrepancies indicate that errors exist in the VLSI circuit. However, the fault models involved in this traditional approach are inadequate; the usage of more complex fault models will increase the rigor of generating input test patterns [8, 11, 46]. In addition, the input test patterns that are automatically generated may not have sufficient fault coverage, and manual generation of input test patterns would otherwise take up a significant amount of time. Since many input test patterns would have been generated and need to be stored for retrieval by the automated tester, a significant amount of storage devices need to be used to store these input test patterns. Moreover, to improve the time complexity of executing the tests, multiple testers are required to run these tests concurrently to save the amount of time needed to exhaustively run input test patterns [47].

Built-in self-test techniques are employed to automatically generate input test patterns, execute tests with these inputs, and ascertain if the output signals are correct. However, there is considerable difficulty in determining the fault coverage for pseudo-random generated input test patterns to test sequential and combinational circuits, and in the concurrent analysis of test results. Consequently, exhaustive testing is used to guarantee complete test coverage. To avoid long input test sequences, the circuit is recursively partitioned into segments/partitions so that the input test sequences for each partition is manageable [47].

As more and more transistors can be packed on an IC, due to decreasing size of transistors,

clock frequency of ICs are also expected to increase exponentially. Also, the number of I/O pins of the ICs will increase. Unfortunately, power consumption will also ramp up. Therein lies the challenge of developing or improving Design-for-Test (DFT) and testing methodologies to cope with this exponential growth in VLSI technology so that ICs can be shipped to customers with a low defective rate. For example, testing equipment will have to improve tester speed and overall timing accuracy to cope with decreasing cycle times. Else, these testing equipment will not be able to accurately test manufactured ICs [48].

As manufacturing costs per transistor in ICs decrease, the costs of depreciation for test equipment amortized over each transistor remains fairly constant. Thus, if this trend continues, the cost of manufacturing a transistor will keep reducing and approach the testing costs of a transistor. Exacerbating the problem is the steep rise in the price of automated test equipment, which is a result of increasing I/O pins and complexity of the ICs. Other challenges facing the electronics industry at the ultra deep sub-micron (UDSM) scale are heat dissipation, electromagnetic interference, and decreasing noise margin that is due to larger currents and smaller voltages. In addition, the effects of parasitic inductance, capacitance, and resistance in interconnects will dominate gate delays. A suggested solution is to modify the testing paradigm for fault modeling and coverage, current handling, test accuracy, and defect isolation [48].

In fault modeling, new failure modes in the solution space for UDSM design have to be captured by contemporary fault models. For example, the prevalence of capacitive and inductive coupling in UDSM VLSI design results in the dominance of interconnect delay over gate delay [46, 49]. Other failure mechanisms include defects in signal routing (bridge defects), and delay-related and speed-related defects. This requires the development of cost-effective models to capture the complexity of these new failure modes, without requiring too much computational time to model and process information [48].

Testing methodologies have to handle the combination of increasing switching and background currents, and off currents (I_{DDQ}) in current handling. It also has to deal with improving power supply to help control noise and power delivery. For improving test accuracy, accuracy of testing equipment needs to match device speed so that the output timing signals associated with the test equipment can be controlled. Also, the phase-locked loops (PLLs) in these test equipment must be designed for these speeds with minimum clock jitter [48].

Finally, the challenges facing VLSI testing and DFT in electronic design automation and testing equipment should not impede the progress of the semiconductor and its related industries. That is, improvements in VLSI testing and DFT must keep up with the exponential growth of electronic design, semiconductor packaging, and silicon process technology. Also, the growth rate of VLSI testing and DFT should match that of desired VLSI testing and DFT technologies, which needs to be developed in time for use [48].

6 Built-In Self-Test (BIST)

6.1 Software BIST

Built-in test (BIT) mechanisms for software provide a methodical approach to internally verifies the functions of software components, and their interaction with other components. This helps to reduce the need for performing those tests using external software testing tools that may add to run-time overheads, and simplify the setup of internal tools to test the software; this is analogous to built-in self-test (BIST) mechanisms in very large scale integrated (VLSI) circuits [29,50,51]. Since these test and verification structures are required to be incorporated into the design of the software, the software design must include such considerations in the development of the software architecture; see section 5.1 on Software DFT. Also, the facilities to test preconditions, postconditions, and assertions that are based on mathematical models shall be inserted into software without modifying the original code [6].

Besides, these built-in test facilities shall also be independent of the data structures used by the components, and be performed as separate operations from the behavior of the components. Furthermore, when these tests fail, the cause of the failures are dealt with by the appropriate functions so that these software faults will not propagate to other software components during software execution. This enables the internal state of the software components to be observed as well. All of these occur without client software components detecting their presence, as the test mechanisms should not interfere with the functions of the components; client software components are those that interact with the software components that are under test. Again, this is comparable to the design for test paradigm in VLSI design [6, 27, 35, 47, 48, 52].

Since the software developers only have to provide a simple interface for these built-in test mechanisms, automated testing facilities can be generated, and executed without interfering with the system and detect undesired interactions. With its flexibility to handle different system development and integration methodologies, it can be cost-effectively used to improve software testing if automated test-case generation is carried out effectively to cover a significant range of input data sequences, test formats, and test patterns [6].

6.2 Hardware BIST

When contemporary integrated circuits are tested externally, their execution cannot be adequately examined to determine their quality and performance. This is due to the enormous number of transistors in the circuit, and the limited number of external I/O pins available to access all the signals in the IC. In addition, the external testing that is carried out at each stage of the production process may increase the risks of damaging the IC, such that

more testing would have to be carried out to ensure that these faults are corrected or do not affect the remaining ICs that are ready for shipping [51].

BIST methodology can provide adequate test coverage without substantial increase in time to market, chip area and volume, and design and production costs. It uses extra circuitry to execute tests internally so that the values of signals can be acquired and measured, and analyzed for possible defects. Hence, it does not have to rely on exorbitantly priced and cumbersome external automated test equipment and tools, such as logic analyzers and digital oscilloscopes, to access specific interconnects in various metal layers. Nor would it be limited by the data transfer speed and bandwidth of interconnects and communication cables of the attached external equipment to access adequate information and many different points in the IC. Finally, at frequencies of gigahertz and beyond, the I/O signals between the IC and the external automated test equipment may get distorted from electromagnetic interference, such as crosstalk noise [51].

As the size of the wires is in the order of several or tens of centimeters, they have a wavelength corresponding to the following [53]:

$$c = f \cdot \lambda$$

Rearrange the equation to yield:

$$f \quad = \frac{c}{\lambda}$$

Substitute example values of the length of wires, which are 1 cm and 10 cm, used to connect testing equipment to the IC package:

$$f_1 = \frac{3.0 \times 10^8 \ m/s^2}{0.1 \ m} = 3 \text{ GHz, and}$$
$$f_2 = \frac{3.0 \times 10^8 \ m/s^2}{0.01 \ m} = 30 \text{ GHz}$$

ICs used in electronics for wireless local area networks (WLAN) and satellite communications have respective operating frequencies in the range of gigahertz or tens of gigahertz [54, 55]. For the testing of ICs used in such electronics in the gigahertz range of operating frequencies [56], the wavelengths of emitted electromagnetic waves from such devices, which is between 0.01 m to 0.1 m, is approximately the same length as connecting wires between the ICs and test equipment. Hence, the parasitic inductive and capacitive effects of these wires will add to the surrounding electromagnetic interference on the IC, and further complicate the analysis of the IC to ensure strong signal integrity in the communications between the IC and the test equipment. In addition, the propagation of electromagnetic waves from surrounding communication systems, like WLAN networks, may add to the noise and distortion affecting the tests on the ICs, if shielding from unnecessary electromagnetic waves is not properly carried out [51, 53].

Moreover, BIST improves the ease of configuration management by reducing the amount of information needed to be maintained for test results pertaining to each hierarchical level of design, version of developed hardware, and the configuration of tools and components used under specified operating conditions. The test results obtained from BIST implementations can be stored in memory devices or overwritten. Consequently, information such as the type of software and firmware used, and the benchmarks used to evaluate the ICs' performances, configuration of tools, and version number can be omitted from data storage. Lastly, standard intellectual property (IP) blocks used for the BIST of specific components can encourage reuse to lower development and testing costs, while providing control of what test functions are carried out. The control of which set of test functions are employed will ensure that a certain amount of test coverage is provided [51].

The scan interface embedded into ICs in BIST sequentially transfers data between between the registers in the IC and the IC's external interface. This allows test results from BIST circuitry to be transferred to external testing equipment. Next, by providing information on the specifications on the IC in the data input to the IC's I/O pins, the IC's internal test generators can generate adequate amounts and variety of test input patterns and test functions to obtain desired test coverage.

Also, the internal analysis and temporary storage of test results can simplify the connection of test equipment to the IC to determine if they meet specifications. For example, in scan chain testing, a series of shirt registers (scan cells) are used to sequentially access to devices under test (DUT). Such scan cells will receive input from the test generators to feed these test input through the logic in the IC. Consequently, the output from various logic blocks are placed into selected scan cells are data analysis. Thus, scan-based testing allows the user to observe values in registers and logic blocks by the provision of virtual probes into them. However, scan-based testing runs in a time complexity order of $O(n)$. To improve its computational complexity, more scan cells can be added internally to provide more access to various points in the IC without modifying the I/O interface [51].

Standards developed for test access ports enable boundary scan to be carried out between electronic components at different levels, ranging from components on the printed circuit board (PCB) to the cellular/modular level in ICs. While they offer a cost-effective approach to testing interconnection between modules without burdening the I/O facilities of the IC, it is slow because the standard interface that was originally designed for functional board level testing of interconnects between chips on PCBs [52, 57–59]. Also, the flow of control signals used to facilitate the communication of modules during testing may use the same buses as normal data flow in the IC, and the boundary-scan control device may serially connect boards in the scan-chain. These may cause strong competition for bandwidth in the data and control buses, and impede boundary-scan testing and performance of the IC [60]. Lastly, the

availability of emulators, field-programmable gate arrays (FPGAs), and application-specific ICs (ASICs) enhances the uptake of boundary scanning when more internal registers can be used to test logic components to speed up testing while avoiding the usage of expensive external test equipment [51].

For BIST to be economically feasible, test pattern generation must be included in the IC to produce test input sequences and combinations to exercise the logic components under operating conditions. Test pattern generations based on pseudo-random processes (PRTPG) can deterministically reproduce test stimulus to exercise all components in the IC with approximate equivalence. In addition, analysis of the test results are made with comparisons of these results with expected outputs based on input test patterns. Due to the large number of test input patterns being executed and test data to be collected, these test results are compressed using signature-generation hardware to capture only the important details so that excessive memory space in the IC is not used to store these results. A number of such compressed test results, which are also known as signatures, and expected outputs can also be concurrently compared using multiple-input signature register (MISR) hardware. Finally, the problem of aliasing a compressed test result with test failures into an error-free signature is one in 2^n, which is insignificant if the data compression technique did not reduce the number of representative bits, n, into a small number [51].

Scan-based BIST of logic structures can be implemented by connecting PRTPG to the inputs of such logic structures, and MISR to their outputs. These scanning techniques in VLSI testing are able to detect up to 90% of errors using the pseudo-random generated test inputs. However, some hardware faults are difficult to be uncovered with these pseudo-random generated test inputs. Consequently, extra connections are made from locations in the IC where these faults may occur to the scan cells so that observability and controllability can be improved. That is, these response of these additional points in the wiring of the IC to the executed test cases can be determined by analyzing test results in the scan cells. Also, the pseudo-random generated test inputs would be able to elicit a response at such points when test inputs are driven into the logic structures to produce expected outputs [38–40]. Since BIST is affordable, more test patterns of a larger variety can be carried out to detect specific hardware faults that are more likely to occur. Consequently, the extra testing that is carried out in BIST, compared to external testing with automated test equipment, can uncover unexpected outputs as well [51].

Test input sequences for memory devices can be deterministically generated to have regular structures so that a new input pattern will overwrite an old pattern that had already been analyzed. Since these test patterns are regular in structure, their expected outputs can be generated in real-time for RAM devices when required by a finite-state machine implementation. This reduces the demand to store expected test results in a compressed

format [51].

Contemporary tools in electronic design and test automation have facilities for specifying the behavior and functions of the BIST blocks. The specifications for their functions and behavior can be specified in a system modeling language, or programming language. They are subsequently synthesized into designs at the register transfer level (RTL), before physical synthesis in IC layout. Hence, the inclusion of BIST blocks in the front-end VLSI design process avoids needless amounts of iterations between the front-end and back-end phases of the design process to meet timing, area, power, and heat dissipation requirements [51].

Finally, the economical value of employing BIST in VLSI circuit design depends on the seamless integration of BIST design into the IC design process. As such, BIST technology shall be able to test complex VLSI designs at the system level. Advancements shall also be made to provide BIST for mixed-signal and analog ICs, and to monitor different circuit parameters pertaining to deep-submicron (DSM) design. This will vastly reduce the need for expensive external test equipment, and greatly reduce costs associated with testing [51].

The usage of built-in self-test (BIST) in VLSI circuit design can also improve the effectiveness of testing integrated circuits (ICs) to meet stringent requirements in a shorter time to market. Reusable and modular hardware components are built into the integrated circuits to perform internal testing of the software. It is also capable of facilitating the internal diagnosis of such test results and correct detected faults [50].

Automating the verification and testing of electronics using BIST enables VLSI circuit designers to reduce time taken to develop, verify, and test the IC. It enables IC verification and/or test engineers to perform other tasks when test automation is being carried out on a system or module. Since these BIST components are modular and generic, they can be combined differently to meet specific needs for cost reduction. That is, new components for basic BIST design does not have to be generated for each IC project [50].

Also, BIST components allow signals at various points in the circuit to be acquired for analysis, without requiring cumbersome automated test equipment that are externally connected to the IC. Consequently, time and effort required to set up the test facilities with the components in the IC can be conserved. In addition, BIST components enable ICs to have robust error recovery mechanisms; such components can detect faults, and repair these errors. Hence, the costs to maintain ICs with BIST components that facilitate error-recovery will be decreased with the those of error diagnosis and error-correction [50].

The test pattern generator consist of the following modules: a test pattern generator, output data compactor, and a test controller. The test pattern generator creates test input vectors and sequences for the BIST components to execute. Next, the output data compactor creates distinctive indicators for the success or failure of running the test cases. Subsequently, it compresses these data in terms of memory space over a shorter period of time. Together,

they are used to determine the architecture of the BIST. Finally, the text controller is used to initiate the tests, and examines the test output for anomalies from expected output. The control module also facilitates hierarchical testing by testing each high-level module, which in turn triggers tests for low-level modules and their submodules. This enables components of the IC to be tested with the BIST technique at each level in the hierarchy [50, 56].

Metrics for determining the effectiveness of using BIST modules should cover the fault coverage of the sequences of input test vectors, the increase in the chip area, and heat and power consumption due to the addition of BIST components. Moreover, the yield of the manufactured ICs due to the extra area, and heat and power consumption have to be factored in, along with timing performance issues. For example, the addition of BIST components may affect the routing and placement of cells in the IC, and consequently affect the yield of the ICs. Lastly, while there are extra overheads in terms of area and power consumption, the savings in time and effort for testing during the manufacturing process is more significant [50].

BIST is highly dependent on the architecture of the IC; it has been used successfully in ICs with regular structures such as RAM, ROM, and other memory devices. The regularity of such devices and the resultant simple design, and the use deterministic algorithms to test for design and manufacturing errors result in a high test coverage. Also, such devices cannot be adequately tested for errors using external equipment [50].

In testing digital logic blocks, deterministic stored-pattern BIST and pseudo-exhaustive BIST trade-off fault coverage for memory space and execution time of the tests. Consequently, if circuits are suitable for pseudo-random testing, pseudo-random BIST is used to to provide better trade-offs for fault coverage over area and performance overheads. A circuit can be made more appropriate for pseudo-random testing by biasing the probabilities of bits in input vectors being set high or low. Signals at certain points in the IC can also be probed to improve the amount of control and observation on the signals of the IC; this facilitates the detection of elusive faults in pseudo-random testing by inserting random patterns to make them more emergent [50].

By executing the tests with the BIST components at the same clock frequency as the IC's system clock, the ease of identifying clock timing errors is improved [56]. However, all of these tests cannot be executed in parallel due to constraints in the heat dissipation and temperature gradient between adjacent regions of the IC. Other constraints include the structure and placement of the modules, and the execution time of running tests on them [50].

Some practices used in design of modules in ASICs to facilitate BIST include level sensitive scan design (LSSD), avoidance of asynchronous circuitry, nesting sequential circuits, feedback loops, internal tri-state buses, and boundary scan testing. LSSD uses level sensi-

tive latches and flip-flops to replace storage devices so that the feedback loops is logically covered. This results in the external transformation of the networks logical behavior from a sequential circuit to a logical circuit. Lastly, boundary scan determines how testing is carried out, and has an architecture that facilitates the integration of testing modules into the I/O region to speed up testing of its I/O interface, intra- and inter-module communications, and functions [29].

7 Application of Design of Experiments in Test Automation

7.1 Usage of Design of Experiments in Software Engineering

Testing in software engineering consists of executing test suites with varying inputs to the software so that this software is tested for different test cases. Subsequently, the output from the software or its behavioral response can be verified to determine if the software is functioning correctly. Techniques in design of experiments (DOE) can be used to develop a robust software testing process, where external sources of variability can be reduced with the aid of software test automation [61].

Using DOE techniques, the software quality assurance engineer can determine the parameters that influence the behavior of the software and its output response. This engineer must also determine the factors and external environmental conditions that should be controlled in software testing. In addition, the engineer must ascertain how should the software, and its modules and sub-modules be tested. Also, this engineer has to find out how to analyze the data obtained from executing the tests, and be able to store the test results for quick and easy retrieval [61].

When scripts are written to automate software testing so that human interaction with the software can be eschewed, the performance of the software is influenced by the hardware (type of processor, and its memory and input/output devices) that it is executed on. Also, if the software is made to be executed as part of an embedded computing system, it depends on the external environment in which the embedded systems is placed in. Some of these input factors such as test input parameters and hardware can be controlled, while the external environment that embedded computer systems are used in operation cannot be controlled. Hence, the controllable factors of the software that exhibit the largest influence on the output of the software or its behavioral response can be identified. It allows these controllable factors to be adjusted so that the variability of the software's output response or behavior is minimized. In addition, these controllable factors can be modified to minimize the effects of uncontrollable parameters [61].

For a software with k factors, the exhaustive testing of this software will require m^k test executions with unique input combinations, or input parameters and environmental factors. The reason for the base m is because a factor may have m different levels of influence in the software's function and output response. Consequently, fractional factorial testing is a DOE technique that can be used to minimize the number of test executions during software testing. It does this by selecting a subset from the list of exhaustive test executions that can be run. This subset excludes aliases high-order interactions, as significant interactions between most factors that influence the software is unlikely. Hence, the amount of time and resources spend on testing can be reduced [61, 62].

Fractional factorial testing is used to determine the main factors and interactions influencing the software's output response. As alluded to, it also considers possible interactions between different factors that can affect the software's performance or behavior [62]. Ignoring such interaction of factors will result in incorrect attribution of software behavior and function to the factors. Moreover, aliases arise from estimating the influence of two or more main factors because it repeats the estimation of some interaction terms between these factors. Thus, they should be removed from the regression model used in fractional factorial testing. As an aside, in statistical design and analysis of experiments, it is often known as fractional factorial experimental design [61].

Next, fractional factorial testing can also be used to evaluate and compare the performance of different software architectural designs and algorithms. That is, it can be used to consider design alternatives, and optimize the performance of the software. The selection of input test parameters and their interactions can be used to determine if the software is robust to the variety of test input patterns. Also, the mean and standard deviation of the response variables give adequate indication of the software's output response and behavior. Thus, the metrics and methods to measure the response variables should adequately indicate the minute differences in the response variables, so that small effects from the manipulation of controllable factors can be determined. As the regression model for fractional factorial testing gets refined with each test execution, insignificant factors in the regression model can be removed. The tests are executed again to verify that the removed factors were insignificant, and barely affected the software's behavior and function [61].

Taguchi's method can be used to develop software that is robust to environmental conditions and modification of its software components. It can also be used in software development to minimize the variation of the software's behavior and functions around the desired/expected objectives. For example, orthogonal arrays for main factors and their interactions can be set up so that variation between uncontrollable factors can be minimized. This allows the usage of controllable factors that are least sensitive to variations in the uncontrollable factors. Thus, by selecting appropriate metrics for the response variables of the

software system, the interactions between controllable and uncontrollable parameters can be minimized, or rendered insignificant. However, Taguchi's method has been proven to be inefficient, and does not help provide insight into the software development and quality assurance processes [61].

7.2 Usage of Design of Experiments in VLSI Circuit Design

In DOE for VLSI circuits, a model for measuring the performance of the circuit can be developed to capture designable parameters and noise parameters, and their interaction terms. These parameters include factors that are comprised of controllable and uncontrollable variables [62]. By using such computationally efficient performance models with appropriate degrees of accuracy, exhaustive and long simulations of the circuit can be eschewed. For example, a fractional factorial design (including Taguchi's Orthogonal Arrays) or central composite design in DOE is used to determine the minimum number of simulation runs for the verifying the circuit. Also, Latin hypercube sampling (LHS) is a sampling method that can be used to select input test patterns for simulations to verify the performance of the ICs, while providing a greater coverage of all possible input patterns [11].

In parametric yield estimation, the direct Monte Carlo and performance model methods can be used to describe the how well can this IC design be manufactured, and be able to meet acceptable performance criteria despite random variations in the fabrication process. In addition, Monte Carlo-based methods and geometrical methods can be used to maximize parametric yield by adjusting the controllable variables in its function. In the former method, circuit simulation is used to evaluate circuit performances for a given set of controllable variables, along with constructing analytical response surface models. This enables the acceptable region of the circuit parameter space to be determined so that the optimum combination of circuit parameters can be determined. For the latter method, geometric approximation, such as the use of simplex method, or analytical models of the circuit performance, such as response surface models, is used to estimate the acceptable region of the circuit parameter space. Subsequently, the design centering technique is used to select a design solution that lies in the middle of this acceptable region [11].

In worst-case analysis, tolerances for the semiconductor fabrication process are determined by determining the worst values for the performances of the circuit under stochastic manufacturing, chemical, and material processes. In addition, the worst-case values for each uncontrollable variable, or noise parameter, can be determined under these statistical fluctuations; this vector of uncontrollable variables can be used to ascertain that circuit performances are acceptable with such variables. Also, by determining the best, worst, average, and nominal noise conditions for the uncontrollable variables, IC designers can approximate the circuit's range of random variations in its performance. The corners technique

is commonly used in worst-case analysis, and assumes that the noise parameters can be varied independently to determine how sensitive each noise parameter is to random variations. Hence, if an IC can be designed to meet all of its requirements with a vector of noise parameters set to their worst values, this IC design would improve its parametric yield considerably. Consequently, this results in conservative designs that create bottlenecks for further optimization and augmentation of DFM techniques [11].

Stochastic process variations in semiconductor fabrication and random noise fluctuations can be minimized to reduce their impact on the performance of the circuit. The variance of such random variations is used to quantify the extent to which the circuit's performance has been affected. This variance is a function of controllable variables, so different values of these controllable variables can be selected to ascertain the smallest variance, or nominal random variations. Consequently, this leads to optimizing the circuit design for multiple criteria since a good IC design needs to be selected along with a good circuit performance; that is, an optimum selection of IC design and circuit performance needs to be made [11].

8 Conclusions

Several techniques related to software and VLSI test automation have been discussed. A comparative study on the similarities and differences of these techniques is provided. Systems architecting heuristics can be used to design the architecture for a software or VLSI circuit. Also, design for testability techniques, such as built-in self test, can be included in the architecture design to improve and facilitate test automation. Lastly, design of experiments can be used to reduce the number of test execution runs to save time while achieving an adequate test coverage.

A Appendix: A Guide to the Software Engineering Body of Knowledge

A guide to the software engineering body of knowledge (SWEBOK), which is based on consensus, is being developed by the Institute of Electrical and Electronic Engineers (IEEE) and the Association for Computing Machinery (ACM). Its goal is to facilitate the development of software engineering as a profession by providing knowledge that is expected from people entering the industry. From this core set of skills and knowledge that budding software engineers should have, professional organizations in software engineering can set the syllabus for examinations that are held for the acquirement of professional licenses, and set the standards for accrediting program curricula of universities that provide degrees in software engineering [63].

An overview of the project status and its progress thus far is provided. The SWEBOK guide hierarchically breaks down the set of core knowledge and skills into several fields that software engineering students and professionals should be acquainted with. In turn, each of these fields are broken down into topics of interest within that field; each topic provides sufficient a description for students, educators, and professionals to find reference material about it. Consequently, topic description will not be biased towards certain software development methodologies, business models, styles of management, and areas of application [63].

The relevant area of knowledge for this report is about software testing, where the program is executed to ascertain correct functioning for a selected set of scenarios in testing. Some of the interesting subareas discussed include test automation and various techniques used in testing [63].

B Appendix: Application of Automated Software Testing in Academia

Test automation is used in the grading of submitted student programming assignments to save time required to manually verify if the students have achieved the objectives set out for the assignments. In addition, the faster turnover rate for graded assignments lead to prompt feedback to students that can be used to improve their performance; student feedback can be provided at the time of submission to warn them of compiling errors and failure to meet particular test cases. These automated test suites have a structure that uses the specifications of the assignment to determine if the student has fulfilled them. Consequently, based on how satisfactory have the students fulfilled the requirements, an appropriate grade is awarded to them [64].

However, this implies that instructors have to spend a considerable amount of time designing specifications for an assignment so that scripts for automated testing can be written for them. The amount of effort spent on this will result in tremendous savings in the time that would have been spent manually grading each submitted program. When test automation is used, test scripts can be executed concurrently to grade different sets of assignments without requiring user interaction. Subsequently, the scripts may include commands to populate and update a database that keeps track of the students' information, such as their names, student numbers, the number of specifications fulfilled, the requirements that were not met, and grades. Therefore, the process of disseminating the specifications for assignments, collecting submitted student programming assignments, grading these assignments, providing assignment feedback, and maintaining a database of their progress can be automated [64].

In fully automated testing of student programs, the test scripts may require graders to end programs that have excessive execution time. These scripts can also incorporate built-in self-tests for checking against infinite recursion and loops so that the test scripts will not be executed perpetually without user interaction. In addition, these test scripts have to determine if the specifications for the assignments are met and how much of the specifications that have been fulfilled, such that the test automation process is repeatable and can be checked with reports for audit. To do so, the requirements for the input, output, and functions for the programs have to be unambiguously written to convey information regarding the data types to be used, acceptable range and format of input and output values, user interaction sequence, and how it should perform the given operations [64].

The test plans define how to approach the evaluation of the program during testing; the program should be evaluated based on its function and performance under circumstances that the program is expected to executed correctly, and circumstances that may or will cause the program to fail. Following the design of test plans, test data set have to selected from an enormous amount of possible test inputs that can be entered into the students' programs. Subsequently, test cases have to be developed to provide adequate, irredundant, and effective and efficient test coverage for the specifications of the assignments. Each of these test scenarios indicates the expected behavior in reaction to some given inputs. From this, a plan for assigning grades based on certain inputs for selected test cases can subsequently developed to assess whether the student(s) has(/have) demonstrated adequate comprehension of the class material to apply those acquired concepts. The grading plan will determine how many points have the student accumulated for satisfying each specification of the assignment [64].

This specification driven test automation framework for grading student assignments helps to save time and effort. Initial resistance to automated grading of assignments can be overcome when students learn and appreciate the importance of conforming to the specifi-

cations of the assignments, and when they utilize the prompt and detailed feedback of the automated grading system to. An interesting and challenging avenue for future work involves the automated generation of test scripts from the specifications of the assignments [64].

References

[1] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70–79, Jan/Feb 2000.

[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, December 1997.

[3] R. Black, *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, 2nd ed. New York, NY: John Wiley & Sons, 2002.

[4] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, ser. Object Technology Series. Upper Saddle River, NJ: Addison–Wesley, 2000.

[5] F. P. Brooks, Jr., *The Mythical Man–Month: Essays on Software Engineering.* Reading, MA: Addison–Wesley, 1995.

[6] S. H. Edwards, "A framework for practical, automated black–box testing of component–based software," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 97–111, June 2001.

[7] D. Hoffman, "Test automation architectures: Planning for test automation," in *Proceedings of the* 12^{th} *International Software Quality Week Conference.* San Jose, CA: Software Research Institute, Inc., 24–28 May 1999.

[8] J. M. Rabaey, A. Chandrakasan, and B. Nikolić, *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Upper Saddle River, NJ: Pearson Education International, 2003.

[9] P. J. Ashenden, *The Student's Guide to VHDL.* San Francisco, CA: Morgan Kaufmann, 1998.

[10] C. Kaner, "Cem kaner on rethinking software metrics: Evaluating measurement schemes," *Software Testing & Quality Engineering*, vol. 2, no. 2, pp. 51–56, March–April 2000.

[11] S.-M. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits: Analysis and Design*, 3rd ed. New York, NY: McGraw–Hill, 2003.

[12] R. C. Aitken, "Modeling the unmodelable: Algorithmic fault diagnosis," *IEEE Design & Test of Computers*, vol. 14, no. 3, pp. 98–103, July–September 1997.

[13] N. S. Eickelmann and D. J. Richardson, "An evaluation of software test environment architectures," in *Proceedings of the* 18^{th} *International Conference on Software Engineering, 1996.* Berlin, Germany: IEEE Press, 25–30 Mar 1996, pp. 353–364.

[14] P. Stevens and R. Pooley, *Using UML: Software Engineering with Objects and Components.* New York, NY: Addison–Wesley, 1999.

[15] W. Pree and H. Sikora, "Design patterns for object–oriented software development," in *Proceedings of the* 19^{th} *International Conference on Software Engineering, 1997.* Boston, MA: ACM Press, 17–23 May 1997, pp. 663–664.

[16] D. J. Ram, K. N. A. Raman, and K. N. Guruprasad, "A pattern oriented technique for software design," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 70–73, July 1997.

[17] T. Mikkonen, "Formalizing design patterns," in *Proceedings of the* 20^{th} *International Conference on Software Engineering, 1998.* Kyoto, Japan: ACM Press, 19–25 April 1998, pp. 115–124.

[18] J. Bosch and P. Molin, "Software architecture design: Evaluation and transformation," in *Proceedings of the IEEE Conference and Workshop on Engineering of Computer–Based Systems, 1999 (ECBS '99).* Nashville, TN: IEEE Press, 7–12 March 1999, pp. 4–10.

[19] A. L. Guennec, G. Sunyé, and J.-M. Jézéquel, "Precise modeling of design patterns," in *Proceedings of the* 3^{rd} *International Conference on the Unified Modeling Language: Advancing the Standard, 2000 (UML 2000)*, ser. Lecture Notes In Computer Science, A. Evans, S. Kent, and B. Selic, Eds., vol. 1939. York, UK: Springer–Verlag, 2–6 October 2000, pp. 482–496.

[20] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural styles, design patterns, and objects," *IEEE Software*, vol. 14, no. 1, pp. 43–52, January/February 1997.

[21] D. Gross and E. Yu, "From non–functional requirements to design through patterns," *Requirements Engineering*, vol. 6, no. 1, pp. 18–36, February 2001.

[22] C. S. Wasson, *System Analysis, Design, and Development: Concepts, Principles, and Practices*, ser. Systems Engineering and Management. Hoboken, NJ: John Wiley & Sons, 2006.

[23] P. Alencar, D. Cowan, T. Kunz, and C. Lucena, "A formal architectural design patterns-based approach to software understanding," in *Proceedings of the* 4^{th} *Workshop on Program Comprehension, 1996.* Berlin, Germany: IEEE Press, 29–31 March 1996, pp. 154–163.

[24] H. Kim and C. Boldyreff, "Software reusability issues in code and design," *ACM SIGAda Ada Letters*, vol. XVII, no. 6, pp. 91–97, November–December 1997.

[25] J. A. McDermid, "Complexity: concept, causes and control," in *Proceedings of the* 6^{th} *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000), 2000.* Tokyo, Japan: IEEE Press, 11–14 September 2000, pp. 2–9.

[26] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel, "Design patterns application in UML," in *Proceedings of the* 14^{th} *European Conference on Object-Oriented Programming, 2000 (ECOOP 2000)*, ser. Lecture Notes In Computer Science, E. Bertino, Ed., vol. 1850. Sophia Antipolis and Cannes, France: Springer–Verlag, 12–16 June 2000, pp. 44–62.

[27] H. Lam, "New design-to-test software strategies accelerate time-to-market," in *Proceedings of the* 29^{th} *IEEE/CPMT/SEMI Electronics Manufacturing Technology Symposium, 2004.* San Jose, CA: IEEE Press, 14–16 July 2004, pp. 140–143.

[28] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *IEEE Computer*, vol. 36, no. 6, pp. 47–56, June 2003.

[29] N. Shastry, "Tutorial on design for testability," in *Proceedings of the* 5^{th} *Annual IEEE International ASIC Conference and Exhibit, 1992.* Rochester, NY: IEEE Press, 21–25 Sep 1992, pp. 139–142.

[30] T. Taylor and G. A. Maston, "Standard test interface language (STIL): A new language for patterns and waveforms," in *Proceedings of the International Test Conference, 1996.* Washington, DC: IEEE Press, 20–25 October 1996, pp. 565–570.

[31] *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data*, Test Technology Standards Committee of the IEEE Computer Society Std. IEEE Std 1450-1999, March 1999.

[32] R. Kapur, B. Keller, B. Koenemann, M. Lousberg, P. Reuter, T. Taylor, and P. Varma, "P1500-ctl: Towards a standard core test language," in *Proceedings of the* 17^{th} *IEEE VLSI Test Symposium (VTS '99), 1999.* San Diego, CA: IEEE Press, 25–30 April 1999, pp. 489–490.

[33] R. Kapur, M. Lousberg, T. Taylor, B. Keller, P. Reuter, and D. Kay, "CTL the language for describing core-based test," in *Proceedings of the International Test Conference, 2001.* Baltimore, MD: IEEE Press, 30 October–1 November 2001, pp. 131–139.

[34] *IEEE Standard for Digital Test Interchange Format (DTIF)*, IEEE Standards Coordinating Committee 20 on Test and Diagnosis for Electronic Systems Std. IEEE Std 1445-1998, December 1998.

[35] V. Chickermane and K. Zarrinch, "Addressing early design-for-test synthesis in a production environment," in *Proceedings of the International Test Conference, 1997.* Washington, DC: IEEE Press, 1–6 November 1997, pp. 246–255.

[36] J. Bentley, "Programming pearls: Algorithm design techniques," *Communications of the ACM*, vol. 27, no. 9, pp. 865–873, September 1984.

[37] H. C. Cunningham, Y. Liu, and C. Zhang, "Using the divide and conquer strategy to teachJava framework design," in *Proceedings of the* 3^{rd} *International Symposium on Principles and Practice of Programming in Java, 2004*, ser. ACM International Conference Proceeding Series, vol. 91. Las Vegas, NV: Trinity College Dublin, 16–18 June 2004, pp. 40–45.

[38] C. L. Phillips and H. T. Nagle, *Digital Control System Analysis and Design*, 3rd ed. Upper Saddle River, NJ: Pearson Educational International, 1998.

[39] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 9th ed. Upper Saddle River, NJ: Prentice-Hall, 2001.

[40] J. Dorsey, *Continuous and Discrete Control Systems: Modeling, Identification, Design, and Implementation.* New York, NY: McGraw-Hill, 2002.

[41] 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, no. 1, pp. 29–41, January–February 2002.

[42] S. D. Millman, E. J. McCluskey, and J. M. Acken, "Diagnosing CMOS bridging faults with stuck-at fault dictionaries," in *Proceedings of the International Test Conference, 1990.* Washington, DC: IEEE Press, 10–14 September 1990, pp. 860–870.

[43] P. G. Ryan, W. K. Fuchs, and I. Pomeranz, "Fault dictionary compression and equivalence class computation for sequential circuits," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1993 (ICCAD 1993).* Santa Clara, CA: IEEE Press, 7–11 November 1993, pp. 508–511.

[44] T. Lee, W. Chuang, I. N. Hajj, and W. K. Fuchs, "Circuit-level dictionaries of CMOS bridging faults," in *Proceedings of the* 12^{th} *IEEE VLSI Test Symposium, 1994.* Cherry Hill, NJ: IEEE Press, 25–28 April 1994, pp. 386–391.

[45] B. Arslan and A. Orailoglu, "Fault dictionary size reduction through test response superposition," in *Proceedings of the* 20^{th} *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2002 (ICCD 2002).* Freiburg, Germany: IEEE Press, 16–18 September 2002, pp. 480–485.

[46] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd ed. Boston, MA: Pearson Educational, Inc., 2005.

[47] E. J. McCluskey and S. Bozorgui–Nesbat, "Design for autonomous test," *IEEE Transactions on Circuits and Systems*, vol. 28, no. 11, pp. 1070–1079, November 1981.

[48] G. Singer, "The future of test and DFT," *IEEE Design & Test of Computers*, vol. 14, no. 3, pp. 11–14, July–September 1997.

[49] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors, 2005 Edition: Interconnect," Semiconductor Industry Association, Tech. Rep., 2005.

[50] Y. Zorian, "Built-in quality assurance," in *Proceedings of the* 3^{rd} *International Conference on the Economics of Design, Test, and Manufacturing, 1994.* Austin, TX: IEEE Press, 16–17 May 1994, pp. 9–12.

[51] B. Könemann, B. Bennetts, N. Jarwala, and B. Nadeau-Dostie, "Built-in self-test: Assuring system integrity," *IEEE Computer*, vol. 29, no. 11, pp. 39–45, November 1996.

[52] J. M. Miranda, "A BIST and boundary-scan economics framework," *IEEE Design & Test of Computers*, vol. 14, no. 3, pp. 17–23, July–September 1997.

[53] S. Ramo, J. R. Whinnery, and T. Van Duzer, *Fields and Waves in Communication Electronics*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 1994.

[54] D. M. Pozar, *Microwave and RF Design of Wireless Systems.* New York, NY: John Wiley & Sons, 2001.

[55] A. B. Carlson, P. B. Crilly, and J. C. Rutledge, *Communication Systems: An Introduction to Signals and Noise in Electrical Communication.* Singapore: McGraw-Hill, 2002.

[56] Y. Zorian, "A distributed BIST control scheme for complex VLSI devices," in *Proceedings of the Eleventh Annual IEEE VLSI Test Symposium, 1993: Digest of Papers.* Atlantic City, NJ: IEEE Press, 6–8 April 1993, pp. 4–9.

[57] A. Hassan, J. Rajski, and V. K. Agarwal, "Testing and diagnosis of interconnects using boundary scan architecture," in *Proceedings of the International Test Conference: New Frontiers in Testing, 1988.* Washington, DC: IEEE Press, 10–14 September 1988, pp. 126–137.

[58] A. A. Setty and H. L. Martin, "BIST and interconnect testing with boundary scan," in *Proceedings of Southeastcon, 1991*, vol. 1. Williamsburg, VA: IEEE Press, 12–14 September 1991, pp. 12–15.

[59] F. De Jong, J. S. Matos, and J. M. Ferreira, "Boundary scan test, test methodology, and fault modeling," *Journal of Electronic Testing*, vol. 2, no. 1, pp. 77–88, March 1991.

[60] N. Jarwala, C. W. Yau, P. Stiling, and E. Tammaru, "A framework for boundary-scan based system test and diagnosis," in *Proceedings of the International Test Conference: Discover the New World of Test and Design, 1992.* Baltimore, MD: IEEE Press, 20–24 September 1992, pp. 993–998.

[61] D. C. Montgomery, *Design and Analysis of Experiments*, 4th ed. New York, NY: John Wiley & Sons, 1997.

[62] R. L. Mason, R. F. Gunst, and J. L. Hess, *Statistical Design and Analysis of Experiments: With Applications to Engineering and Science*, ser. Wiley Series in Probability and Mathematical Series. New York, NY: John Wiley & Sons, 1989.

[63] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. Tripp, "The guide to the software engineering body of knowledge," *IEEE Software*, vol. 16, no. 6, pp. 35–44, Nov/Dec 1999.

[64] E. L. Jones, "Grading student programs – a software testing approach," *Journal of Computing Sciences in Colleges*, vol. 16, no. 2, pp. 185–192, January 2001.