DESIGN AND IMPLEMENTATION OF AUTOMATION APPROACHES FOR EFFICIENT SOC VERIFICATION

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

VLSI Design

Submitted By

PALLAVI SAINI

6024620014

Under Supervision of

Dr. Hardeep Singh
Associate Professor

Dr. R. S. Kaler
Sr. Professor & Dean (R & D)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
JUNE, 2026

DECLARATION

I, **Pallavi Saini** hereby declare that the work presented in this thesis entitled **"Design and Implementation of Automation Approaches for Efficient SoC Verification"** in partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design)**submitted at the **Electronics and Communication Department, Thapar Institute of Engineering & Technology, Patiala** is an authentic record of work carried out under the supervision of **Mr. Varun Malhotra** (Senior Staff Engineer), STMicroelectronics Private Limited (Greater Noida). The matter presented in this has not been submitted either in part or full to any other university or institute.

Pallavi.

Date: 27 – 06 – 2026

Pallavi Saini
6024620014

Mr. Varun Malhothra Senior Staff Engineer STMicroelectronics Pvt. Ltd. Date: 27 – 06 – 2026	**Dr. Hardeep Singh** Associate Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 27– 06– 2026
	Dr. R.S. Kaler Sr. Professor & Dean (R & D) Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 27 – 06 – 2026

STMicroelectronics Private Ltd.
Plot No.1, Knowledge Park-III,
Greater Noida – 201 308.
Uttar Pradesh, India
Tel : +91-120-4003000
Fax :+91-120-4003007
Email : infostm.india@st.com
CIN : U32109DL1990FTC039906
www.st.com

July 31, 2026

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **Pallavi SAINI** has undergone internship in our **MDRF** Group from **July 02, 2025 to July 31, 2026** and has successfully completed the project on: **SR6P5E, SR6G7E, SR7X8, SR6G5.**

During the internship period, Pallavi was found to be sincere and professional in conduct.

We wish her all the best for the future endeavors.

for **STMicroelectronics Pvt. Ltd.**

Sanjay Kumar PIPLANI
India Talent Acquisition Head

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my thesis supervisor, **Dr. Hardeep Singh** and **Dr. R.S. Kaler**, for their invaluable guidance, unwavering support, and constant encouragement through- out this research. Their expertise and insightful feedback were instrumental in shaping this work. I also extend my thanks to the faculty and staff of the Electronics and Communication Engineering Department at Thapar Institute of Engineering & Technology for providing a conducive academic environment. Finally, I am incredibly grateful to my industry mentor **Mr. Varun Malhotra** for their support and guidance. Thanks a lot, to my mentors for their valuable guidance during the report work, they have given me valuable advice and support which has helped me in understanding the technical aspects of the report. Last but not the least I thank my family, friends, and colleagues for their timely help and valuable suggestions.

Pallavi Saini .

ABSTRACT

The increasing complexity of modern System-on-Chip (SoC) designs has significantly increased the effort required for functional verification, making it one of the most time-consuming phases of the semiconductor design cycle. Efficient management of verification activities such as simulation execution, regression testing, test management, and result analysis is essential to ensure design correctness while meeting project timelines. However, many stages of the verification flow still involve repetitive manual processes, which can reduce productivity and increase the possibility of human error.

This work presents the design and implementation of automation techniques aimed at improving the efficiency of selected stages in the SoC verification flow. Proposed Approach In the presented work, the effort will concentrate on the automation of the relevant verification processes such as simulation execution, regression management and log file or result analysis. The proposed automated techniques reduce the manual work for engineers. Through the implementation of automated scripts and an automated flow, the effort required for the verification can be minimized and the verification productivity can be increased. Within this verification flow, an efficiency gain in single stages is intended by the utilization of automated techniques by eliminating tedious, manual effort and facilitating automated procedures.

Presently, the goal is to set up the automation framework and find critical points in the verification flow to make a beneficial contribution. Ultimately, a result of this project is an optimized verification workflow and simplified verification process management. After the setup, more implementation and integration and measurement of automated solutions in the existing verification flow shall follow in other parts of the project.

TABLE OF CONTENTS

Sr. No	Name of the Chapters	Page No
	DECLARATION ……	*ii*
	CERTIFICATE	*iii*
	ACHNOWLEDGEMENT……	*iv*
	Abstract……	*v*
	List of Figures……	*vii*
Chapter 1	Introduction……	*8*
1.1	Verification and Levels of Verification …….	*10*
1.2	Types of Verification	*12*
1.3	Role of Automation in SoC Verification	*15*
Chapter 2	Microcontroller Safety Concept with ASIL	*17*
2.1	ASIL and ISO 26262 Overview	*18*
2.2	Hazard Analysis and Risk Assessment (HARA)	*19*
2.3	ASIL Classification Levels	*20*
2.4	ASIL Impact on System, Hardware, and Software Development	*20*
2.5	Importance of ASIL in Automotive Safety	*21*
Chapter 3	Literature Survey	*22*
3.1	Challenges in SoC Verification	*22*
3.2	Universal Verification Methodology (UVM)	*23*
3.3	Automation in SoC Verification Flow	*23*
3.4	Emerging Trends in SoC Verification	*24*
Chapter 4	Problem Statement & Proposed Methodology	*26*
4.1	Proposed Solutions	*27*
Chapter 5	Current Work and Implementation	*31*
5.1	Overall Automation Framework	*32*
5.2	Automation of Testcase Generation	*34*
5.3	Automation in Regression Result Analysis	*35*
5.4	RTL Testbench to Tester Testbench Automation	*37*
5.5	Conclusion	*38*
	REFERENCES	*40*

LISTS OF FIGURES

Sr. No	Figure Details	Page No
1.1	*SoC Block*	*11*
1.2	*Coverage vs time in direct stimuli verification*	*13*
1.3	*Coverage vs time in constraint random verification*	*14*
1.4	*Formal Based Verification*	*15*
2.1	*Standard included in IEC 61508*	*17*
2.2	*Phases of ISO26262 serial-wise*	*18*
2.3	*Phase of the ISO 26262 automotive safety lifecycle*	*19*
2.4	*ASIL levels of components present in automotive*	*19*
2.5	*Flow from hazard identification to ASIL assignment*	*20*
2.6	*ASIL levels defined in ISO 26262*	*20*
5.1	*Sample Test Generated Using Automation Script*	*35*
5.2	*Regression Result Analysis*	*37*

CHAPTER 1

INTRODUCTION

The continuous advancement of semiconductor technology has enabled the integration of very large number of components on a single integrated circuit, giving rise to the concept of System-on-Chip (SoC) design. Because more and more functionality can be integrated onto a single integrated circuit as the technology advances, System-on-Chip (SoC) design arises. An SoC chip integrating various functional blocks, such as processor, memory elements, I/O interface, peripheral and so on onto a single chip results in the system with lower cost, lower power consumption and small size. It's widely applied to various fields, for example, cell phone, portable equipment, computer system, automobile and embedded system etc.

As SoCs get more complicated and involve many I/O devices and functional blocks developed by many different groups and integrate these into a common chip, checking the functional accuracy of SoC systems becomes one of the key issues in the semiconductor design field.

Without thorough functional validation, any design errors will eventually lead to re-design that is much more expensive and delay product launch. Therefore, verifying chip design accuracy prior to fabrication is very critical. Verification means making sure the implemented design functions as it is designed. We provide simulation stimuli to the chip under test (CUT), monitors the output response and checks it against its expected behaviour.

Verification refers to ensuring that a design is implemented as specified in the functional specification, by applying stimuli to the design and verifying the response from the design is as intended, given a certain stimuli from the external world. In todays design, verification contributes to one of the highest percentages of time among various stages in a development project. Studies, experiences in the industry show that contribution of verification alone could account to more than fifty percent of the total design effort.

A typical SoC verification cycle consists of different steps that include testbench development, simulation execution, debugging and coverage analysis.

In most of the modern chip design projects, functional verification of the SoC may occupy more than half of the total effort based on different types of verification styles and approaches.

In this verification work-process, we have to simulate thousands of test-cases and generate a huge set of simulation tests, called regression suites, to verify the whole system at various scenarios. Automating simulation process: Simulating SoC models requires generating a lot of stimulus files for it, running thousands of simulations (known as regressions in case of modifications in the original design), observing the outputs and running comparisons. These activities consume a lot of effort when carried out manually.

There may be thousands of test cases to be simulated to obtain complete functional coverage of the SoC. There are several tasks required when performing verification like creating design test environment, developing test-cases for each feature, executing each test-case for the design and comparing the simulation output against golden data or expected output for each test-case, analysing and logging results. A lot of these tasks have to be repeated, and it will cause errors and is inefficient when performed manually, for example, managing of regression, organizing of logs, running regressions in a particular sequence and so on.

To overcome this limitation, automation methods are widely adopted for simplifying and streamlining the chip verification flow.

Automating repetitive tasks saves human effort and reduces error risks. Automation, whether it is script-based or tool-driven, significantly contributes to accelerating and optimizing verification process including simulation running, regression management, result checking, analysis and report generation, etc. This allows the verification engineers to concentrate on higher-value activities such as design bug detection and system analysis rather than low-level execution management tasks. Verification automation, particularly for regressions is also very important in order to reduce the overall verification time required for a complex SoC.

Automated regression systems typically execute test cases automatically, distributed among available computational resources to speed up overall verification completion time.

Other forms of automation in SoC verification: Besides using automation scripts in running the regressions, other methodologies such as employing a structured approach to verification, for instance following Design for Verification (DFV) practices, using well-defined verification IPs (VIPs), employing advanced functional verification techniques (like directed tests, constrained random tests, and assertion-based verification) are commonly used. Even with this help, some steps in the verification work-process remain repetitive and could benefit from a greater level of automation. Some methods of automation, when executed efficiently in the right steps within a verification process, could still boost the productivity by reducing the number of manually executable tasks while still providing sufficient verification flexibility.

Our Work This paper describes design of some methods of automation to simplify and enhance efficiency of SoC verification process. These automation techniques can simplify the way some typical SoC verification tasks are performed in our design flows, and help support improved productivity and manageability.

The remainder of this report presents the background concepts related to SoC verification, discusses commonly used verification techniques and methodologies, and describes the automation approaches intended to enhance the efficiency of the verification workflow.

1.1 VERIFICATION AND LEVELS OF VERIFICATION

Verification is the biggest challenge for companies which are working on IP and SoC products to meet the customer requirements. The complex in recent technology circuits is very high which leads to high time for verification which can be done only through the application of advanced verification techniques and optimal reuse. It is noted out that almost 70% of the entire chip design time is devoted to verification. Prior to moving the expensive chip for manufacturing, all bugs and errors in the design must be eradicated. This is the elementary principle and inspiration for verification. Levels of verifications are:-

1. IP (Intellectual property)/Module Level
   The IPs RTL is written in a generic way that includes a lot of parameters so that the functionality of the IP can be altered just using the parameters. So the goal of

the IP level verification engineer is to verify the IP regressively for every configuration so that RTL can be integrated into a higher (SoC/Subsystem) level. In IP level verification, an IP is verified in its own test environment called IP environment it usually SV/UVM based. The purpose is to verify the functionality of IP. In IP level verification coverage percentage of 100% is expected.

2. Subsystem Level in the Subsystem/Platform level,
   The goal is to ensure that all the critical subsystems are verified before integrated to main SoC. For e.g. A subsystem is a platform which consists of 6 Core (processor) along with other bus masters, there is a bus matrix (like NoC) and some slaves. The interaction between masters and slaves is verified at the subsystem level before this subsystem is moved to complete SoC.

3. SoC Level
   Platform and all the non-critical/peripheral IP are integrated into one single code called the SoC (System on Chip) RTL then this RTL code is verified by the SoC verification engineer. The goal of the engineer is to verify the integration of various generic pre-verified IPs in with all the I/Os of the IP

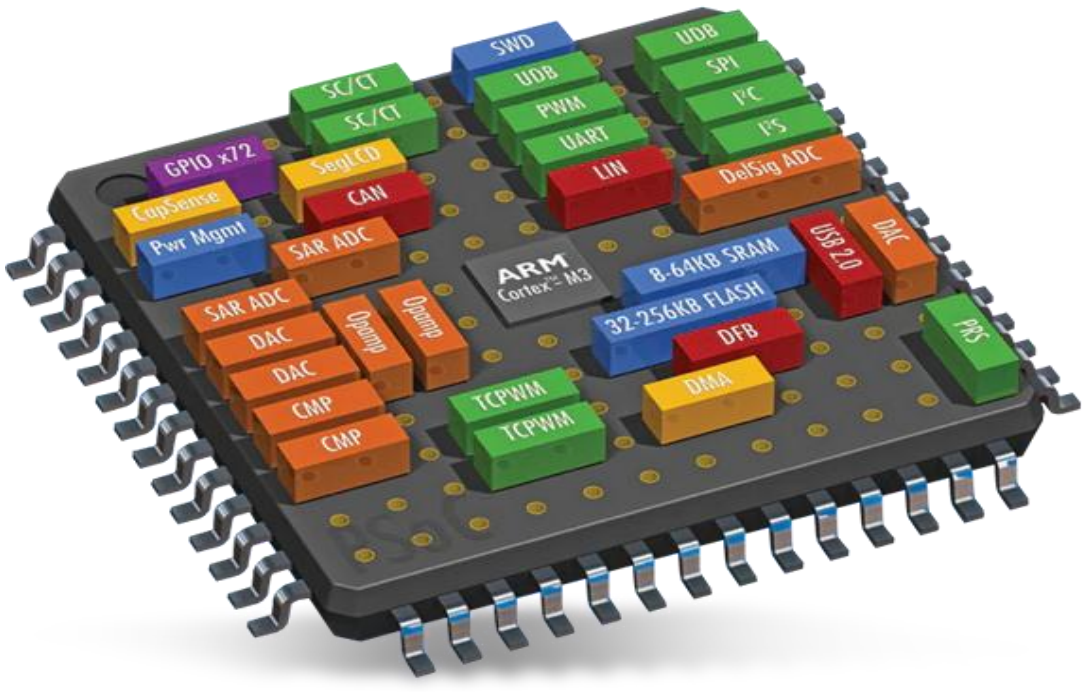


Figure 1.1 SoC Block [11]

1.2 TYPES OF VERIFICATION

Verification plays a crucial role in ensuring that a digital design functions according to its specifications before fabrication. With the increasing complexity of System-on-Chip (SoC) designs, multiple verification methodologies are employed to achieve high functional coverage and improve design quality. The three widely used verification techniques are Directed Verification, Constrained Random Verification, and Formal Verification. Each technique has its own advantages and limitations. Therefore, modern verification environments typically combine these approaches to achieve a comprehensive verification goal.

1. Directed verification

   The directed verification method, is a Specification-driven approach to verify the design under test (DUT), where each of test cases are designed by humans according to functional specification of the DUT. A rigorous analysis of the design is undertaken, and a verification plan is detailed that lists all the required functional properties for the design, then individual tests are written for each functionality.

   This methodology ensures directed execution of tests on all requested features and leaves complete control to the user. The effectiveness of direct directed verification is especially prominent during initial phase of verification, as the designer/ engineer could ensure the fundamental design functions and readily find flaws in the implementation of the same. Since tests are directed towards some individual feature functionality, there is steady and predicable coverage improvement.

   As design size scales, so does the required number of tests to ensure overall coverage with this approach. Maintenance of test case could become burdensome.

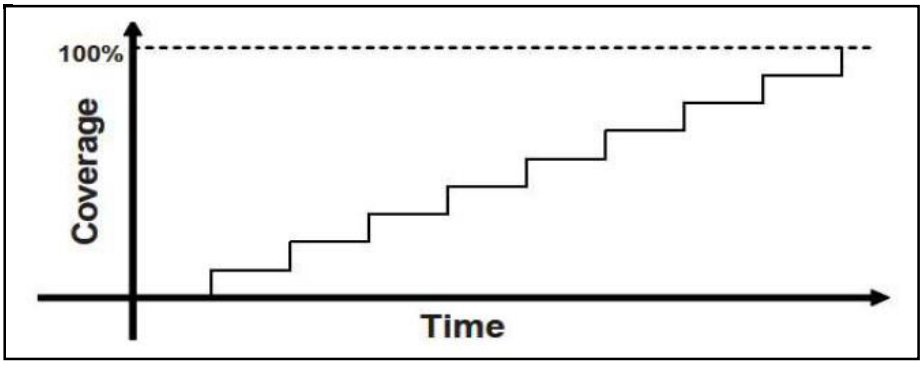


Figure 1.2: Coverage vs time in direct stimuli verification [12]

2. Constrained Random Verification

   Constrained Random Verification (CRV) CRV is the methodology used in modern day SoCs most extensively in terms of verification. Different to directed verification, CRV synthesizes stimuli to a test stimulus through randomisation and variation of variables whilst adhering to a specified value in user written constraints, i.e. They define the permissible space of possibilities from which variables will be random, generated with varying seeds per run, will choose valid test stimuli.

   System Verilog uses the rand key word to declare random variables. The use of random variables allows different runs of the simulation to produce different input stimuli based on varying random seed values whilst still ensuring that only permissible value ranges of those stimuli are generated.

   The major benefit is the ability of constrained random verification to uncover corner-case bug and unforeseen behaviour that manual test benches are typically not capable of finding. Functional and code coverage figures are also tracked during simulations, measuring overall verification progress.

   Based on the figures obtained, other test scenarios are generated using coverage-driven methods to further the simulation and to add additional coverage requirements to constraints until the required coverage criteria have been attained. Despite a large improvement in the quality of the verification, difficulties in the debug process are sometimes encountered since finding a specific random test requires the availability of its corresponding simulation seed value.

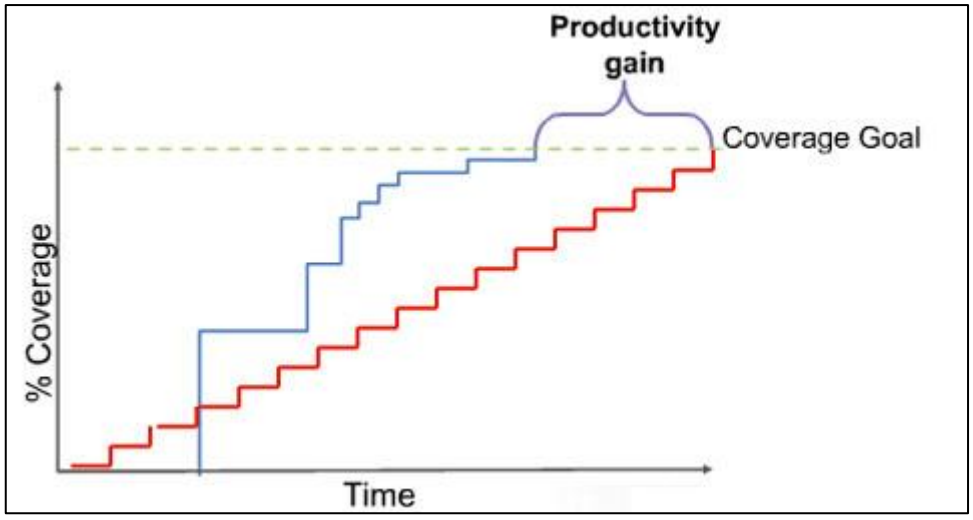


Figure 1.3: Coverage vs time in constraint random verification [13]

3. Formal Verification

   Formal verification is a mathematical method for proving that a digital design is functionally correct without having to run any simulation based on input vectors. The approach is not based on the simulation of a set of directed input vectors for example, but checks whether properties always hold by exhausting all possible input combinations of the design within the state space. The functional correctness of the design is described using properties or assertions.

   These properties, normally written in SVA (SystemVerilog Assertion), are given to a formal verification engine that either provides a mathematical proof that the property always holds for the design or returns a counterexample, a trace of inputs and outputs, that demonstrates when the property does not hold.

   Formal verification is most suitable for verification of protocol compliance, safety properties, detection of deadlocks and checking the correctness of control logic. Bugs which cannot be detected in simulation remain hidden, as it checks all possible reachable states instead of just the few that correspond to the defined simulation scenarios. Formal verification has some limitations though. With increased complexity, the size of the state space increases in an exponential fashion (well known state explosion problem) making the task of state space traversal computationally infeasible.

Writing correct and robust properties can also be tedious and the task of finding the cause of failing properties may be very hard in complex designs.

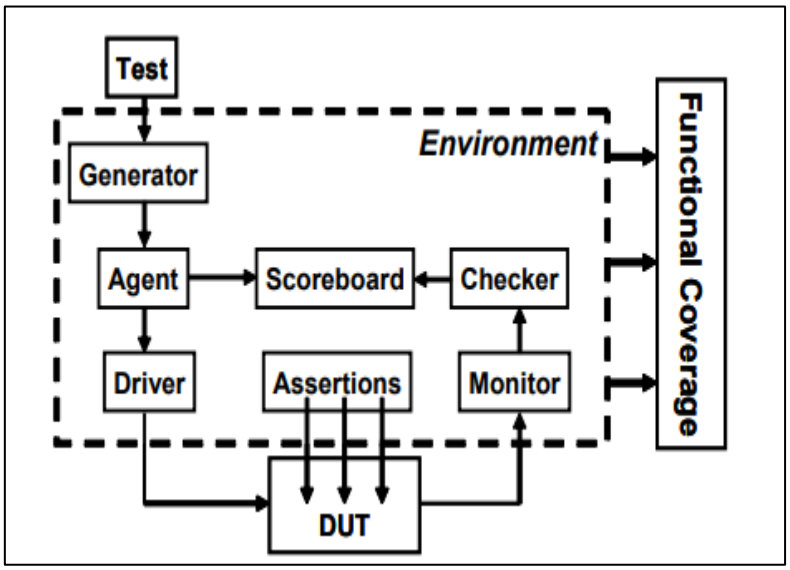


Figure 1.4: Formal Based Verification [14]

For a variety of reasons, no single methodology or technique has ever proved to be enough to achieve full verification of the more complex SoC designs. Directed verification offers determinism as we use test vectors that specify tests, while constrained random verification, used for dynamic verification, improves coverage by testing for unexpected behaviours, and formal verification applies formal mathematical reasoning to verify properties. This has led to a hybrid verification approach where multiple methodologies are integrated with silicon design to achieve a higher level of functional coverage, improve bug finding effectiveness, and ensure a higher probability that the silicon will function as intended.

1.3 ROLE OF AUTOMATION IN SOC VERIFICATION

Automation The main reason to make extensive use of automation in a verification flow is to enhance the efficiency of modern verification environments. In large scale SoC verification, executing a large number of simulations runs and verifying result in numerous log files. Manual management of simulations and regressions would cause verification to run extremely slowly and generate errors. Automation enables that most repetitive verification activities be performed through script and automatic workflows,

automate test and simulation management and facilitate result analysis. Automation streamlines and automates repeated verification tasks, reduces effort and accelerates the verification process by managing huge test suites efficiently.

Commonly used languages such as python and shell script automate simulations to be launched, manage their test execution and gather and analyse simulation logs to report summarized test results.

In this study, automation will be employed to the efficiency of certain steps of the SoC verification flow and a proposed method based on the automation concept to decrease tedious jobs and enhance workflow will be implemented to assist verification engineer.

CHAPTER 2

MICROCONTROLLER SAFETY CONCEPTS WITH ASIL

ISO 26262 (Functional Safety – road vehicles), an adoption of IEC 61508 into automotive specific requirements, is a key risk classification standard with ASIL as its core. The standard aims to give a coherent, systematic method of risk management of the malfunctioning electrical and electronic system within automobiles and meet required automotive-specific safety levels ASIL Automotive Safety Integrity Level.

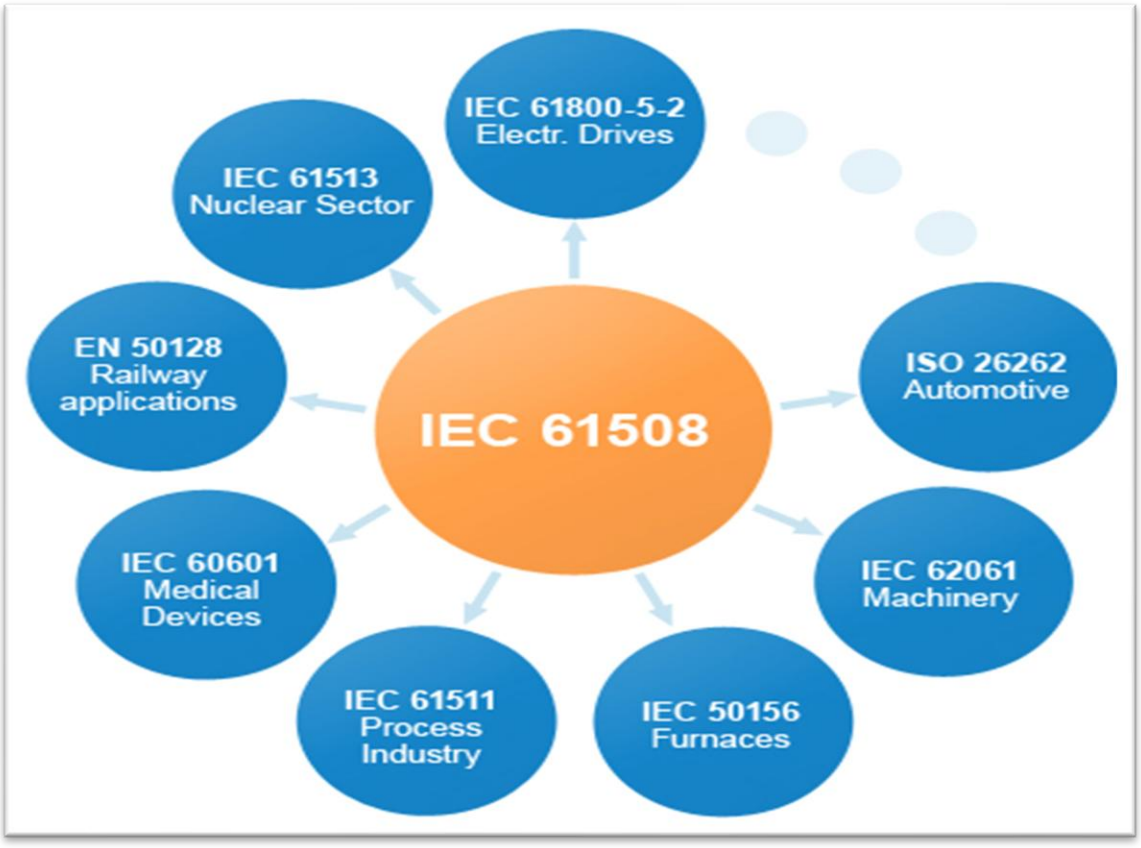


Figure 2.1: Standard included in IEC 61508 [15]

2.1. ASIL AND ISO 26262 OVERVIEW

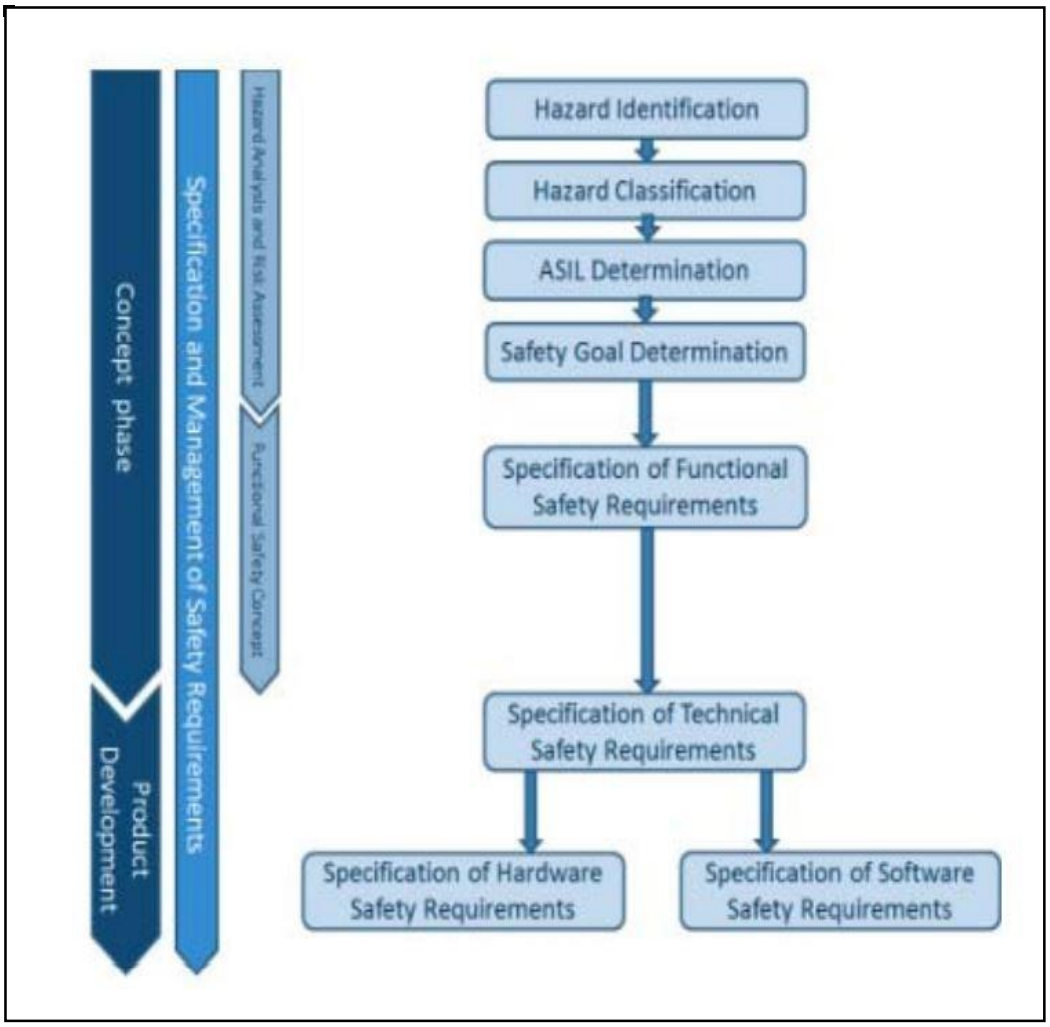


Figure 2.2: Phases of ISO26262 serial-wise [16]

ISO 26262 gives a specific definition to the concept of Functional Safety; functional safety is the achievement of the absence of unreasonable risk due to hazards caused by the malfunction behaviour of E/E systems. ASIL helps to apply the ISO 26262 definition to specific safety requirements. By providing an ASIL to each safety goal the risk of a hazard informs the necessary engineering work.

Automotive OEMs can use ASILs to improve trust with their customers, satisfy expectations for regulatory compliance, and ensure they can deploy innovative vehicle features.

Figure 2.3: Phases of the ISO 26262 automotive safety lifecycle

With ever growing complexity of the system (due to the addition of the high level safety standard ASIL which plays a major part as we have added more and more high level safety standards like assisted driving functions, automation functions and software functions), an integration higher the and a higher software portion adds threats from systematic failure and hardware random fault (a hardware will work, or it won't, then how do we protect? ASIL Based Requirements. These impose a level of design constraint, implementation constraint, V&V constraints.

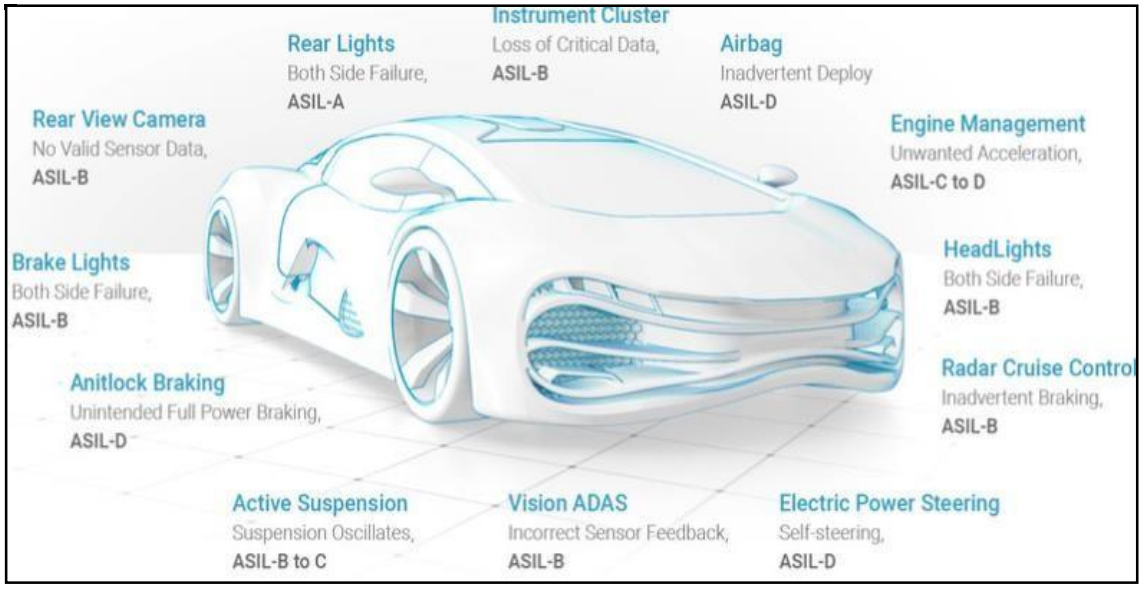


Figure 2.4: ASIL levels of components present in automotive [17]

2.2. HAZARD ANALYSIS ANS RISK ASSESSMENT (HARA)

HARA is carried out in the concept phase of ISO 26262 and thus the basis of ASIL classification. This process aims to identify possible hazard caused by system failures and assess the potential risk in different operating conditions. For every identified hazardous event there are 3 elements that are analysed: Severity, Exposure and Controllability.

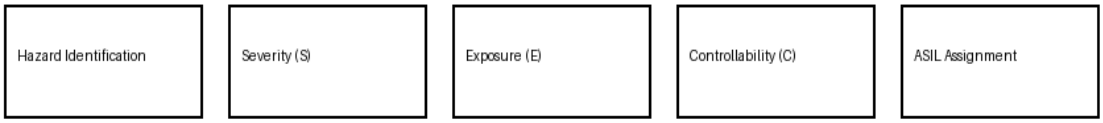


Figure 2.5: Flow from hazard identification to ASIL assignment

The Severity depicts possible injury level, Exposure deals with the probability of the operation conditions and Controllability describes how the driver / system can influence the event to prevent harm. By the aggregation of these 3 parameters, one gets ASIL level or a QM grade

2.3. ASIL CLASSIFICATION LEVELS

There are four ASIL classifications according to ISO 26262: ASIL A, ASIL B, ASIL C, and ASIL D, with ASIL A as the lowest integrity level and ASIL D as the highest integrity level (i.e. Most severe hazard rating). An ASIL level drives both the depth of the safety analysis required and the constraints on architecture.

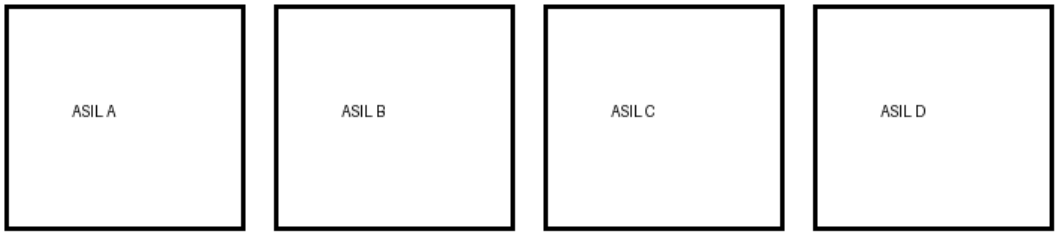


Figure 2.6: ASIL levels defined in ISO 26262

High ASIL rating typically mandates redundancy, redundancy tolerance, robust diagnostics and highly robust systems (e.g. ASIL D); ASIL A levels permit lower levels of integrity, yet they still offer reasonable risk reduction.

2.4. ASIL IMPACT ON SYSTEM, HARDWARE, AND SOFTWARE DEVELOPMENT

ASIL has implications for every layer of product development. In the system-level design stage ASIL impacts the functional decomposition as well as the decision of the safe

architecture. ASIL influences the target failure rate, the diagnostic coverage and the freedom from interference in hardware development.

ASIL also sets strict requirements in terms of coding standards, validation approaches and tools qualification for software development. For micro-controllers the ASIL-qualification involves an implementation of safety features such as lock step cores, watchdog mechanisms, memory protection units and error correction codes. The safety mechanisms help fulfil the intended ASIL.

2.5. IMPORTANCE OF ASIL IN AUTOMATIVE SAFETY

To achieve a suitable level of safety within current and next-generation automobiles ASIL is crucial for a modern automobile manufacturer in the safety aspect. ASIL determines how much safety risk exists for an auto part or auto component and measures that in a development program to a defined amount risk. As the ASIL level for an item becomes higher, then an improved quantity of safety testing should take place to test the same object and achieve functional safety within an automobile.

CHAPTER 3

LITERATURE SURVEY

The progress of semiconductor technology has pushed SoC complexity to extremely high levels nowadays. Modern SoC integrates tens of processor cores, memory blocks, communication interfaces, I/O devices and specific accelerators in a single chip. With such complex integration, the number of components and their interaction among them lead to the verification of the whole system becoming more and more difficulty.

Therefore, for the last decades, there are number of research and experts provided approaches, automation technique and platform toward the high-level complexity of SoC verification. In this chapter, we survey some important researches about SoC verification methods, verification automation, and novel verification platform.

3.1. CHALLENGES IN SOC VERIFICATION

Simulation is among the largest contributors to semiconductor-design flow. It's said that as much as 60-80 percent of the entire design cycle may be used for simulation, owing to the proliferation of complex SoC architecture and incorporation of several IP blocks, according to a recently published study [1]. Many of these components often come from various vendors or development teams working concurrently.

Correct data communication between components and validating the overall system behaviour on an integrated platform become a daunting task for verification engineers.

In some views, conventional simulation processes have been noted to be ineffective, particularly for sophisticated SoC platforms and may require more organized processes and portable verification environments. (2) For example, with many of these components operating within the various functions of a typical mobile system SoC, millions of simulation runs are run throughout verification in testing a large number of test configurations; thus, verification engineers are now focused on effectively managing these workloads.[3][4].

3.2. UNIVERSAL VERIFICATION METHODOLOGY (UVM)

UVM is a proven methodology for functional verification and is considered the de-facto standard by design houses. UVM is built using SystemVerilog and provides a modular and scalable verification environment design that can be reused across various designs.[5] UVM provides a framework which is organized around the basic building blocks that communicate with the design under test (DUT) to stimulate test cases, collect test observations, and verify the design function.

One of many major advantages of UVM is its emphasis on reusability and modular design. Verification components developed for a project can often be reused in other projects, reducing development time and improving verification productivity.

The UVM encourages the re-usability and the concept of modular design, thus saving time and efforts in the verification space.

Existing work shows success of using UVM based verification methods on several projects like communication protocols, cache memories and memory controllers in the design of the order of large scale systems on chip[6].

3.3. AUTOMATION IN SOC VERIFICATION FLOW

The efficiency of the SoC verification process is significantly improved by the use of automation techniques. Often the verification engineer carries out multiple sequential but repetitive work that includes running the simulations, performing regression jobs, parsing the logs, creating the verification reports etc., which could consume a large amount of time and human effort while also being highly prone to human error. There are various proposed approaches for an automation framework which combines together some part of the verification processes; like, automating the verification simulations, running of the regression suite, post-processing the verification results and then reporting them, which improves the overall productivity of the verification engineer [7].

One very important application of automation is to enable the regression on-chip verification. Since new features or changes to the existing system may possibly introduce

design errors into the logic, regression testing ensures against this by reusing some earlier verification test cases against changed design which is run over several design configurations to quickly identify and fix the design flaws. Running regression helps speed up the overall verification time and hence the turnaround time for finding bugs in the new design.

The process of running of very large number of simulations can be automated to speed up the process which distribute the work on multiple processors, collect the verification results automatically and bring back quickly, reducing overall verification cycle time [8]. In last few years few advanced techniques on automation which utilize machine learning approach to enhance the efficiency of the verification process have also been suggested. These techniques use information about past simulation results and identify and focus on the test cases that ensure maximum possible verification coverage [9].

3.4. EMERGING TRENDS IN SOC VERIFICATION

While a growing number of researchers are examining novel means to boost the verification efficiency while SoC complexity escalates in a faster and faster pace, this includes for example smart verification environments that take advantage of the strengths of both simulation-based and formal methods of verification. Also, significant progress is observed through integration of artificial intelligence and machine learning into the design process, so as to automate the task of finding of coverage gaps, automatic generation of tests and optimized regression testing. Last, functional safety for safety-critical applications like automotive chips necessitates intensive formal techniques for validation based on compliance to standards of functional safety.

Efforts are under way to make effective use of verification techniques for state-of-the-art SoC designs for safety-critical applications [10].

It is stated in literature that verification is one of the hardest phases of developing today's ICs (semiconductor). Many approaches to tackle with increasingly complexity like constrained random verification, coverage driven verification, assertion-based verification have been established in the last decades. Of all these approaches the UVM has become the dominant methodology of creating verification IPs (VIPs).

In addition to methodologies for developing the IPs the domain of verification has more recently attracted automation in the area of creating them and boosting up productivity.

Review of literature of the problem indicates that SoC verification has benefited tremendously from automation techniques for enhance productivity and to cope with the rapidly complexity in ICs. In the rest of this thesis an endeavour is carried out to introduce these automated techniques to increase SoC verification productivity for more effective verification for the present and the future ICs.

CHAPTER – 4

PROBLEM STATEMENT & PROPOSED METHODOLOGY

With the ever-increasing complexity of Systems-on-Chips (SoC), the amount of verification effort has also increased greatly. These devices integrate hundreds of IP blocks, multiple protocols, safety mechanisms, memories, processors and peripheral interfaces all in a single chip. Each of these blocks should not only be independently verified, but should be integrated and verified within the entire SoC as well to prove the correct functionality under various conditions.

Consequently, verification is indeed the most time consuming and expensive part in a semiconductor product's design cycle.

The design cycle is a complex task that usually involves thousands of simulations and regression tests on numerous versions of the design throughout the product lifecycle. Every time a bug is found, a change is made to a fix or an enhancement is added, a large regression suite has to be re-run on the modified design to confirm that no previously working functionality has been adversely affected. Such runs produce massive logs, coverage reports, waveform database, error logs and other numerous files. It becomes quite tedious for the engineers to manually scrutinize the large volumes of information generated by each of the simulation runs, thereby slowing down bug closure and root cause analysis.

Another significant challenge arises from the ever-increasing development of derivative devices and multiple design instances.

There exist many testcases which is functionally similar between two or more instances but verification engineers manually script or modify the existing test cases to adapt them to specific instances by modifying configuration parameters, interface mappings, instance names and design specific functionalities. This approach requires significant amount of time and introduces manual errors and inconsistency among testcases. Also, the verification results produced by running these tests are widely distributed in many log and report files and folders across different directories in the work directory.

Thus, it becomes a very difficult task for engineers to efficiently find the test case results status, isolate common failure modes, track the history of a regression, or perform failure analysis.

The sheer size of the SoCs and complexity of design is a reason to strive for a methodology that makes the process of verification more efficient. Repetitive activities like test preparation, regression execution, result extraction and failure analysis not only consumes a large proportion of design engineers' time, but this could rather be spent efficiently in the actual design activities or on analysing critical failure symptoms.

Hence, there is a dire need for automation mechanisms which automates repetitive tasks of the verification flow, manages regression results effectively, generates and organises verification outputs easily to accelerate the bug closure process.

Automating regression activities drastically reduces engineering effort, ensures consistency, and brings quick closure to bugs, ultimately lead to a robust design and in return shorten the time-to-market. This thesis proposes an automation methodology for selected stages in the SoC verification flow. Specifically, it addresses verification efficiency by means of providing a script-based automation for various repetitive verification related tasks, automating verification artifacts generation for similar design requirements, enabling central management of verification results and automating verification result analysis.

4.1. PROPOSED SOLUTIONS

The need to verify modern System-on-Chip (SoC) designs is a challenge, because of complexity involving integration of multitude of Intellectual Property (IP) blocks, multitude of design configurations, and plethora of functional scenarios to verify. Since SoCs are growing in complexity with every new version of the chip, the verification process would necessarily involve a massive amount of testing, repeated simulations and continuous verification results analysis. A good portion of the process involves repetitive and tedious tasks like test creation for every design configuration, running vast regression suites and analysing outputs. The amount of verification effort involved when such tasks were performed manually, along with potential human errors, has become an obvious reason for seeking automation.

In a large verification environment, often there exist a variety of design configurations, or instances, which require very similar set of tests, since a number of them do share the similar functionality though they must still be verified individually for successful operation in a complete system. Generally, engineers take existing test cases developed for previous versions of design configurations, and customize them for new instances of the same design. Such customizations usually involve numerous manual parameter changes which can involve, for

instance, tweaking configuration registers or interface signal bindings between instances or other designs, all done with relation to specific instances or their configurations, for example, tweaking module or instance names or interface mappings. As the number of design configurations or instances becomes large, manual adjustments become difficult and unmanageable.

The approach described herein intends to enhance productivity by utilizing automation to various stages of verification at the SoC level without aiming to completely automate all aspects of the verification workflow. Instead, it targets some parts where automation could reduce monotonous work and preserve flexibility. The focus is on adapting pre-verified reference test cases for different instances of design or configuration and analysing the outputs from regressions run on test configurations.

One useful aspect to introduce is the generation of test cases for new design instances using reference test cases for prior instances. Most verification environments do contain certain pre-verified designs where the associated test cases, containing their tests, are already validated. Such reference tests usually have valuable structures, stimulus generation elements and configurations.

This can be used to adapt new test cases.

The process of adaptation would take on the help of automation scripts to parse and create test cases based on reference cases. Herein we will be looking at adapting various reference test case features like instances, interfaces or configuration parameters to fit a specific target design instance without modifying them manually, by writing simple scripts.

Automating the analysis of regression test results the other phase of the verification workflow which benefits from automation is the analysis of regression test results. Regression testing is one most popular technique used in the verification environments to ascertain that any changes in the design would not lead to unwanted functional problems. Since regression test suites are in fact a collection of hundreds or thousands of test cases, many of them can give the simulation results and log files as output and all of them need to be scrutinized and analysed.

Analysing a large quantity of simulation logs by hand is one time-consuming and ineffective process. With large regression suites this analysis of results can indeed take several hours or days. With proper automation scripts we can parse regression result log files and extract useful

information from a very large pile of data. They can identify which test cases are Passed or Failed and any errors encountered or warning messages thrown, during the course of regression run. In addition to determining which test cases Failed, one can further parse regression log to find which of the IPs, functional blocks, instances or design units in the design hierarchy are related to each of these failures.

A typical output of our regression analysis would be a summarized report highlighting the failed functional areas of the design, related IPs and instance numbers.

The process automates the search for patterns in large files to identify test statuses, error messages, warnings, assertion violations, functional failures, block level failure. It automatically reports failure at the design unit level, identifies specific IPs or instance IDs that are failed as a part of a specific failure. Automation scripts to aid in understanding regression output This work uses simple scripting languages like Python and Perl to automate some aspects of the verification workflow.

In the semiconductor verification world, this form of scripting-based automation to automate tool flow is very commonly encountered, for the sake of its flexibility and the power of string manipulation offered. Python provides an extremely clean and easy-to-learn language with rich libraries that facilitate automation. Perl provides exceptional string parsing abilities that make it particularly adept at this task of wading through large quantities of log files to pinpoint desired pieces of information.

As the automation relies mainly on analysing simulation logs or regression test output, the primary languages we will utilize for automation script writing are Python and Perl.

In the remainder of the discussion, we provide a brief outline of how we can utilize these scripting languages to automate tasks which would otherwise involve a significant amount of manual effort. Not all parts of verification are automated We acknowledge that, not all aspects of an SoC's verification need to be automated and not all activities within the verification workflow need to be automated, and are willing to achieve this by selectively automating the areas where we can expect to gain the most. The goal here is to augment the process of verification by using scripts to automate tasks such as creating appropriate test cases adapted to different design revisions or new instances of blocks and generating useful reports summarizing large amounts of raw output data, not to fully automate the SoC verification flow. Our framework helps the engineer by automating manual, repetitive processes where

consistency is crucial to enable verification engineers to concentrate on analysing the behaviour of the IP under design and resolving the issues raised by the simulation test suite.

Further research the automation approaches that will be described herein will, as this research continues, be developed further and potentially integrated with the formal verification environment for experimentation and to verify the efficiency gains achieved in terms of human productivity.

CHAPTER – 5

CURRENT WORK AND IMPLEMENTATION

The complexity in modern day System-on-Chip (SoC) designs made functional verification one of the hardest and longest stage in semiconductor design flow. In today's world of Verification environment, a verification engineer would be spending the majority of his/her time writing a huge number of testcases, executing long regression runs, digging into log files of simulation, and changing the verification environment as the design evolves. While modern verification techniques such as Universal Verification Methodology (UVM), Constraint random verification and Coverage driven verification have significantly improved the quality of verification but one aspect of the verification process remains the repeated manual effort required for verification process.

The process where engineers need to modify the existing test case for the similar instances of the design, organizing the regression runs, searching through hundred thousand of log files, tracking the history of test runs, transferring a verification environment from one platform to another.

Doing these manually can make verification turnaround time lengthy and also brings chances of human errors and variations across the different verification environments. In order to overcome this issue, this thesis presents and implements several automation approaches that make the overall SoC verification process more efficient. Rather than replacing the current verification methodology, we have put our efforts into introducing automation at few places of the verification flow where the manual engineering effort is highest and we automate the process. Our goal is to reduce the manual efforts with the help of automation and at same time maintain the quality of verification with the existing verification flows.

Automation implemented throughout the chapter has several automation modules for better productivity in verification.

It is responsible for automatically generating test cases for similar verification purpose, for analysing test runs results, centrally managing all the regression information and also for

automatic transfer of RTL code into the tester testbench. Automation implemented in this thesis is primarily done using python and Perl scripts. Automation tools were implemented into an industrial SoC design verification flow, working with Cadence Xcelium simulator.

In this chapter, we present the implementation methodology that we followed, explain in detail the automation framework, verification environment, and individual automation modules implemented and finally present and discuss some experimental results from the automation done.

5.1. OVERALL AUTOMATION FRAMEWORK

In an SoC verification flow, numerous steps are executed with several repetitive tasks that need high levels of manual intervention. Manual involvement often requires the verification engineers to perform similar tasks at multiple intervals in a particular testcase and during regression activity, thus leading to a reduction in productivity. To overcome this, we propose an automation framework aimed at maximizing productivity in SoC verification by automating repetitive activities that currently demands more effort from the engineers.

Instead of altering the existing verification flow, this automation framework complements the flow with the insertion of scripting-based automation at various steps in the verification process.

We aim to automate tedious and time-consuming manual activities so that the design and verification engineers can utilize more time and effort for enhancing verification quality. The overall workflow of the automation framework can be represented as follows. Initially, verification inputs such as design specifications and previously verified reference testcases are considered as the primary inputs. Using this information, automation scripts either generate or adapt new verification testcases based on the current design requirements that fall under the same design specification criteria. These new testcases then executed in the existing System Verilog based verification environment.

Simulation result in the form of log files is collected by the regression automation framework after each testcase is simulated.

This automation module parses the collected log files and determines the PASS / FAIL conditions, identifies the warning / error messages and based on them, classify the simulation failures into categories and provides a summarized view of regression results, eliminating the need for manual scrutiny of hundreds of log files by the verification engineers. To track testcase executed so, regression information is stored in an centralized regression results database. This database keeps track of all information related to each testcase such as number of times tested, time taken for each testcase, results, type of failures, time taken to analyze etc and this will help in tracking the status of a particular verification suite, comparing results of regressions conducted at different times, and identifying failure trends and patterns etc.

In addition to regression automation, we also propose an automation script to automate the RTL Testbench to Tester Testbench conversion. The tester testbench is derived by analysing the RTL testbench and by adapting parts of the RTL testbench for tester-based verification. The generated tester testbench helps in maintaining a consistency between silicon and pre silicon verification.

Automation framework developed has the following modules Functional Modules:

- Test case Generation Automation
- Regression Result Analysis
- Regression Result Database for missing test cases
- RTL testbench to tester testbench conversion

All these automation modules run as standalone scripts and shared information for inputs & outputs, thereby building an integrated automation framework for effective SoC verification. A conscious effort was made to make the automation framework modular so that, individual automation scripts can be executed standalone or can be used in an integrated manner with existing flows of industrial usage, making the script to reuse and maintainable.

5.2. AUTOMATION OF TESTCASE GENERATION

The manual effort of creating verification testcases for present day SoC Designs. In a complex Verification Project, lots of designs with identical functionality but with a differing parameters, interfaces mappings, register configuration, instance names need to be verified. Even with these subtle differences the Verification Engineers re-create or manually port or amend the existing test cases for the new instance.

This causes wastage of development time and also possibility of manual errors.

To make this process efficient an automated script is generated. The automation re-uses previously Verified Reference test cases and Generates the New Test cases which fulfill the similar verification requirements without need of recreating testcase again. The automation script identifies the fields that are configured and alters the instance-specific data accordingly rather than creating the test cases manually. The test cases are being generated by 1) Reading the existing testcase 2) Identifying the fields which needs to be modified 3) Update the instance-specific data and 4) Writing new test case without any change in verification goal of the reference test case.

In essence, the automation enables reusing same logic across several Design instances without much human involvement.

The automated test case creation reduces test case development effort and inconsistency and it would make future maintenance easier as modifications can be passed on to newer instances easily. Major features are

- Automatic utilization of verified test cases
- Test cases generation based on parameters
- Automatic update of instance names, interface mappings

- Reduced manual coding effort

- Test cases re-usability and consistency

- Fast adoption for derivative designs and new configuration.

```
// DEPARMENT NAME  :   MDRF
// AUTHOR NAME     :   Pallavi
// ---------------------------------------------------------------
//


#include <environment.h>
#include <cem_functions.h>

#include <context_cmu.h>

#define STIM    cmu_B23_fll_evt.c


 int main() {
    CMU_CONTEXT* cmu_context[0];
    create_context_cmu23(&cmu_context[0]);

    CEM_CONTEXT  cem_context;
    create_context_cem_8(&cem_context);
    cmu_fll_evt(cmu_context[0]);
    fccu_cmu_fll_evt_clr(cmu_context[0],cem_context);
 return 0;
 }
 //automated generated test-case
```

Fig 5.1: Sample Test Generated Using Automation Script

5.3. AUTOMATION IN REGRESSION RESULT ANALYSIS

During the validation of a system on chip (SoC), regression tests are run regularly and consistently to make sure that changes in the design do not have any negative side effect on the functionality that has been tested earlier.

A test run of a regression may contain many testcases, hundreds and there, where each testcase has a log, messages, an error list, and a status at the end. Manually processing all those results and reports is neither easy, neither productive. That's why we tried to automatize the analysis process.

That process can be split in two main parts. First, after the regression run has been completed, we want to get rid of simulation logs as soon as possible and get relevant information about the passed and failed testcases, as well as warnings, errors and messages, and second, we want to be able to store all the relevant information in a database, to trace the history and get even more information about the results of a particular test.

Regression result analysis automated. The result files from a regression simulation run are available as log files for each testcase and test suite. These files will have messages, warnings, errors, fatalities, PASS/FAIL status at the end of it. We automated the process to parse all the relevant information out of the simulation files immediately after a regression run has completed.

These Python and Perl script can be executed just after the run and then they gather information like Pass/Fail status of every testcase, the amount of warnings, errors and fatalities, etc. No more individual log parsing! A comprehensive regression summary will contain the needed information for the verification engineers and will be displayed, as soon as the run finished! The amount of important debug messages in failures are extracted too; this helps to find and debug failed testcases more rapidly.

A central database for regression results. For storing of the results, we made use of a database. In our case this is a relational database which allows to store many tests runs in it, along with the information as captured previously. For instance, in such a database we could have columns for the name of the testcase, the run time of the testcase, the overall testcase status, how many errors or warnings occurred, the description of a failure, etc. This allows the verification engineers to compare the results from different regression run, to see the evolution of the quality of tests, and to see the root of failures.

Using that automation the amount of work for test analysis can be reduced by up to a factor 10. Also, the debug process could be speeded up and more control over the validation process can be gained.

C9 | 4

	A	B	C	D	E
1	**stingray_vectors**	**First Failure Description**	**Count**	**VerifResp**	**Total Fail**
2	can_intfls	TESTCASE PASS missing	4	Verif Owner Name	26
3	cem_intfls	RE32: A=71739100 R=0000080e E=00000000	19	Verif Owner Name	62
4		REM32: A=71739108 R=00001000 E=00000000	4	Verif Owner Name	
5	adc_vector	***[Makefile: 340:	8	Verif Owner Name	51
6		Why hasn't the C code terminated the	5	Verif Owner Name	
7	debug_swd_intfls	TESTCASE PASS missing	5	Verif Owner Name	56
8		core status CS_CL0_C0 is not right HALT_YES	4	Verif Owner Name	
9	new_ch_mux	REM32: A=71739108 R=00001000 E=00000000	4	Verif Owner Name	121
10		REM32: A=7b000b40 R=00000000 E=ffffffff	4	Verif Owner Name	
11	dsph	capi_int.c: Interrupt/Exception occurred without	6	Verif Owner Name	17
12		check_tea_counter: tea_counter does not match	4	Verif Owner Name	
13	dspl_intfls	Simulation not completed	6	Verif Owner Name	24
14		RE32: A=64000000 R=00000000 E=0000a5a5	4	Verif Owner Name	
15	fccu_intfls	REM32: A=717391bc R=000000d0 E=000000f0	7	Verif Owner Name	31
16		RE32: A=71739100 R=0000080e E=00000000	4	Verif Owner Name	
17	flexray_intfls	RE16: A=70e18802 R=0000 E=4000	9	Verif Owner Name	14
18	gtm_intfls	RE32: A=7400210c R=00000000 E=007ff7ff	4	Verif Owner Name	29
19	ips_read_mux_intfls	check_tea_counter: tea_counter does not match	6	Verif Owner Name	28
20	ipu	Run cancelled because dependency job RTL_sessions.malhotrv.26_03_02_20_58_52_4065 _RTL_sessions___RTL_tests___group_rtl_TESTCAS	34	Verif Owner Name	34
21	linflex_intfls	LINFLEX-UART VIP : Framing Error detected	9	Verif Owner Name	77
22	mc_me	Assertion testbench.cluster0_reset_assertions_inst.cl_rst_	4	Verif Owner Name	36
23	mc_rgm_intfls	REM32: A=263ec010 R=00000000 E=00000006	9	Verif Owner Name	46
24	mc_soc	RE32: A=71739100 R=0000080e E=00000000	18	Verif Owner Name	20
25	mc_system	RE32: A=71739100 R=0000080e E=00000000	4	Verif Owner Name	17
26					

Fig 5.2: Regression Result Analysis

5.4. RTL TESTBENCH TO TESTER TESTBENCH AUTOMATION

However, RTL testbenches created for pre-silicon RTL verification cannot be used directly in the tester-based validation phase as production tester systems have different execution guidelines. Verification engineers typically have to transform RTL testbenches into tester testbenches through significant manual redesign work. In an effort to reduce this manual conversion effort, we have created an automation script that generates tester-compatible testbenches from the RTL testbenches.

The script parses the RTL testbenches and reuses all or most of the verification components, such as signal declarations, stimuli sequence, clock and reset activity, configuration parameters, and interface descriptions.

Any simulation-related constructions are adapted to meet the tester environment requirements, or are removed where appropriate, ensuring that the testbench functionality is the same as before. The resulting testbenches can be used with the tester, thus running all the functional tests performed earlier on the RTL, which helps achieve better correlation between RTL and tester environments, and avoid re-doing repetitive work. The automation can do the following: Extracting reusable RTL verification components. Generating tester-compliant testbenches from RTL testbenches.

Eliminating manual redesign effort. Maintaining consistency between RTL and tester environments. Allowing faster integration with new versions of RTL. Increasing verification reusability. Using the automation dramatically reduce engineering efforts and enhance overall verification productivity by saving on time and manual work required for such tasks.

5.5. CONCLUSION

Today's modern SoC designs have led to complex verification environments, where functional verification has become the most resource-intensive stage of the SoC design flow. Large verification environments have extensive verification tasks that include test case generation for similar design instances, extensive simulation regression runs, post simulation analysis, test bench translation from RTL test bench to tester test bench etc. All these tasks add significant effort and time to the verification cycle.

This work presents design and implementation of automation approach to enhance the productivity of SoC verification flow.

Instead of replacing the existing verification flows the automation flow complements the existing verification methodologies. This approach tries to automate certain boring and time-consuming verification activities thus improving overall efficiency and productivity. Automation in the form of scripting framework was proposed for multiple parts of the verification flow. Test Case Generation Automation can utilize validated reference test cases to generate similar test cases reducing repetitive coding effort and enhancing consistency for similar verification specifications.

Regression Result Automation can efficiently analyse the results of large regression suites by identifying test case status and logging all the warning and error messages along with generating organized verification reports.

Centralized regression database provides history of all previous simulation results thus helping to track verification progress effectively. Furthermore, an approach for automating RTL Testbench to Tester Testbench conversion from pre-silicon RTL verification environment to post-silicon tester test bench was also presented. All the automation scripts were successfully verified using Cadence Xcelium and they provide significant improvements in terms of reduction in repetitive engineering effort, increase in reusability of verification assets, fast analysis of large regression and thus reduction in verification turnaround time.

This allows verification engineers to concentrate on high level debugging rather than boring mundane work. Thus, automation approach proposed in this work will have a profound impact on SoC verification methodology and helps in achieving faster, robust and high quality SoC verification in the long run.

REFERENCES

[1] M. Abramovici, M. Breuer and A. Friedman, *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[2] W. Yang, "Current Status and Challenges of SoC Verification for Embedded Systems," *IEEE Design & Test of Computers*, vol. 20, no. 5, pp. 12-19, 2003.

[3] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Springer, 2006.

[4] A. B. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide*. Springer, 2018.

[5] Accellera Systems Initiative, *Universal Verification Methodology (UVM) 1.2 User Guide*, 2015.

[6] M. Tiikkainen, "Automated Functional Coverage-Driven Verification with UVM," University of Oulu, Finland, 2017.

[7] S. Yim and J. Jang, "Accelerating SoC Verification Using Process Automation and Integration," *DVCon Proceedings*, 2021.

[8] R. Aouadja, "SoC Regression Strategy Development," University of Oulu Research Repository, 2023.

[9] S. Ravikumar, "Optimizing SoC Verification: An Innovative Framework for Enhanced Coverage and Efficiency," *European Journal of Computer Science and Information Technology*, 2025.

[10] Z. Xie et al., "An Intelligent Verification Platform for SoC Design," *Journal of Systems Science and Complexity*, Springer, 2013.

[11] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 6th ed., San Francisco, CA, USA: Morgan Kaufmann, 2019

[12] J. Bergeron, *Writing Testbenches Using SystemVerilog*, 2nd ed. New York, NY, USA: Springer, 2006.

[13] Accellera Systems Initiative, *Universal Verification Methodology (UVM) 1.2 User Guide*, 2015.

[14] A. B. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*, Springer, 2018.

[15] International Organization for Standardization (ISO), *ISO 26262: Road Vehicles – Functional Safety*, Parts 1–3 (Concept Phase, System-Level Safety Requirements), Geneva, Switzerland, 2018.

[16] International Electrotechnical Commission (IEC), *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, Parts 1–7, Geneva, Switzerland, 2010.

[17] International Organization for Standardization (ISO), *ISO 26262: Road Vehicles – Functional Safety*, Parts 1–3 (Hazard Analysis, Risk Assessment, and ASIL Determination), Geneva, Switzerland, 2018.

8% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text
- Small Matches (less than 8 words)

Match Groups

40 Not Cited or Quoted 8%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

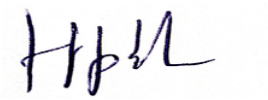
6% Internet sources
2% Publications
6% Submitted works (Student Papers)

Integrity Flags

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Dr. Hardeep Singh



Associate Professor
Department of Electronics and Communication Engineering,
Thapar Institute of Engineering & Technology, Patiala
Date: 30– 06– 2026

Dr. Hardeep Singh

submit_merged

Document Details

Submission ID
trn:oid:::

Submission Date
Jun 30, 2026, 3:17 PM GMT+5:30

Download Date
Jun 30, 2026, 3:23 PM GMT+5:30

File Name
submit_merged.pdf

File Size
779.2 KB

38 Pages

8,596 Words

49,708 Characters

0% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

0 AI-generated only 0%
Likely AI-generated text from a large-language model.

0 AI-generated text that was AI-paraphrased 0%
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.