UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN ICT FOR INTERNET AND MULTIMEDIA

Final Dissertation

"AUTOMOTIVE MICROCONTROLLER MODEL-BASED PRODUCT ARCHITECTURE VERIFICATION USING MATLAB SIMULINK TEST HARNESS"

Supervisor

Prof. Leonardo Badia

Master Candidate

Anjali Santhosh

Academic Year: 2021-2022 Graduation Date: 03/10/2022

CONTENTS

LIST OF FIGURES

ABSTRACT

The thesis concentrates on the early-stage verification of model-based product architecture for Automotive Microcontrollers. Pre-Silicon verification of product architecture eliminates the chances of callbacks of the products due to crucial bugs after production hence minimalizing the losses for the organization. For the development of the test suite, a theoretical analysis is conducted to discover the best approach for developing the test suite. From the mathematical analysis of systems, some formulas are derived.

Using these formulas, the best method and effort required to design and carry out a verification process is finalized. MATLAB's Simulink state flow diagram is used to develop the product architecture. MATLAB's Simulink test harness feature is used to develop the test suite, and the continuous integration is achieved using the MATLAB unit test framework, bitbucket, and Jama. The Application and the input files are stored in a Bitbucket repository, and the execution is performed using Jenkins.

CHAPTER 1: INTRODUCTION

The project concentrates on the model-based product architecture verification of an automotive microcontroller. Model-Based System Engineering (MBSE) [1]is a system engineering approach that applies the modelling, tools, and methods to improve a system's engineering programs and projects. MBSE focuses on expressing and recording requirements, design, analysis, and verification [2].

There is a set of scientific journals published on Model-Based System Verification explains the verification based on component-based architecture and reactive behavior of the system [3]. Structured review of the system based on model-based system engineering to improve the quality in a cost-effective way is also proposed by researches in the past years [4].

The formal expression of needs and constraints that affect and determine a system's structure and behavior is what modeling of system properties is all about. In order to support their evaluation during system design and their monitoring during system operation, system property models enable the verification of system properties through real or simulated experiments [5]. Simulation-based system verification is a major technology to verify a system in MBSE.

However, no scientific research is known or published on evaluating the best possible approach to verify a system. Thus, the first part of the thesis concentrates on finding the best way to approach a system under test. For this purpose, different known methods of verification are studied, and the best approach is finalized after performing some theoretical evaluations.

The theoretical analysis is performed on a function of the Non-Volatile Memory of the system. Three different approaches of verification, the input-based, the output-based, and the input-output-based verification methodologies are studied for the selected systems. Calculations are performed to find the maximum possible test cases that can be designed for a particular system under test when there is certain number of inputs and outputs are available.

Mathematical equations are derived from the theoretical analysis performed, and with the help of these equations, when the number of inputs and outputs of a system are known, the engineer can decide to choose on whether to go on with the input-based approach or the output-based approach or the combination of both to obtain the maximum coverage with minimal effort and time spent.

Thus, the best method to analyze a system can find out with the use of equations derived, and the authenticity of the equations is verified by studying three different functions using all the three methods for the Decision, Condition, and Modified Condition / Decision Coverage.

For verifying the authenticity of the equations, the functions are converted into MATLAB state flow diagrams, and the Simulink Test harness feature is used. Test cases are generated based on the best approach and a hundred percent of the required coverage is obtained with this method [6].

Once finding the best method to verify a system, the product architecture verification of the automotive microcontroller is carried out by following the best approach to generate the maximum required coverage with minimum effort. The product architecture of the microcontroller is studied in detail. The state- flow diagram of the product architecture is used for the study and MATLAB – SIMULINK test harness features are used to develop the test cases for verifying the system.

As the organization uses Enterprise Architect software to develop the product architecture and the extraction of MATLAB state flow diagrams from Enterprise Architect to MATLAB is possible, SIMULINK is chosen for developing the test suite. Thus, the product architecture verification is performed successfully using the MATLAB SIMULINK test harness.

As there are several regressions to be performed over a single product architecture it is necessary to automate the verification of product architecture once very new release occurs. For this purpose, the Jenkins and Bitbucket platforms are used. For the continuous integration, the unit test frame work of MATLAB is used and a script is created to execute all the MATLAB files and the test suite.

The MATLAB software is installed in a remote node, all the MATLAB files along with the script are committed to Bitbucket. The Jenkins is configured such that for every new commit in Bitbucket, the Jenkins will execute the test suite for the verification of the product architecture at a remote node and provide the report after the execution. Thus, a task that would take hours of execution for an engineer to design and verify a system under test shall be completed with-in an hour.

The Model-Based System Engineering verification approaches, the theoretical analysis to define the best approach, the study of the product architecture and the verification and continuous integration of the product architecture will be discussed in detail in the upcoming chapters.

CHAPTER 2: STATE OF THE ART

Before proceeding with any research activity, it is always necessary to have a good understanding on what resources are already available and how we can make use of these resources in our perspective. For this purpose, a certain research activity has been conducted in which different scientific journals has been studied. The initial research activity conducted on **Model Based Software Engineering** (MBSE) is reported in this chapter.

Model-Based Engineering, Model Driven Engineering or Model Driven Development is a system and software development paradigm that "*emphasizes a precise and complete* **System Architecture Model** *"blueprint", organized as an Architecture Framework with multiple Views/Viewpoints, as the primary work artifact throughout the System Development Life Cycle (SDLC)"* [7] [8].

The vehicle design complexities are increasing day by day due to the power train and component technologies for example, connectivity and automation. The increase of new technologies has increased the reliance on Model Based system Engineering to reduce cost and time to market [9].

Another research explains on how to combine the benefits of model-based testing with the software architecture in a unique way [10]. The paper explains on generating test cases from formal behavioral model of a system. It concentrates only on the software architecture verification and describes the entire approach based on a client server relationship.

Although some basic idea on Model-Based software verification can be acquired from the paper, it does not provide any specific information on product architecture verification or useful in our context. Thus, the paper was studied to achieve an initial idea on *What is Model Based Verification?*

There are researches that states that there should be a new software testing technique at the system level. This architecture-based software testing substantially reduces the costs of any problems and improve the quality of the software.

The paper claims that, "***although there exists some formal testing way like unit and module testing criteria which have been well studied, software architecture -based testing is typically done informally, using manual and ad-hoc techniques***" [11]. This makes it difficult for the engineers to measure the quality of testing and to have a lack of formal testing method at the software Architecture level. Thus, the authors concentrate on different methods of

software architecture verification. Although these methods do not fit to our context the idea of discovering different methods of developing test cases is acquired from this study.

The further research focuses on different methodologies of verifying a system under test. There are a number of methods of developing and grouping test cases known [12]. Among which grouping of test cases or developing of test cases based on the following aspects are considered.

- **BEHAVIOR OF THE SYSTEM**

  In this case, the test suite is developed focusing on the behavior of the system to each operation performed on the system. The behavior of a system can be categorized into three, expected behavior, allowed behavior and unacceptable behavior. The test cases can be grouped based on this categorization and format the final test suite with this.

- **INPUTS OF THE SYSTEM**

  As the name suggests, the test cases are developed considering the inputs of the system. The test cases focus on the type of inputs, number of inputs, number of input equivalent classes, and their effect on the system. As the number of inputs increases, the number of test cases also increases. However, if a certain behavior of the system depends only on a single input and the output will be same for all the changes applied to other inputs the redundancy should be eliminated. Based on the type of inputs, concentrate on the combination of inputs of same type reflecting same behavior, find equivalency levels and avoid redundancy.

- **OUTPUTS OF THE SYSTEM**

  Similar to the inputs, the test cases developed for verification are focused on all the possible outputs received on operating the system. Group the test cases that deliver the same output and check whether the system fails to generate a particular output. Redundant test cases should be eliminated in this case also. To obtain the full coverage of the system, in some cases there arises a need to increase the number of test cases or include redundant test cases.

- **REQUIREMENTS**

  Test cases can be grouped based on requirements of the developed system. It is always necessary to verify whether all the requirements are satisfied or not. To verify each

requirement there can be a number of test cases. Similar test scenarios can be identified and the redundancy can be eliminated. Also combining the test scenarios that can satisfy one or more requirements simultaneously helps to reduce the number of test cases.

- **PRIORITIES OF THE RELEASE**

  For every long-term project there will always be a number of releases of the product based on client requirements or bug fixes. In this case, each release comes with certain priorities to be verified. The test cases can be designed based on the priorities of each release. The priority levels can be standardized as High, Medium and Low priority test cases and the verification engineer can decide on which all priority levels to be achieved in each release. The low priority test cases are most often avoided in case of regression testing as they do not affect the functionalities. UI design related features are an example for low priority test cases.

- **FUNCTIONALITY**

  The most commonly used method of developing a test suite is generating the test cases based on functionalities. A system under test will have a certain set of functionalities which needed to be verified. To verify each functionality, there arises the need of developing a set of test cases. As we can think of thousands of test cases for each functionality depending on the complexity of the system under test, we cannot predict a particular trend or method to follow in this case.

- **COMPONENTS**

  We mainly focus on the testing of electronic control units that functions on a million lines of code. The complex automotive system will have a large number of component blocks to be tested. It is always necessary to verify whether each component is working as expected. This is the main goal of the verification engineer too. Therefore, the test suite can be developed by focusing on generating test cases based on components.

Thus, the different ways to approach a system under test is studied in detail. From this information three main contexts were chosen for detail study which suits best in our case. The detailed study is explained in Chapter 3.

The product architecture of an automotive microcontroller not only contains the software, it also contains the hardware analysis [13]. Therefore, a detailed study on architecture analysis of electronic control unit is required which comprises both hardware and software parts. The projects that we concentrate on are long term and the continuous development occurs. Therefore, a detailed study of continuous evaluation is necessary in organizational aspect.

A scientific paper in architectural evaluation [14] focuses on how architectural evaluation can provide useful feedback during development of continuously evolving systems. The study of the journal helps in understanding how architectural evaluation in a continuous setting can provide timely feedback on whether a specific capability is supported in the current architecture, what information should be provided for the architectural evaluation to be performed, and how in principle can existing frameworks for architectural evaluation can be used for continuous evaluation. The main take away points from the study are:

- Decide when to evaluate what. If evaluation results are not necessary at that point, the evaluation could be considered a waste. Evaluate things based on demand.
- Identify whether to have large and continuous evaluation or based-on-demand evaluation to answer particular stakeholder questions.

There are many scientific journals focusing on the benefits of product architecture evaluation and methods of verifying a product architecture such as the ATAM and extended ATAM technologies [15] [16] where ATAM stands for the Architectural Tradeoff analysis method. In ATAM the tradeoffs of each process are calculated or analyzed and the method of verification proceeds with concentrating on the tradeoff values.

As we are focusing on the verification of the product architecture using MATLAB Simulink test harness a research is done on the possibilities of using MATLAB for product architecture verification. Even though no specific solutions or methodologies relevant to the context are available the research helped in understanding the possibilities of using MATLAB features [17] [18] [19].

The possibilities of automating the test suite using the continuous integration features of MATLAB is also a part of the research. Continuous integration, the regression testing and the different stages of verification and the software or product development cycle is also studied for better understanding of the system [20] [21].

CHAPTER 3: ASPICE AND PRODUCT ARCHITECTURE

Automotive SPICE or ASPICE is a process maturity framework established in automobile industry to assess the capability and maturity of Organizational processes to develop the software or embedded systems. For explaining ASPICE, it is necessary to tackle the SPICE or ISO/IEC 15504 standard.

The Software Process Improvement and Capability Determination (SPICE) is a framework developed in 1993 for software process assessment. The purpose of SPICE is to evaluate development factors that allow assessors to determine an organization's capability for effectively and reliably delivering software products.

ASPICE applies the SPICE framework to automotive industry. With its own critical requirements ASPICE differs from functional safety standard (namely ISO 26262) in which it covers how design is conducted when safety is not a concern. To ensure the safety practices the automotive suppliers incorporate both ASPICE and ISO 26262 guidelines.

3.1.WHY ASPICE

ASPICE authorize a uniform evaluation process which enables suppliers and OEMs (Original Equipment Manufacturers) to be technically capable to develop, improve and adapt software or system processes such as ECUs (Electronic Control Units). ASPICE ensures that all participants along the development process of electronic automobile components work according to the state of art best practices thus minimizing the product risks.

From the supplier point of view, OEMs assess their suppliers based on the ASPICE assessment rating. It provides a more controlled system development process to ensure product quality, and shortens the release schedule. It also reduces the cost impact on the product development due to quality issues that could be identified in the later stages of product development. ASPICE provides a benchmark for suppliers to ensure the stability of their process and products leading to an overall improvement in the industry.

OEMs use ASPICE as an input for their supplier selection. From the OEM's point of view, ASPICE helps them to assess the supplier's process quality capability during the supplier selection. OEMs can define their own system development process to be ASPICE complaint which will help to asses and improve the process capability.

3.2. HISTORY AND MOTIVATION

The modern-day vehicles have a large number of electronic control units which substitute the traditional mechanical and hydraulic systems. The mechanically controlled functions are replaced by the ECUs. 80 to 90 percentage of the innovation in automobiles is based on electronics and software in which 50 to 70 percentage of ECU development cost is for software.

Since the year 2000 after the introduction of ECUs in vehicles the number of recalls in German market for the cars had an enormous increase. Owing to this situation, there arose the need of a new industrial standard to assess the developmental processes which will help the OEMs to find the defects in the earlier stages of development.

The ASPICE was introduced first in 2001 as a variant of ISO 15504 (SPICE) by the AUTOSIG (Automotive Special Interest Group). This AUTOSIG incorporate the SPICE User Group, the Procurement Forum and German automotive constructors namely, Audi, BMW, Daimler, Porsche, Volkswagen, and international automotive manufacturers like Fiat, Ford, Jaguar, Land Rover, and Volvo.

The ASPICE V.3.x is the currently used version which was developed by the VDA (Verband der Automobilindustrie) working group 13 comprising of BMW Group, BOSCH Group, Continental Automotive Systems, Daimler AG, Ford Werke GmbH (representing also Volvo Car Corp.), KNORR-BREMSE Group, VOLKSWAGEN AG and ZF Friedrichshafen AG. The ASPICE V.3.1 will be explained in detail in the upcoming sections.

Automotive SPICE V.4.x is under development and is being developed by the VDA QMC working group 13 involving Audi, BMW Group, BOSCH Group, Continental Automotive Systems, Daimler AG, Ford Werke GmbH (representing also Volvo Car Corp.), Infineon Technologies, KNORR-BREMSE Group, VOLKSWAGEN AG and ZF Friedrichshafen AG.

3.3. GOALS AND REQUIREMENTS

APICE builds on the Verification and Validation Model or the V- Model [34]. The V-Model requires aa testing phase corresponding to each stage of development. It also requires a rigid evaluation to guarantee continuous assessment and development. The main goal of ASPICE is to benefit both the providers and clients.

ASPICE benefits the providers by eliminating potential problems at the initial stages. It benefits the clients by assuming a meticulous approach to both ideation and development.

Ensuring continuous innovation and product development at every stage is an additional goal of ASPICE.

3.4. PROCESS CAPABILITY DETERMINATION

Process capability determination by using process assessment model is based on a two-dimensional framework [34].

3.4.1. Process Dimension or Process Reference Model

3.4.3. Capability Dimension or Capability Levels

The capability levels are sub-divided into process attributes which provides the measurable characteristics of process capability. Processes are chosen from a process reference model by the Process Assessment Model (PAM, defined in next sessions) and supplements with indicators that support the collection of objective evidence. These evidences enable an assessor to assign ratings for processes with respect to the capability levels.

The PAM consists of process capability indicators and process performance indicators where the process reference model consists of domain & scopes, process purposes, and process outcomes.

3.4.1. PROCESS REFERENCE MODEL

In Process Reference Model (PRM) the processes are grouped by process categories. In the second level of topology, they are further grouped into process groups according to the type of activity they address. PRM defines all ASPICE processes to be befitting in well-defined automotive ECU development. It can be either software or embedded systems or both. The process refence model is classified into three process categories [34].

3.4.1.1. PRIMARY LIFECYCLE PROCESSES GROUP

The processes that are used by the customer while acquiring products from supplier and by the supplier while responding and delivering products to the customer or while acquiring products from their own suppliers are categorized as primary lifecycle processes. This involves the engineering processes needed for specification, design, development, integration and testing.

The primary lifecycle processes include Acquisition process group, Supply process group, System engineering process group and Software engineering process group. The concentration will be on System and Software engineering process groups for the project [34].

3.4.1.1.1 ACQUISITION PROCESS GROUP

The operations that are carried either by the customer or by the supplier while acting as a customer for its own suppliers in order to acquire a product or service are included in the Acquisition process group (ACQ) [34].

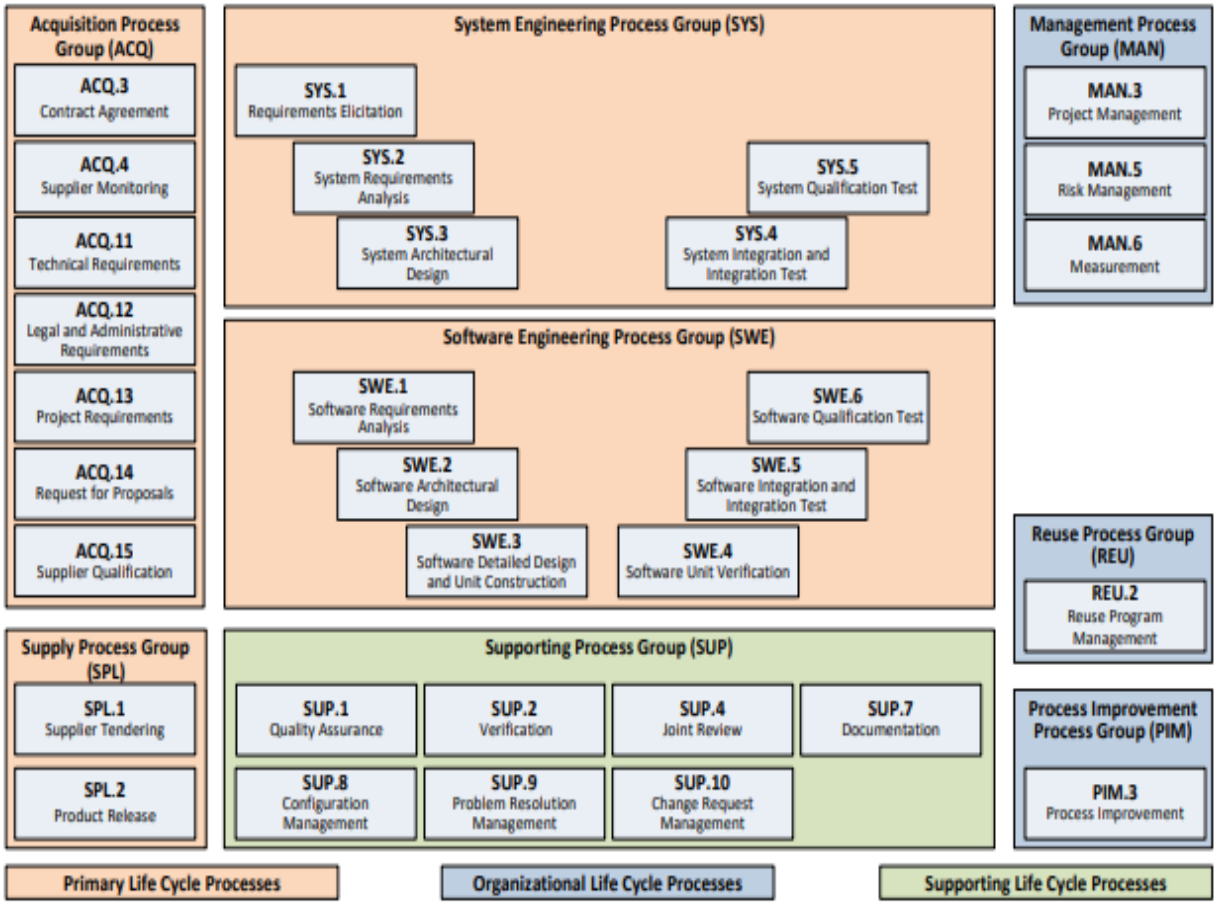


Figure 3.1 : ASPICE Process Reference Model

3.4.1.1.2 SUPPLY PROCESS GROUP

The operations carried out by the provider in order to offer a service or product make up the supply process group (SPL) [34].

3.4.1.1.3 SYSTEM ENGINEERING PROCESS GROUP

The elicitation and management of external and internal requirements, the definition of the system architecture, and the integration and testing at the system level is all covered by the System Engineering process group (SYS) [34].

3.4.1.1.4 SOFTWARE ENGINEERING PROCESS GROUP

The Software Engineering process group (SWE) includes procedures for managing software requirements generated from system requirements, creating the appropriate software architecture and design, and putting the software into use, integrating it, and testing it [34].

3.4.1.2. ORGANIZATIONAL LIFECYCLE PROCESSES GROUP

The processes that develop products, process and resource assets comes under the Organizational lifecycle process group. The main aim of these processes is to help the organization achieve its's business goals. This process category includes the Management process group, the Process improvement process group, and the Reuse process group [34].

3.4.1.3.1. MANAGEMENT PROCESS GROUP

The Management process group (MAN) consists of processes that may be used by anyone who manages any sort of project or process within the life cycle. The management process group consists of Project management, Risk management, and Measurement [34].

3.4.1.3.2. PROCESS IMPROVEMENT PROCESS GROUP

One process covered by the Process Improvement Process Group (PIM) includes techniques to enhance the operations carried out by the organizational unit [34].

3.4.1.3.3. REUSE PROCESS GROUP

One procedure to methodically take advantage of reuse opportunities in an organization's reuse activities are covered by the Reuse Process Group (REU) [34].

3.4.1.3. SUPPORTING LIFECYCLE PROCESSES GROUP

Supporting lifecycle processes as the name implies are the processes that may be employed by any other process at different points in the product development life cycle. It includes Quality Assurance, Verification, Joint Reviews, Documentation, Configuration Management, Problem Resolution Management, and Change Request Management [34].

3.4.2. CAPABILITY LEVELS

The Capability level in ASPICE can be defined as the set of process attributes that work together to provide a major enhancement in the ability to perform processes. These levels contribute to a coherent way of progressing through improvement of the capability of processes [34]. The ASPICE standard is rated from 0-5 with definitions as follows:

- **Level 0: Incomplete Process**

  In this level an organization can at most "partially" achieve ASPICE requirements and should focus more on managing basic tasks than meeting higher standards.

- **Level 1: Performed Process**

  In this level an organization can either nearly or entirely deliver the standard requirements but may have gaps in the processes.

- **Level 2: Managed Process**

  In this level the organization can reliably deliver the work products and nearly or entirely achieve the ASPICE standards along with the work products.

- **Level 3: Established Process**

  In this level, the performance standards have already been established and set for the organization. The organization will be continuously evaluating and learning from these established standards.

- **Level 4: Predictable Process**

  This level is achieved by an organization when the established processes operate predictively within defined limits to achieve the desired outcomes. In this phase the organization measure, record and analyze outcomes to enable objective evaluation.

- **Level 5: Innovating Process**

  The Innovative level is achieved when the predictable process is continually improved to respond the organizational change.

The ISO/IEC 33020 measurement framework offers a defined rating scale with an option for refinement, various rating procedures, and various aggregate methods depending on the class of the assessment to facilitate rating of process attributes. A process attribute is a measurable characteristic of a process capability inside this framework for process measurement. A rating for a process attribute indicates how well the process attribute for the evaluated process has been achieved.

Ratings may be summarized over one or two dimensions while conducting an assessment [34]. For instance, while rating a

- Process attribute for a certain process, one may combine ratings of the corresponding process (attribute) outcomes; in this case, the combining will be done vertically (in one dimension).

- Process (attribute) outcome for a particular process attribute over numerous process instances. For this aggregation to be performed as a horizontal aggregation (one dimension), the ratings of the associated process instances must be taken into consideration.
- For a given process attribute, one may combine the ratings of all the process (attribute) results for all instances of that process; this aggregation will be carried out as a matrix aggregation across the entire range of ratings (two dimensions)

The process assessment model provides indicators to determine the presence or absence of process outcomes and process attribute outcomes (achievements) in the processes that have been instantiated for projects and organizational units [34]. These indicators offer direction to assessors in gathering the essential factual data to back up capability assessments. They are not meant to be viewed as a set of guidelines that must be adhered to.

An assessment gathers factual facts to determine whether or not process outcomes and process achievements are present. All of this information was gathered through the review of work products and repository content from the processes that were being evaluated, as well as testimony from the process managers and performers. To establish the link to the pertinent process outcomes and process attribute accomplishments, this evidence is mapped to the PAM indicators.

Two categories of indicators exist:

- Process performance indicators (applies only to capability Level 1 capabilities). They give a hint as to how much the process results were achieved.
- Process capability indicators, which are relevant to Capability Levels 2 through 5. They give a hint as to how fully the process attribute accomplishments have been realized.

3.5. EARLY-STAGE PRODUCT ARCHITECTURE VERIFICATION

The ASPICE standard is required to be followed by the automotive hardware and software manufacturers all around the world to ensure a certain level of quality for their products before delivery. The ASPICE standard suggests the early-stage verification of product architecture, especially way before the production so as to ensure that there are less errors being generated and the recall of the products can be avoided.

Assume the case of product architecture verification is not performed at the beginning and the product is sent for production. Once it is on chip and a major bug is discovered, it is way too costly for the manufacturer to recall a batch of production.

In certain situations, if the bugs are discovered after the delivery of the product and the product already reached the end user, in this case the manufacturer is even responsible for providing compensation to the end user as per law.

However, if the product architecture verification is performed in the beginning the bugs can be discovered and resolved once it is a software design and this does not cost monetary losses to the manufacturer. Also, the integrity of the delivered products will be guaranteed.

3.6. DETAILED STUDY OF PRODUCT ARCHITECTURE

The product architecture of an automotive Microcontroller developed by Infineon is used for the study and verification. An already available MATLAB model of the product architecture is taken for the study. MATLAB provides a feature to develop state flow diagrams with which we can generate the product architecture of the system similar to a flow chart and decide the branching and functionalities.

The product architecture comprises of 16 inputs and 15 outputs. The inputs of the system are namely, tm_clg,tms, tmaux, swclk,tm_pw_ok, hard_reset, rst_clr, cfs_fail, ram_fail, cfs_test_fail, cfs_user_fail, bsi_unlock,bsi_ram, bsi_nad, soft_reset, and POR. The output terminals are debug_mode, endless_loop, user_mode,rst_cold, rst_hot,rst_warm, bist_pass, bist_done, lest_en, tmaux_latch, swclk_latch, init_pass, nvm_done. The inputs and outputs will be explained in detail in the upcoming chapters as the functionalities and inner block of the product architecture will be defined.

The product architecture of an Infineon automotive microcontroller studied for verification comprises of 11 functional blocks. The functional blocks are Startup, checkRAM, checkNVM, config, reset, branch, and different active modes and resets. Each block and its functionalities, how to verify them, and the boot modes of the system will be explained in chapter four.

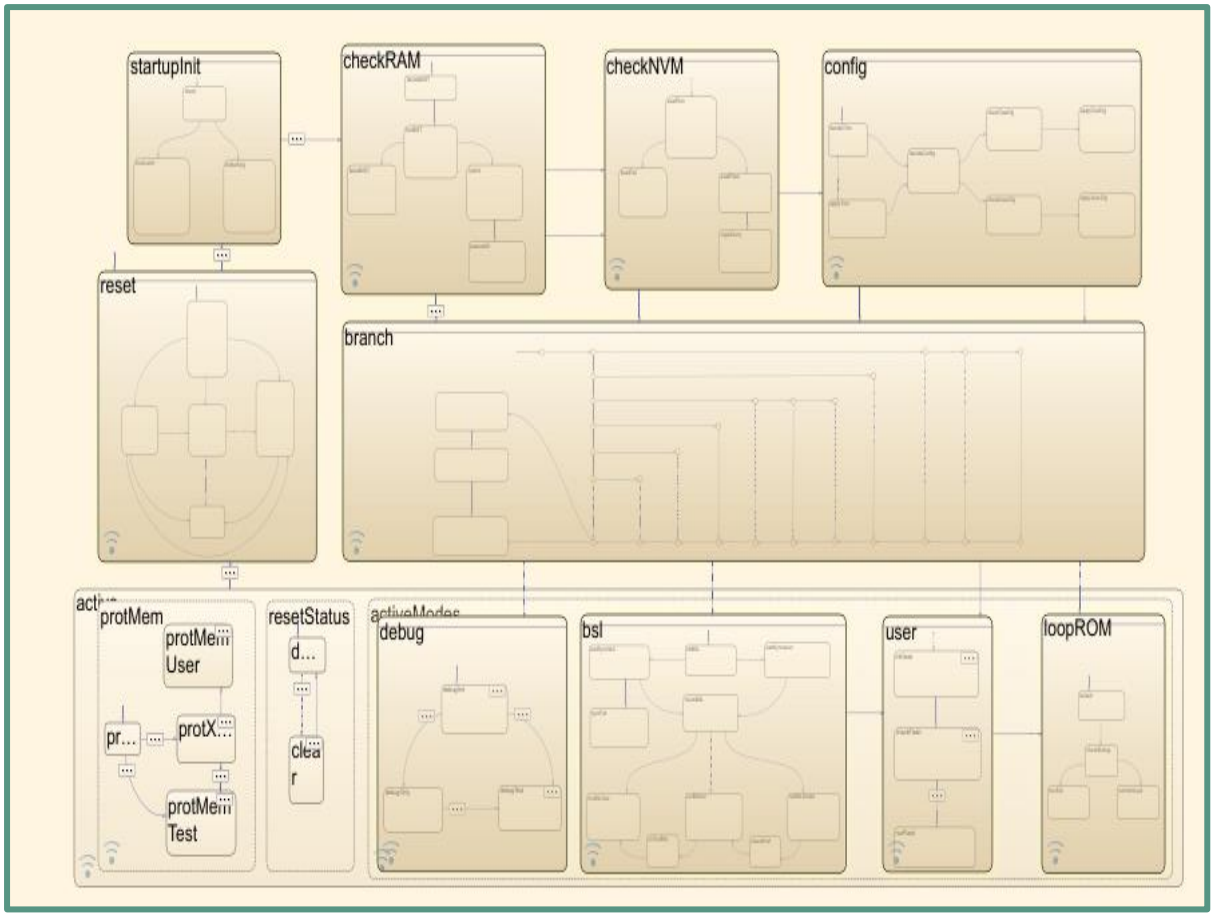


Figure 3. 2: Product Architecture State Flow Diagram

The inputs enter the system through the reset block and the system checks whether any reset is happened in the past, if yes which kind of reset is happened. According to the inputs entering the system the reset block transits to the startup and active blocks. The startup init block where the initial startup occurs. From there, the system checks the RAM and the Non-volatile Memory of the automotive microcontroller.

The active mode block comprises of the blocks of four modes of operation namely the debug mode, user mode, bsl mode and bsl far mode. According to the inputs fed into the system, the system chooses which block should be executed inside the active mode block.

The components inside each block and the explanation on detailed execution is avoided to keep the confidentiality of the project. Therefore, we are not explaining what actually happens inside the product architecture of this automotive microcontroller in this chapter, however the detailed study of each block in the system is conducted to understand the behavior of the system in each mode of operation. This detailed study has helped in finalizing the value of inputs to be fed in each mode of operation.

CHAPTER 4: THEORETICAL ANALYSIS

4.1. VERIFICATION APPROACHES

Software or Hardware verification is a vast field of research and there are different methodologies available for the verification of an automotive microcontroller [35][36]. A system can be verified based on different aspects such as the functionalities, the behavior of the system, the inputs of the system, the outputs of the system, or the use case scenarios. We can make a large list of possible approaches to verify a system but which must be the best approach to analyze a system efficiently is a highlighted question that arises.

As already mentioned in the introduction of this research, no previous research activities known that claims clearly that if your system has the following features the best way to analyze your system will be a particular method. One major aim of this research is to find out an answer for this. For this purpose, four most widely used verification approaches such as verification based on Functionalities, Use-case scenarios, Inputs and Outputs are considered.

4.1.1. FUNCTIONALITIES

The verification of a system based on functionalities is considered to be one of the best methods to approach a system under test. It is believed that once all the functionalities are verified hundred percent coverage will be obtained. But, for verifying the functionalities an engineer can think of hundreds or thousands of test scenarios and there cannot be a general trend to be observed to verify a system if we follow this approach. Also, we cannot state that we will obtain hundred percent coverage with least possible effort with this approach.

4.1.2. USE CASE SCENARIOS

The use case scenarios are the actions performed by the end user of a product. Similar to the test scenarios designed considering the functionalities, if we consider the use case scenarios, we will never be able to find all the possible use case scenarios for a product. The end user can use the system in many unpredictable ways that even a verification engineer can think of.

This is because the verification engineer will be having adequate knowledge of the system but a first-time user does a black box testing of the system and he can create unpredictable situations in which the system may show unpredicted behavior.

Thus, we cannot ensure that with all the use case scenarios that a verification engineer depicts the end user will only use the system in the same way and there will never be a major bug hidden. Also, the use case scenarios can change from product to product or even from one release version to another as the functionalities or behavior may change or add up. Thus, a general trend cannot be obtained and we cannot ensure hundred percent coverage using this approach.

4.1.3. INPUT APPROACH

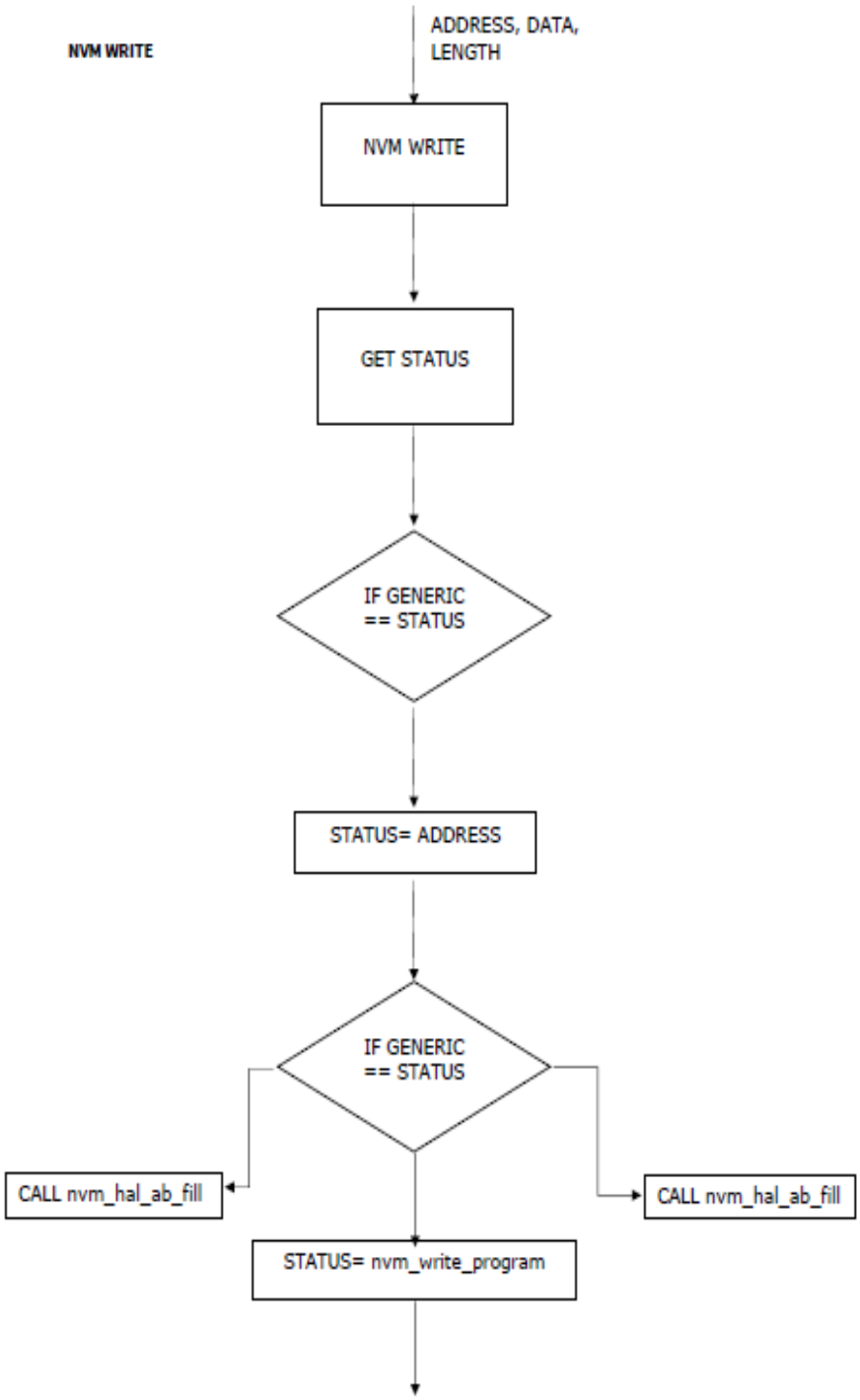
With the input approach we are concentrating on the number of inputs a system has and we design test cases based on the number of inputs. As we know the number of inputs a system has from the initial stage of test design this approach can be utilized to finalize a general trend. How this is attained will be explained in detail in coming sections.

4.1.4. OUTPUT APPROACH

Similar to the input approach in output approach also the number of outputs a system has is initially known and the verification engineer can decide whether to concentrate on the outputs to design the test cases.

Thus, among these four major approaches as our aim is to finalize a general trend for every system under test – the best approach we are concentrating on the input and output approaches. For this purpose, a function to read and write to a non-volatile memory (NVM) written in C language is chosen.

The C language function is already a used functionality in all microcontrollers (products of Infineon) and the function is converted to a flow chart first. The designed flow chart of the C function is displayed in the figure below.


NVM WRITE
ADDRESS, DATA,
LENGTH
NVM WRITE
GET STATUS
IF GENERIC
== STATUS
STATUS= ADDRESS
IF GENERIC
== STATUS
CALL nvm_hal_ab_fill
CALL nvm_hal_ab_fill
STATUS= nvm_write_program

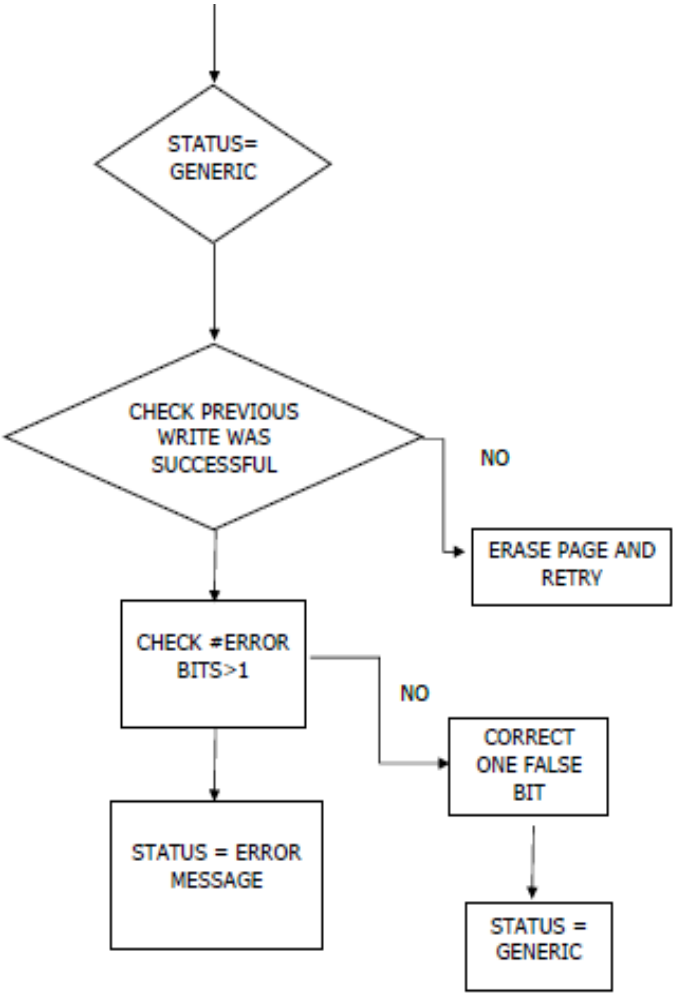


Figure 4.1: Flow chart of a read/write functionality of an NVM

The flow chart is further converted into a MATLAB state-flow diagram of a system with three inputs and one output with four equivalent classes. Now we have to study the system that we have designed manually to understand the maximum possible test cases that can be created focusing on both input and output-based approaches and which will be our best approach. The state-flow diagram of the system designed is displayed in the image below.

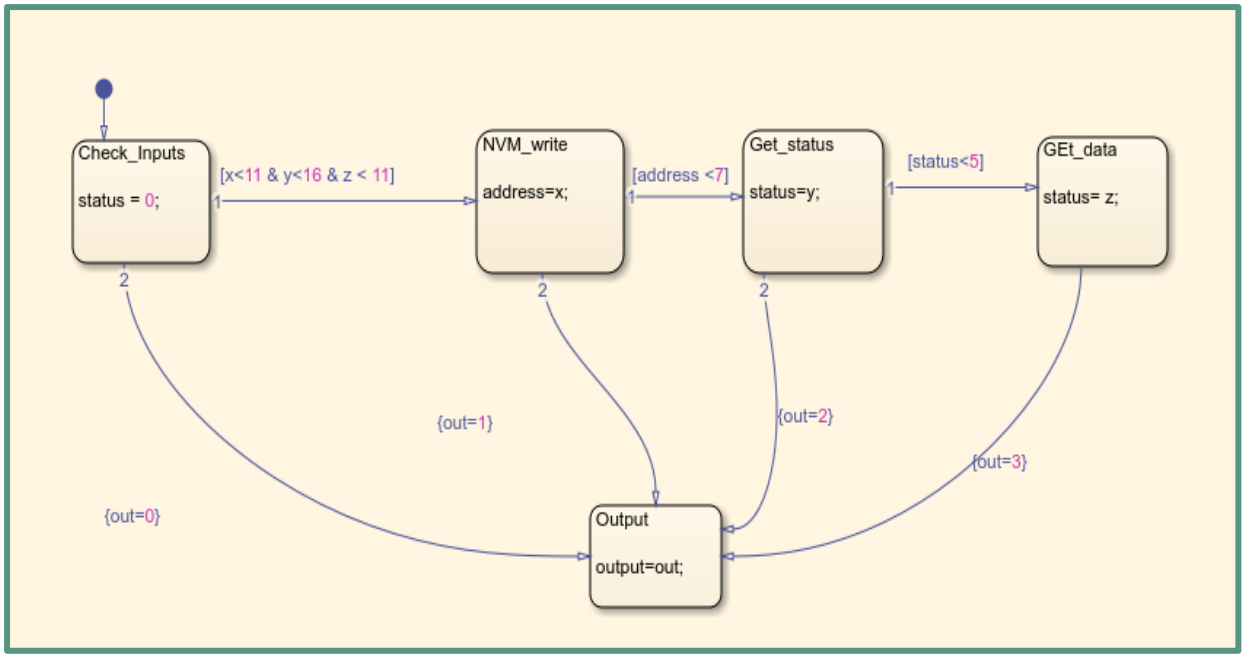


Figure 4.2 State-flow diagram of the system under test

Here from the figure, we can understand that the system is having only one output, but the output has four equivalent classes. Similarly, the input also has a number of equivalent classes. So, first we need to understand what an equivalent class is and then how the number of equivalent classes will affect the total possible number of test cases a system has.

The equivalence relation is a set of pairs of equivalent things. In our perspective, Input Equivalent Class is a range of input values that provides the same result irrespective of the value. Output Equivalent Class is the set of output data a single port can provide with respect to the variations applied to the inputs. so according to the definition of input and output equivalent classes, our system has 4 output equivalent classes and three equivalent classes for input one (x), input two(y) and two equivalent classes for input three (z).

If we concentrate on the outputs there are only four possible outputs that we can obtain whatever use case scenario or functionality check we perform. The system will provide us with any one of these four outputs. Thus, to verify the system we just require four test cases. If we concentrate on the input equivalent classes, we require more than four test cases.

Now, with this method we need to find a common trend or a generalized equation for the maximum possible test cases that can be designed for a system when the input and output equivalent classes are known. For this purpose, we are going to evaluate systems with different input equivalent classes and different output equivalent classes.

The study starts with two different cases to be analyzed, that is when all inputs of the system affect all outputs of the system and when all inputs of the system does not affect all outputs of the system.

4.2. ALL INPUTS OF THE SYSTEM DOES NOT AFFECT ALL THE OUTPUTS

When all inputs of a system do not have an effect on all the outputs, we need to check individually each input-output combination. Assume a system, having two inputs and two outputs and both the inputs simultaneously have no effect on each output. Thus, to test the effect of each input on each output terminal we need to individually test all the possible distinct combinations. Therefore, for a system in which all inputs of the system do not affect all outputs the maximum possible test cases will be the maximum distinct possible combinations.

For a system with two inputs and two outputs, all possible test cases will be (In1, Out1), (In1, Out2), (In2, Out1), (In2, Out2), (In1, In2, Out1), (In1, In2, Out2). So, the maximum possible test cases in this case can be calculated by simply applying the relation to find the maximum distinct combinations that is $(2_{C_1} + 2_{C_2})$ if there is only one output. When there is more than one output the relation changes into $(2_{C_1} + 2_{C_2}) * \text{no of outputs}$.

Thus, **maximum possible test cases** = $(\boldsymbol{in_{C_1}} + \cdots + \boldsymbol{in_{C_n}}) * \textbf{no of output}$

where i is the number of inputs and n is the number of equivalent input classes.

4.3. WHEN ALL INPUTS AFFECT ALL OUTPUTS

Now on considering the systems in which when all inputs simultaneously affect all the outputs, that means only applying input one and keeping input two vacant will have a major effect on the output of the system we need to begin with the number of inputs and number of outputs. Here, there are two different possibilities to start with. The number of inputs of the system can be equal to the number of outputs of the system and the number of inputs of the system are not equal to the number of outputs of the system.

4.3.1. NUMBER OF INPUTS EQUAL TO THE NUMBER OF OUTPUTS

When the number of inputs of the system is equal to the number of outputs of the system we need to find the maximum possible test cases to design the test suite. For finding the general trend, we can start from a system having one input and one out and increase the number of inputs and outputs gradually.

All the possible combinations of test cases are calculated manually and displayed in a tabular format in figure 4.3.1.

# Inputs	# Outputs	# Test Cases
1	1	1
2	2	4 ([1,1],[1,2],[2,1], [2,2])
3	3	9
4	4	16

Table 4.1: Maximum possible test cases when number of inputs are equal to number of outputs.

4.3.2. NUMBER OF INPUTS NOT EQUAL TO NUMBER OF OUTPUTS

The second scenario is when the number of inputs of the system is not equal to the number of outputs of the system. In this case we can have two possibilities to work on, either the number of inputs is greater than number of outputs or the number of inputs is less than number of outputs of the system. Both the cases are analyzed manually and all the possible test cases are noted down in a tabular format for both the cases. Here also we follow the same procedure we have followed in case 4.2.1.

# Inputs	# Output	# Test Case
1	2	2 ([1,1],[1,2])
1	3	3 ([1,1],[1,2],[1,3])
2	3	6 ([1,1],[1,2],[1,3], [2,1],[2,2],[2,3])
3	4	12
2	4	8

Table 4.2: When number of inputs are less than number of outputs of the system

# Inputs	# Output	# Test Case
2	1	2 ([1,1],[2,1])
3	2	6 ([1,1],[2,1],[3,1],[2,1],[2,2],[2,3])
3	1	3
4	3	12
4	2	8

Table 4.3. When number of inputs are greater than number of outputs of the system

To move forward and make the analysis easier go ahead with the assumption that in the systems that we are going to analyze all inputs have a simultaneous effect on all outputs or all outputs are dependent on all inputs. Till now we were only discussing about the number of inputs and number of outputs, but the function that we chose to analyze has both input and output equivalent classes. Thus, we need to analyze the system in which all inputs have an effect on all outputs and both input and output equivalent classes are present. Consider three scenarios to begin with

4.3.3. Multiple input equivalent classes are present in a system with single output equivalent class

4.3.4. Multiple output equivalent classes are present in a system with single input equivalent class

4.3.5. A system with multiple input and output-equivalent classes.

4.3.3. MULTIPLE INPUT EQUIVALENT CLASSES AND SINGLE OUTPUT EQUIVALENT CLASS

Consider a system having two inputs and one output as displayed in figure 4.3.3.1 for the evaluation.

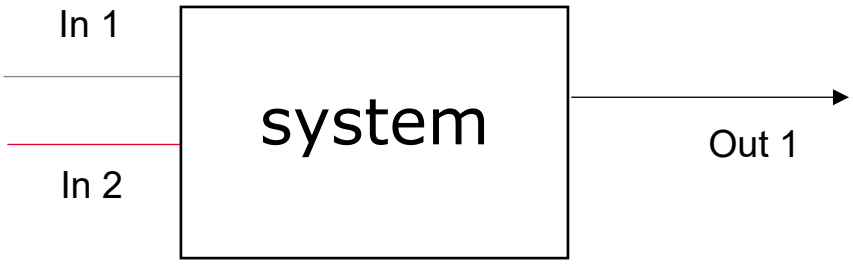


Figure 4.3.3.1. System with two inputs and one output

Here, In1 and In2 are the two inputs of the system under test and Out 1 is the output of the system. Now as per our first assumption both inputs that is In1 and In2 simultaneously affect Out 1. To analyze the first scenario, a system with multiple input equivalent classes and single output equivalent class, consider that In1 is having two input equivalent classes and In2 is also having two equivalent classes.

Let us mark the equivalent classes for input one (In1) as In11 and In12. For the second input, In2 mark the equivalent classes as In21 and In22. As per the condition the output is having only one equivalent class thus there is no need to specifically mark the output equivalent class. Thus, we can call the output equivalent class as Out1. The system with both input-equivalent classes is displayed in the diagram below.

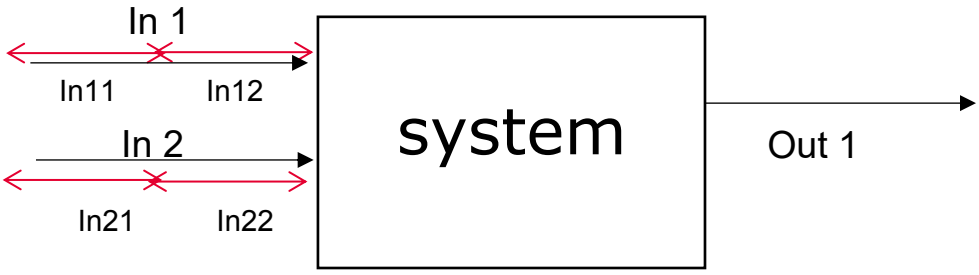


Figure 4.3.3.2. System with two inputs and input equivalent classes

Now, we have to find all the possible test cases that can be performed on this system to realize all the possible outputs that can be obtained from the system. When input one is applied to the system, we have two possibilities, either we are applying In11 or In22. Similarly, for input two we are applying either In21 or In22. Thus, to test the system we need to analyze all the combinations between input equivalent classes.

Here, we have two equivalent classes for each input. Therefore, the maximum possible combinations that we can obtain from this scenario is four. The maximum test cases which can be performed on this system are as follows:

- When In11 and In21 are applied to obtain Out 1
- When In11 and In22 are applied to obtain Out 1.
- When In12 and In21 are applied to obtain Out 1.
- When In12 and In22 are applied to obtain Out 1.

In this case we can conclude that the maximum possible test cases are the maximum possible distinct combinations of input equivalent classes. However, here there is only one output port available for our system. Our assumption was there is only one equivalent class for the output of the system. This does not mean that the system will only have one output. There will be a number of outputs for the system but the number of equivalent classes for each output is one.

For finding a general equation in this scenario we need to perform more analysis on the system when there are:

- More than one output for the system and,
- Systems with more than two input equivalent classes.

This is because with considering only one scenario in which a system having two equivalent classes does not provide us any information on how we can find the maximum possible test cases in a scenario where there are more than two equivalent classes for each input. For this purpose, we have to increase the number of equivalent classes for each input and manually calculate all the possible test cases that can be performed while focusing on the input approach. Then finally we have to conclude with the general equation which is derived from the calculations performed.

Note that we are increasing the number of equivalent classes for each input and for the convenience of evaluation we are keeping the number of inputs as two. Now consider a system with two inputs in which input one is having three equivalent classes and input 2 is having two equivalent classes. The system has a single output with one equivalent class as per the initial condition. Let the equivalent classes of Input 1 be In11, In12, and In13. The equivalent classes for input 2 be In21 and In22. The maximum possible test cases will be:

- In11 and In21 applied to get Out 1.
- In11 and In22 applied to get Out 1.
- In12 and In21 applied to get Out 1.
- In12 and In22 applied to get Out 1.
- In13 and In21 applied to get Out 1.

- In13 and In22 applied to get Out 1.

The maximum possible test cases that can be developed to test a system with two inputs and one output in which input one having three equivalent classes and input two having two equivalent classes are six. On increasing the numbers further and analyzing we can conclude that if there are only one output equivalent class the maximum possible test cases will be the product of the input equivalent classes available.

Our second condition is to increase the number of outputs. Therefore, consider the first system with two inputs with two equivalent classes for each input and increase the number of outputs to two.

The maximum possible test cases in this scenario will be:

- In11 and In21 applied to get Out 1.
- In11 and In22 applied to get Out 1.
- In12 and In21 applied to get Out 1.
- In12 and In22 applied to get Out 1.
- In11 and In21 applied to get Out 2.
- In11 and In22 applied to get Out 2.
- In12 and In21 applied to get Out 2.
- In12 and In22 applied to get Out 2.

Now on analyzing this, we can see that there are redundant test cases available. On eliminating the redundancy, we get four test cases in this scenario also. So, we can conclude that the number of outputs does not affect our calculation when the outputs are having only one equivalent class. Therefore, from the analysis we can conclude with a general equation as explained below.

The maximum possible test cases while concentrating on input approach when a system is having only one equivalent class for each approach will be as follows.

$$\#\text{Test Cases} = \left[\prod_{i=1}^{n}(no\ of\ equivalent\ classes\ of\ In_i)\right] * \text{Out}$$

On eliminating the redundancy, the equation becomes,

$$\#\text{Test Cases} = \left[\prod_{i=1}^{n}(no\ of\ equivalent\ classes\ of\ In_i)\right.$$

Where ‘n’ is the number of inputs and ‘i’ is the number of equivalent classes for corresponding input.

We can conclude this analysis by realizing the graphical representation of the maximum possible test cases a system can have with different number of input equivalent classes. The equivalent classes for input 1 and input 2 are represented on x and y axes and the maximum possible test cases for each combination is represented on the z axis.

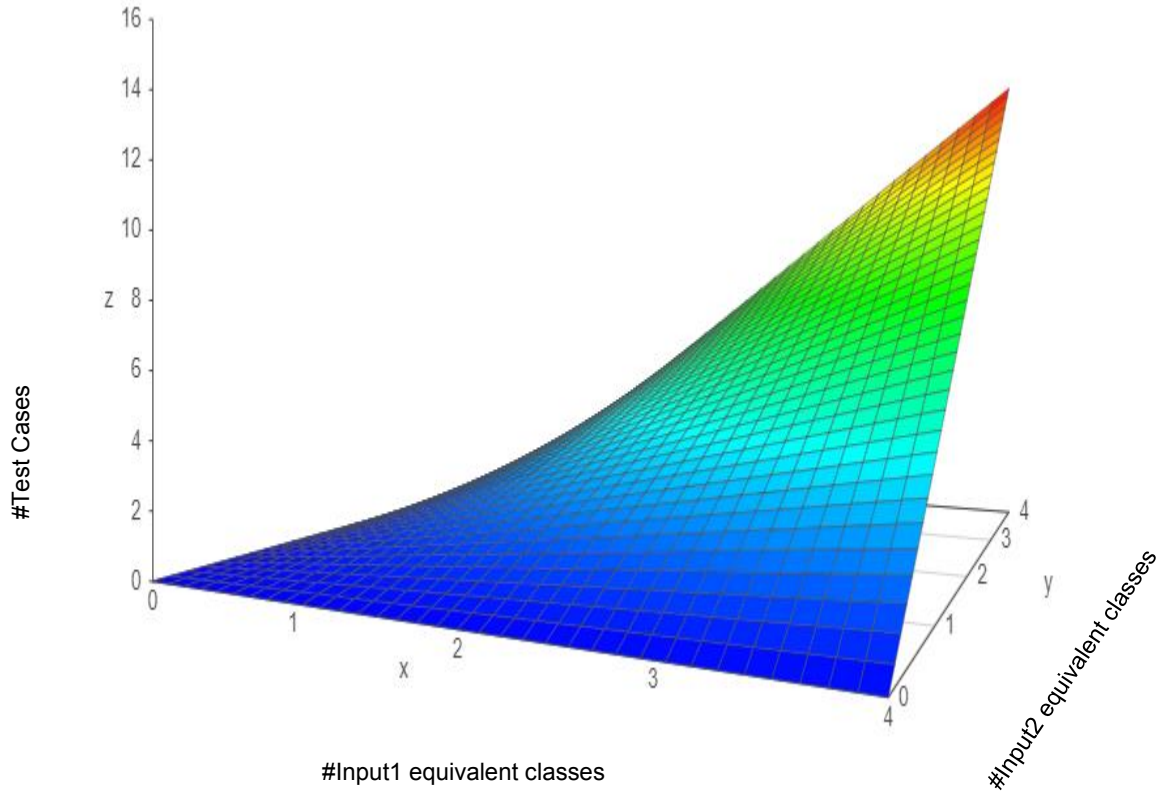


Figure 4.3.3.3. graphical representation of the maximum possible test cases a system can have with different number of input equivalent classes

The graph represents the maximum possible test cases when input one and two are having 1, 2, 3, or 4 equivalent classes. As we have eliminated the redundancy, the general equation satisfies for each real point in the graphical representation.

4.3.4. MULTIPLE- OUTPUT AND SINGLE INPUT EQUIVALENT CLASS

In scenario 4.3.3, we have analyzed a system with multiple input equivalent classes and single output equivalent class. In real life scenario there will be more than one equivalent class present for each output. Therefore, we need to find what will be the general trend when there are more than one output equivalent class present. For the convenience of the analysis to be performed

we are considering that there is only one input is available and there is only one input equivalent class.

To begin with, consider the following system with one input with one input equivalent class and one output with two output equivalent classes. Let In1 be the input and Out11 and Out12 be the output equivalent classes of output Out1.

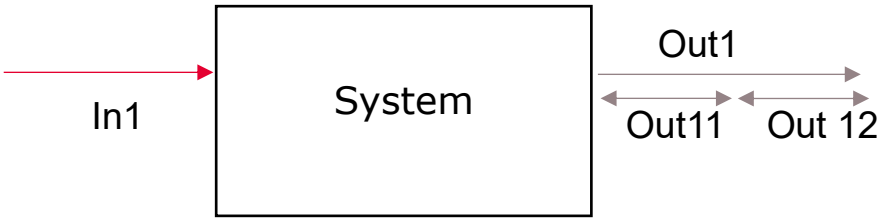


Figure 4.3.4.1. System with one input and two output equivalent classes

Here we have the possibility of two test scenario, either we will be applying input one and we can get Out 11 as the output or apply input one and get Out12 as the output. Therefore, the maximum possible test cases to obtain hundred percent coverage on the system is two. Now we have to increase the number of outputs to concentrate more on the output approach.

Consider the system with two outputs and each output is having two equivalent classes respectively. Let the outputs be Out 1 and Out 2. The output equivalent classes for Out 1 be Out11 and Out12. The equivalent classes for Out 2 be Out21 and Out22. Our first assumption is valid in this scenario also, that is all inputs affect all outputs simultaneously. Therefore, on applying input one we can obtain the following possible combinations of outputs.

- Input one applied to obtain Out11 and Out21.
- Input one applied to obtain Out11 and Out22.
- Input one applied to obtain Out12 and Out21.
- Input one applied to obtain Out12 and Out22.

Here the maximum test cases which can be obtained when a system with two outputs each having two output equivalent class and two inputs are four. Now increase the number of output equivalent classes and calculate the maximum possible test cases that can be obtained for each scenario.

Note that the number of outputs is kept constant as two and the number of output equivalent classes are increased. Increasing the number of outputs does not create any change in the final

formula obtained. Only the number of equivalent classes for each output affects the calculation. The result of this calculation is expressed in graphical format as below.

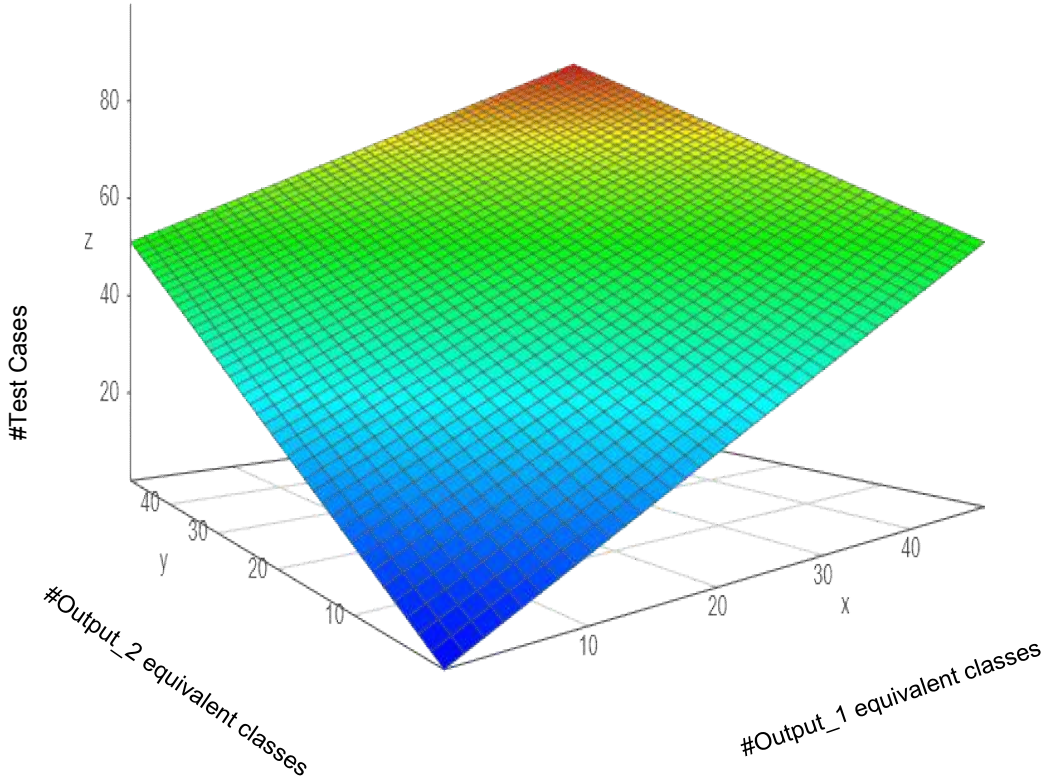


Figure 4.3.4. Graphical representation of System with multiple output equivalent classes and single input equivalent class

The x and y axes represent the number of equivalent classes for output 1 and output 2. The z axis represents the maximum possible test cases that can be derived for each value on the x and y axes. All the values are manually calculated and verified so as to find the maximum distinct test cases for each co-ordinate.

From the analysis, a general trend is observed and a mathematical equation is derived for finding the maximum possible test cases that can be developed when a verification engineer if following the output-based approach to analyze a system with multiple output equivalent classes. The formula is as follows:

Maximum possible test cases = $\sum_{i=1}^{n} \boldsymbol{number\ of\ equivalent\ classes\ for\ output_i}$,
Where the number of input equivalent class is one.

4.3.5. SYSTEM WITH MULTIPLE INPUT AND OUTPUT EQUIVALENT CLASSES

In scenarios 4.3.3 and 4.3.4 we were analyzing on the basis of making either the number of input or output equivalent classes static. This was done for the convenience of analysis and to help the test engineer to decide whether the input or the output approach allows him or her to reach the goal with minimum effort.

In real life situations there are situations where a system will have both input and output equivalent classes. Therefore, the test engineer can decide whether to focus on input-based approach, output -based approach or use the combination of both input and output-based approach to generate the maximum coverage with minimum effort.

For evaluating this scenario, consider a system with two inputs and two outputs. Let the inputs and outputs be In1, In2 and Out 1, Out 2 respectively. Let the input equivalent classes be In11, In12 for In1 and In21, In 22 for In2. The output equivalent classes will be Out11, Out12 for Out1 and Out21, Out22 for Out 2.

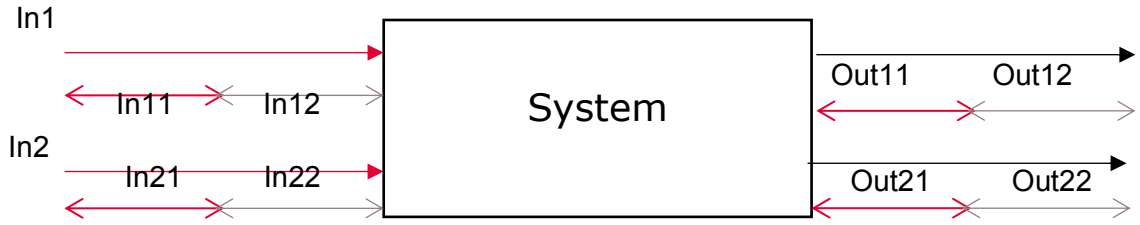


Figure 4.3.5.1. System with two input and two outputs with two equivalent classes for each input and output

Here, there are sixteen test cases generated for a simple two input two output system. Thus, we can understand clearly that how small the system be, depending on the number of equivalent classes the system can get complex at the time of verification.

By increasing the number of input and output equivalent classes and generating the maximum possible test cases for each combination a graph is plotted as follows to characterize a general trend so that we can derive a general equation. The plotted graph is displayed below.

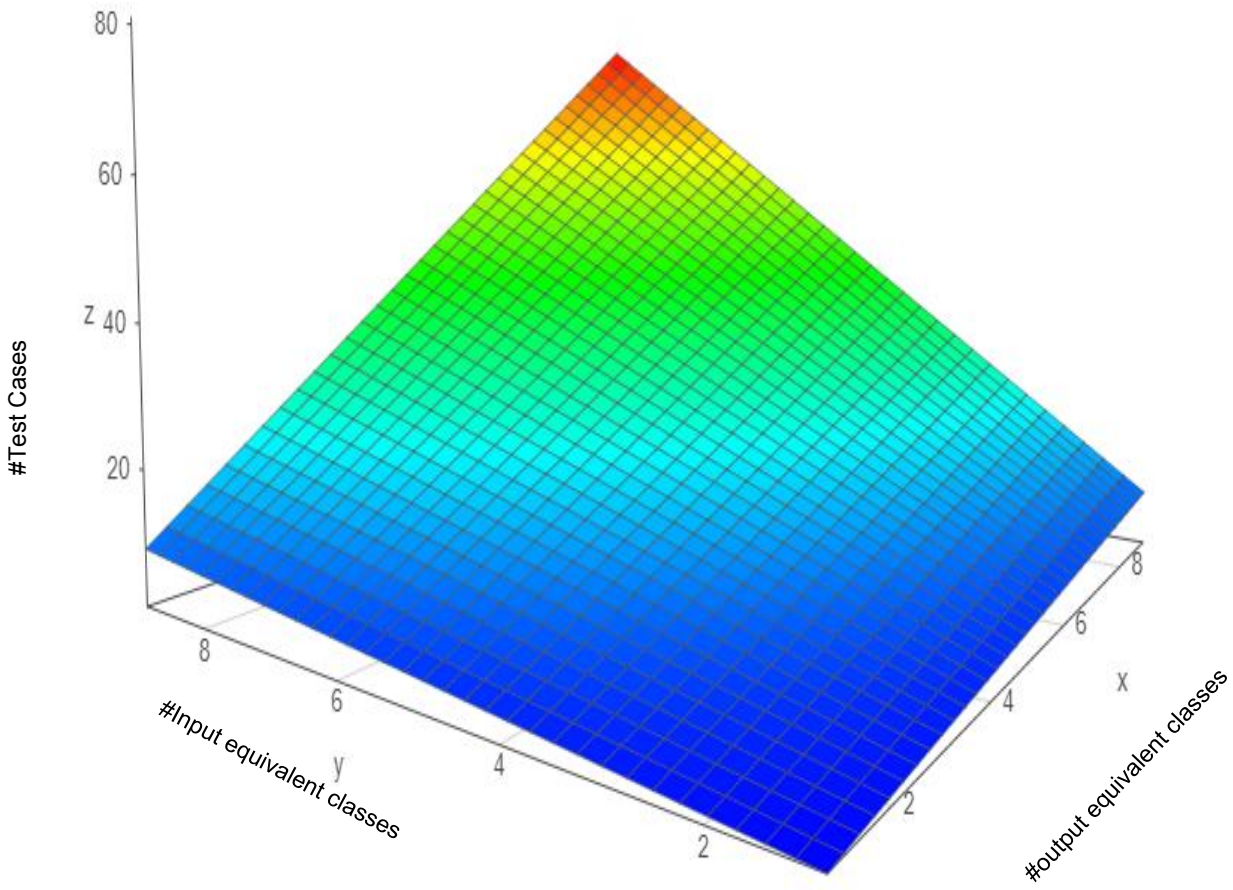


Figure 4.3.5.2. Graphical representation of maximum possible test cases for system with multiple input and output equivalent classes

The x axis represents the output equivalent classes. The y axis represents the input equivalent classes. The maximum possible test cases that can be generated are represented by the z-axis. For the convenience of plotting the graph the x and y axes represents the total sum of equivalent classes in each stage, that is it if input 1 is having two equivalent classes and input 2 is having 2 equivalent classes corresponding y axis cordinate is 4.

From the above graphical representation we can understand that for a simple system , if there are more equivalent classes present for the input and output the maximum possible test cases will be more as compared to the input-based or output-based approach.

A general trend has been plotted and a mathematical formula has been derived for calculating the maximum possible test cases that can be generated when a test engineer follows this approach. The mathematical representation is given below:

Maximum possible test cases =

$$\prod_{i=1}^{n}(no\ of\ equivalent\ classes\ of\ In_i)\ *\quad \prod_{i=1}^{m}(no\ of\ equivalent\ classes\ of\ Out_i)$$

From the theoretical analysis, we have concluded with three different set of equations which can be applied while designing the test cases for each approach that can be followed. Remember the function mentioned in figure 4.2, our main aim is to design test cases for this function using the best approach.

Now we have the theoretical equations, however we have not yet proved that the theoretical analysis is practically correct for our system under test. Therefore, we have to verify the genuinity of the formulas for each approach using this system. Chapter 4 explains the designing of test cases using the results of theoretical analysis on NVM function depicted in figure 4.2.

CHAPTER 5: RELIABILITY CHECK FOR THEORETICAL ANALYSIS

In the previous chapter we have seen that there are three main methods to approach a system under test and we also derived the mathematical formulas to calculate the maximum possible test cases that can be generated. Now we have to verify whether our formulas are true in practical scenario and we can apply it on our systems to receive desirable results. For this purpose, we are using MATLAB Simulink Test harness to design the test cases, get the coverage results and also the outputs.

The following diagram shows the system under test. We are initially using the same NVM function described in previous chapter (figure 4.2).

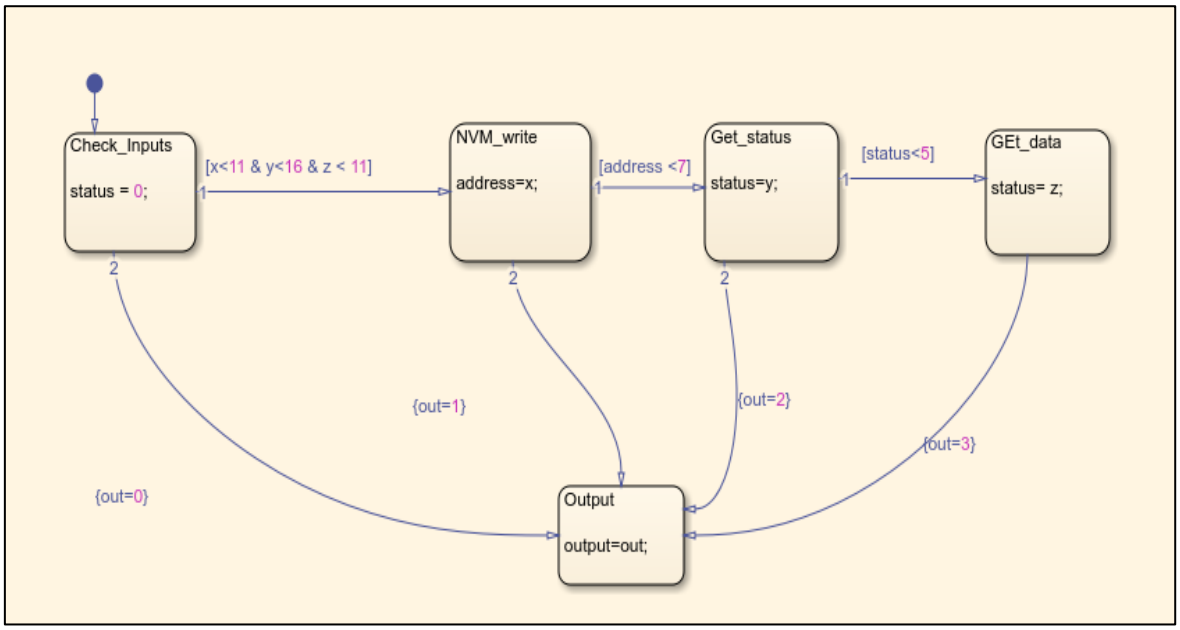


Figure 5.1: System Under Test

Here, the system has three inputs namely x, y and z. The system is having one output named out with four equivalent classes. Now we are applying our input-based approach and output-based approach on this system to find out whether our theoretical analysis is reliable and which is the best way to verify the system to have the desired coverage.

For the practical analysis to begin with, first we are focusing on the output-based approach. As the system is having only one output and the output is having four equivalent classes, as per our conditions there will be a maximum of four test cases designed to get the desired results. For the analysis we are using the test harness feature of Simulink as we have the MATLAB model of the system. Therefore, we have to generate a Simulink Test harness for the system and the generated harness is displayed below.

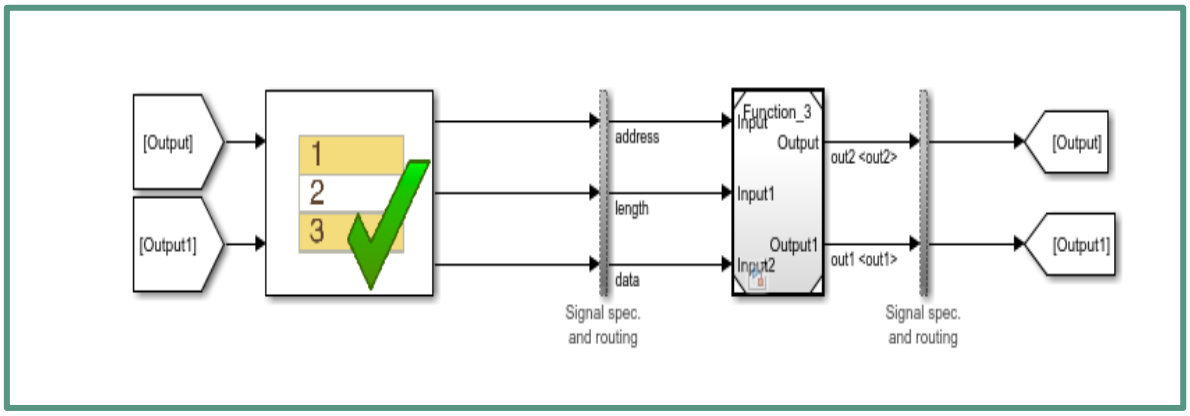


Figure 5.2: MATLAB Simulink Test harness of figure 5.1

The above diagram represents the Simulink test harness generated in which the input is given to the system through the test sequence feature of Simulink. With the test sequence we can design different scenarios. Here we have taken each scenario as one test case. Thus, we have four scenarios inside the test sequence with which we will generate all the four outputs.

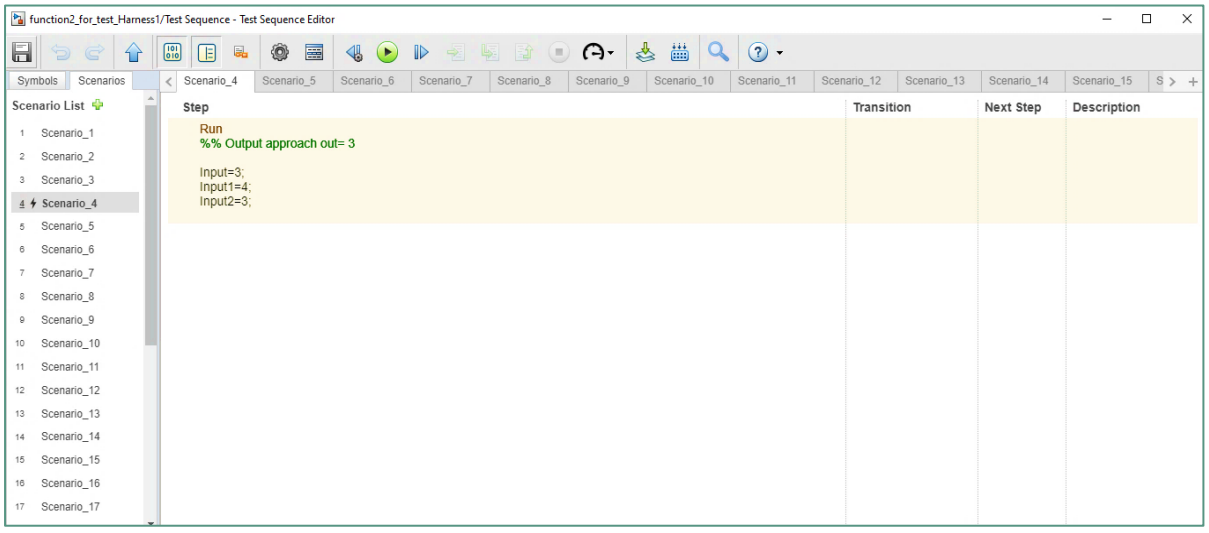


Figure 5.3: Test Sequence Editor

Figure 5.3 shows the test sequence editor of MATLAB in which we have switched the test case mode to scenarios. Scenario_1, Scenario_2, Scenario_3, and Scenario_4 shows the scenarios designed based on the output approach for the system. On checking the contents inside each scenario, we can see that we have chosen the inputs in such a way that they will give us the desired output that is, the values of x, y and z are calculated such that for scenario_1 it will give the output value as out0, for scenario_2 it will give out1 and so on.

The next step is to create a test suite inside MATLAB to run the scenarios created automatically. So, using the test manager feature inside MATLAB we can create attest suite and inside the test suite we can create a test case for each scenario. Thus, once we are executing the test suite all the test cases inside the test suite will be executed and the report will be generated. This will be explained in detail in chapter 6.

We only concentrate on the designing of test cases and the results generated in this chapter. So, we have designed four test cases on following the rule. Now on executing these four test cases we have received the following output results.

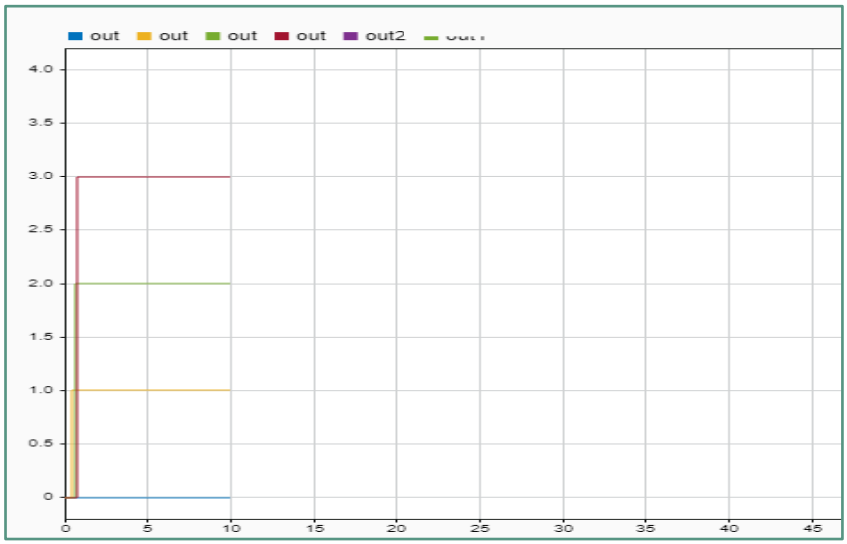


Figure 5.4: Four outputs generated using the execution of the test suite

Here, on concentrating on our output-based approach we have received all our four output equivalent classes as outputs with just four test cases. Now we need to focus on input-based approach. We need to find how many test cases will be required to receive all the output equivalent classes and the desired coverage. We will compare and analyze the coverage results of both the approaches at the end.

On concentrating on the input approach, the system has three inputs x, y, and z. Input x has three equivalent classes, y has three equivalent classes and x has two equivalent classes. These equivalent classes are found out as follows:

- If the value of x is greater than 10 it will provide out0, if the value of x is greater than 7 it will give out1 and the values 0 to 7 gives another output.

Thus, we can see that there are three different range of values for which a different set of output is received. Therefore, we can conclude that the input x is having three equivalent classes (Re-collect the definition of equivalent classes in previous chapters).

- Input Y is also having three equivalent classes. If the value of y is greater than 15 output 0 is received, if the value of y is greater than 4 the output will be Out2. For the values of y from 0 to 4 the output received will be based on the other inputs.

Thus, we have three input equivalent classes for input y. Now we have to consider input z and their equivalent classes before we finally concentrate on designing the test cases.

- Here, input z has two equivalent classes, if the value of z is greater than 10, the system will give the output as 0 and for all values of z less than or equal to 10 the output will be 3.

We have calculated all the equivalent classes for all the inputs and the next step is to concentrate on designing the test cases based on the equivalent classes of each input. We are concentrating on all possible combinations of input equivalent classes so that we will not miss any possible test scenario. There will be eighteen combinations of distinct test cases when we approach the system through input method and they are explained in detail below.

- **When x<11, y<16 and z <11**

  This is the first and the most basic condition to enter into the system. If the values of x, y and z are in this range, then only the system will function. If any of the inputs are not in this range then as we know, we will receive an error output that is 0 as output.
- **When x>10 or y<16 or z < 11**

  When the value of x, y and z are not in the range that is when x is greater than 10, y is greater than 15 and z is greater than 10 we will always receive zero as the output. Even though we are having redundant test cases, on concentrating on the inputs we have to design three test cases with these scenarios. This will check all the possible conditions to be evaluated for the system under test.

- **When x > 6, y<16 and z<11**

  When the value of x is greater than 6, the NVM_write function will check for the value of address and it will see that the address is not in the range to move forward to the Get_status function. Thus, an error will be produced which in turn will result into output 1.

- **When x<7, y<16 and z<11**

  When the value of x is less than seven, the system will move forward to Get_status block from the NWM_write and according to the value of y and z the further output will be declared.

- **When x< 7, y > 4 and z<11**

  When the value of y is greater than four, the system will not be able to move forward to the next block so it will provide an error message as output 2.

- **When x<7, y<5 and z<11**

  When the value of y is less than 5, the system will move forward to the Get_Data block and we will receive a positive output at the end as 3.

With the above conditions we will be able to make eighteen distinct combinations which can be redundant. The redundant test cases are not mentioned in the above explanation. The above list only depicts the conditions which are leading to the designing of the test cases. Numeric values can be added to the conditions as per the requirement and running the test cases will provide the required output.

5.1 DECISION, CONDITION AND MCDC COVERAGE

We have been using the terms Decision coverage, condition coverage and MCDC coverage while explaining about designing the test cases in previous chapters. Now to move forward with the results and explanations of the test scenarios designed, we must know what are these terms in detail.

5.1.1. DECISION COVERAGE

The Decision coverage of a system aims to ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable code is executed. In simple terms it means that all the decision points in a system is executed at least once in a test suite. If this is successfully done, we will obtain hundred percent

coverage on Decision parameter. The 100 % Decision coverage ensures the test engineer that he or she has designed the test scenarios in such a way that the entire system is covered.

5.1.2. CONDITION COVERAGE

The Condition coverage also known as Predicate Coverage in which each one of the Boolean expressions have been evaluated to both TRUE and FALSE. Thus, in a system especially in our scenarios, it always emerges like if a condition is true follow one branch and if a condition is false follow another branch. If a system is designed such that all the conditions have their own true and false branches and the test cases designed by the test engineer evaluates all these true and false conditions the final report will provide a hundred percentage condition coverage.

5.1.3. MODIFIED DECISION OR CONDITION COVERAGE

In Modified Decision or Condition Coverage or the MCDC Coverage, each condition should be evaluated at least once which affects the decision outcome independently. In our context attaining hundred percentage in MCDC is not mandatory.

The coverage analysis settings of MATLAB enable the test engineer to enable the required coverage settings. The coverage metrices of MATLAB are Decision, Condition, MCDC, Lookup Table, Signal Range, Signal Size, Simulink Design Verifier, Saturation on integer overflow, and Relational Boundary. We are only using the Decision, Condition and MCDC coverages.

5.2. TEST RESULTS AND REPORTS

For executing the test cases designed using the test scenarios of the test sequence editor feature of the harness navigate to Test Manager. Inside test manager create a new test suite with a recognizable name for the function. Enable the required coverage matrices for the test suite under Coverage Settings. Also choose the required format of the report to be generated and the location to be saved under the Test File Options.

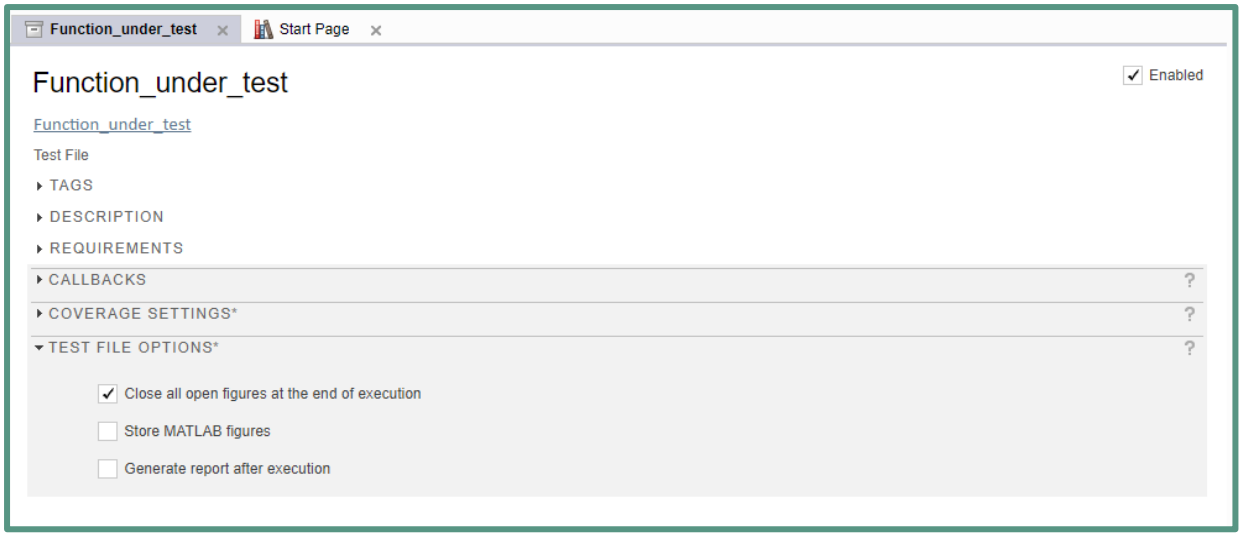


Figure 5.5: Test Suite Created Inside MATLAB Test Manager

Now a folder named input approaches is created inside the test suite and eighteen test cases are created under the folder. For each test case choose the system under test option and provide details of the model and the harness to be tested. Inside the Inputs option, Choose the test sequence block and override the default value with the appropriate scenario to be run within the corresponding test case. The images below show a test case inside a test suite.

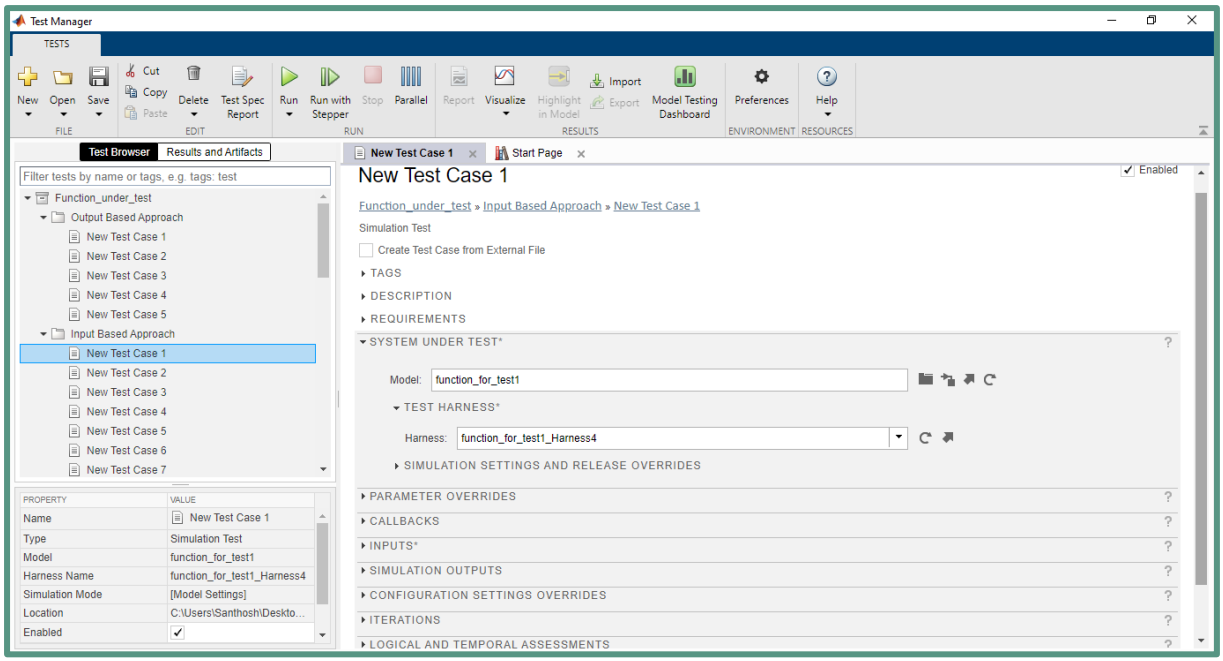


Figure 5.6: Test Case Created Inside a Test Suite in MATLAB Test Manager with settings to choose the system and harness under test.

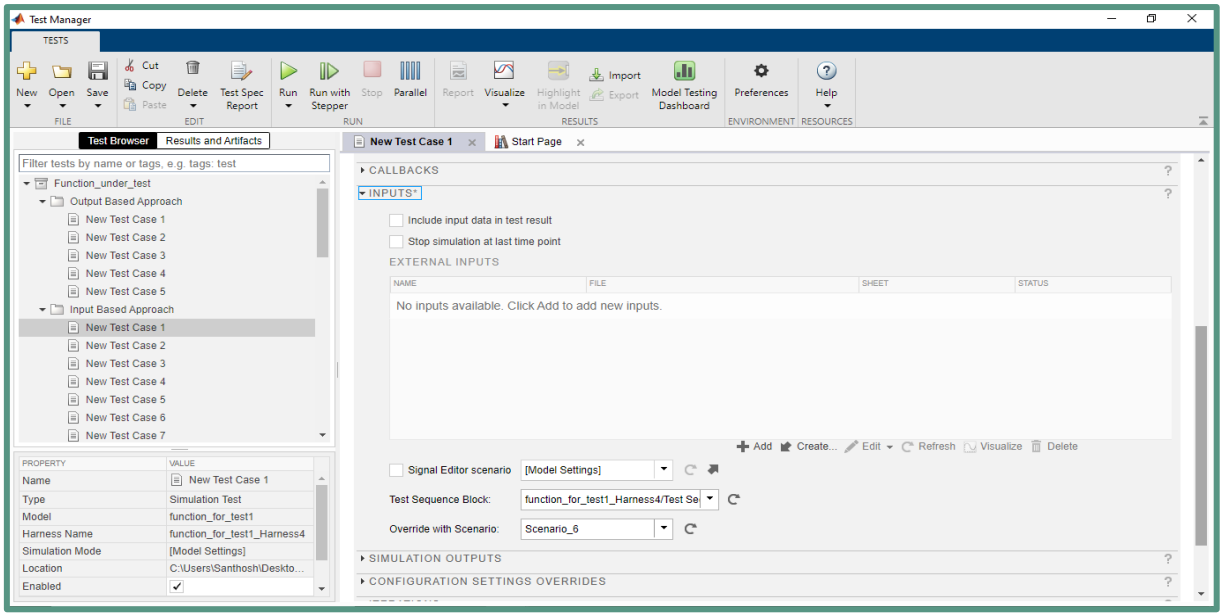


Figure 5.6: Test Case Created Inside a Test Suite in MATLAB Test Manager with settings to override the default input scenario.

When we run the folder, all the scenarios will be executed and the output results will be displayed in graphical format inside the generated report. Also, the coverage results and the percentage of coverage obtained for each coverage matrix can be viewed inside the report.

Now we have designed both input-based and output-based test cases for the system in Figure 5.1 and we have created a test suite named function under test for executing the scenarios. There are two folders created inside the test suite which are named as Output-Based Approach and Input-Based Approach. The output-based and input-based approaches as the name implicates contains the test cases which are designed as per our theoretical analysis. Now each folder is executed separately and the results and reports are verified.

The coverage matrics such as Decision, Condition and MCDC are evaluated the results can be observed inside the test manager panel as shown below.

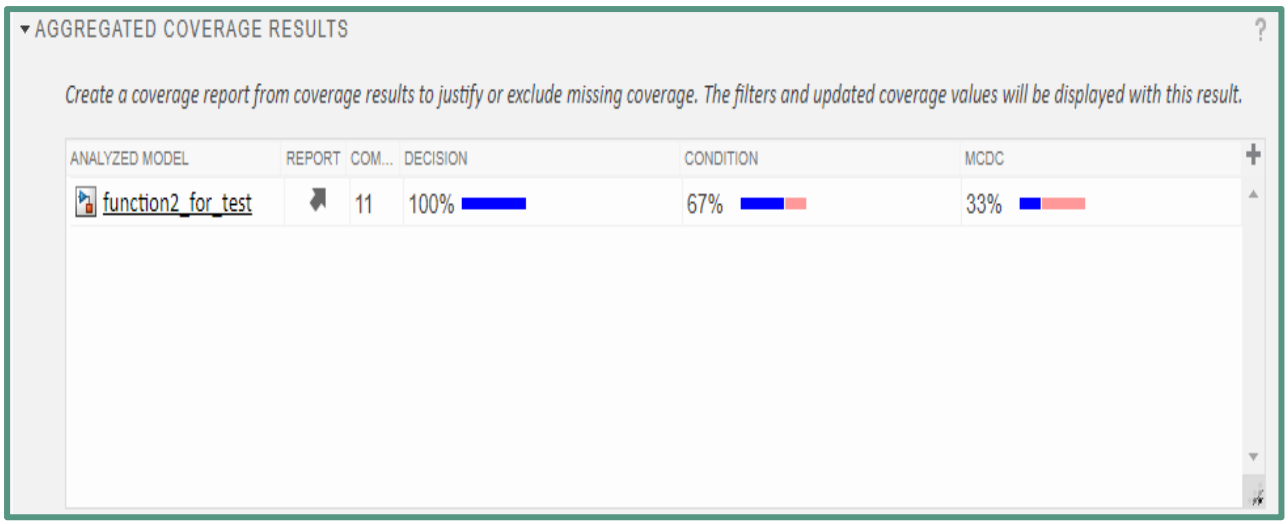


Figure 5.7: Coverage Results for Output Based Approach for the system under test from figure 5.1

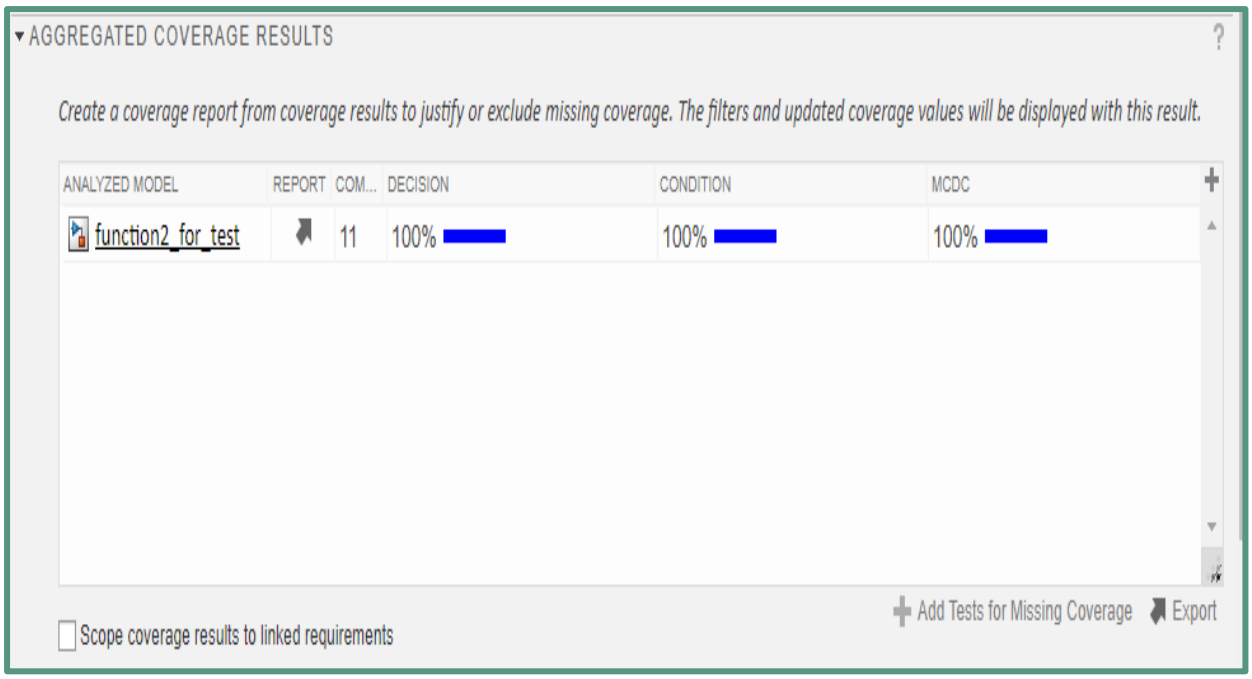


Figure 5.7: Coverage Results for Input Based Approach for the system under test from figure 5.1

From both the results we can conclude that we have obtained 100 percentage coverage for Decision parameter with four test cases in output approach and eighteen test cases in input-based approach. If the priority of the development team is to only obtain decision coverage, we can proceed with the output-based approach.

In the second image, Figure 5.7 hundred percentage coverage is obtained for decision, condition and MCDC parameters using the input-based approach. So, when the project requires all the three parameters to be hundred percentage we can proceed with the input-based approach. If we are only focusing on Decision parameter the output-based approach is best in this scenario.

The below diagram depicts the output of the eighteen test cases that are developed based on the input approach. We can see that only four output equivalent classes are acquired in the output end, but there are many redundant test cases which provides the same output level.

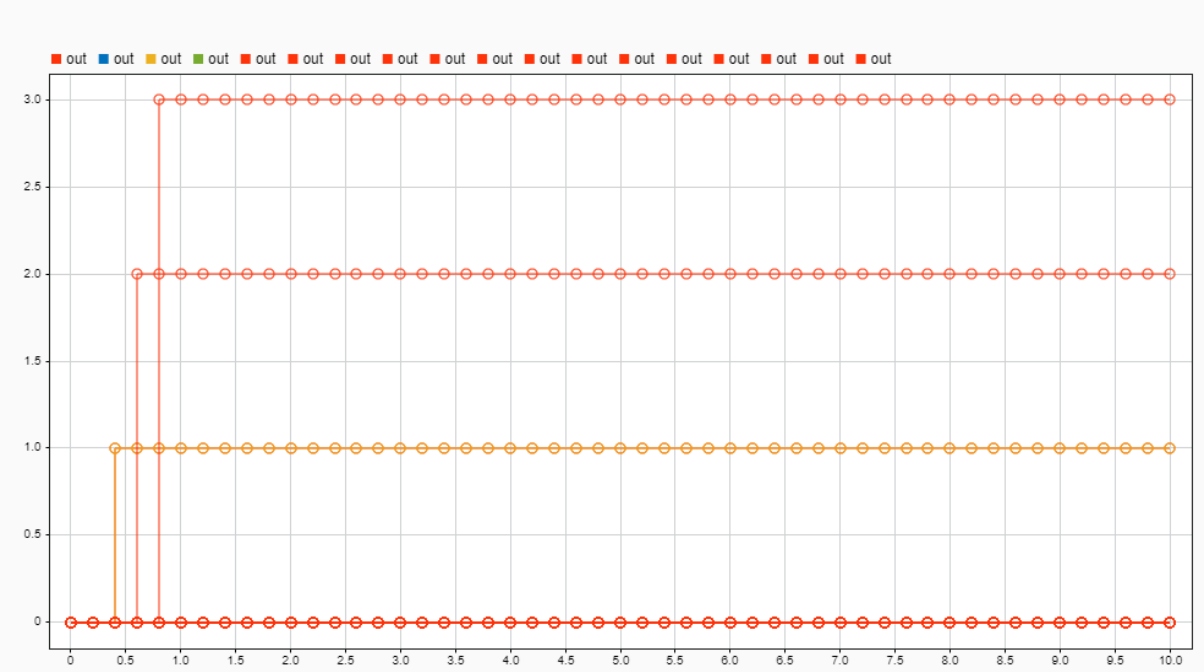


Figure 5.8: Output obtained by using input-based approach

As explained above, if we eliminate the redundancies in test cases, the hundred percentage coverage in Condition and MCDC will not be acquired.

Our main aim was to prove that the theoretical equations that we have developed and the way to find the best approach using these equations are valid in real life scenarios. The above example states that the theoretical equations and methods hold true for this system. But just with verifying one system we cannot claim that these equations will be true for every system under test. For verifying the integrity of the theoretical findings, we have chosen two more functions and performed similar operations.

Consider the system under test depicted in the following state flow diagram which reads two inputs and compares the input values with already available reference values.

Here x and y are the inputs of the system and z is the output of the system. On looking to the system, we can see that there are four equivalent classes for the output and we can obtain all the four outputs of the system by designing four test scenarios. Thus, on following the output- based approach four test cases have been designed for the system.

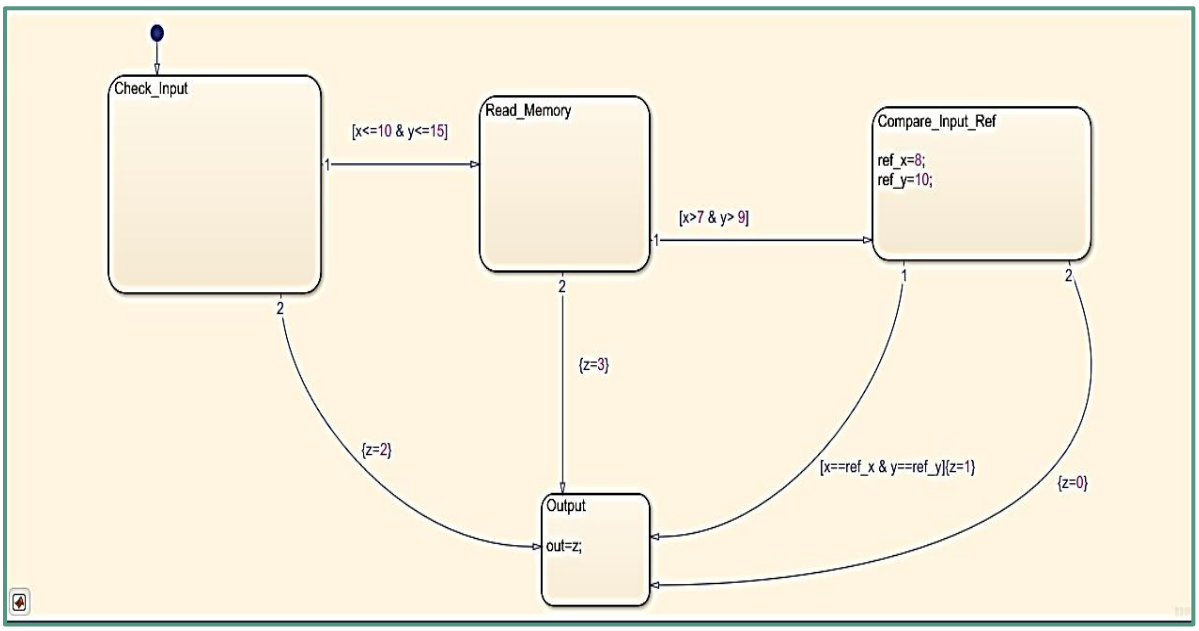


Figure 5.9: System Under Test 2- Function to compare inputs.

The system gives an output value of z = 2 when the value of x is not less than or equal to 10 and the value of y is greater than 15. Thus, the first test case for the system in output-based approach will be giving values for x greater than 10 and y greater than 15. The second test scenario will be the value of x less than seven and y less than 9. This scenario will provide the output value 3. When the value of x is equal to the reference value eight and value of y is equal to the reference value 10, the system will provide an output value of 1. When the value of x is greater than six and value of y is greater than 9 and at the same time the values of x and y are not equal to the reference values the output received will be zero.

Thus, the output-based approach describes these four test cases and on executing these four test cases with the same procedure as explained for the function in Figure 5.1, we receive hundred percentage decision coverage for the system.

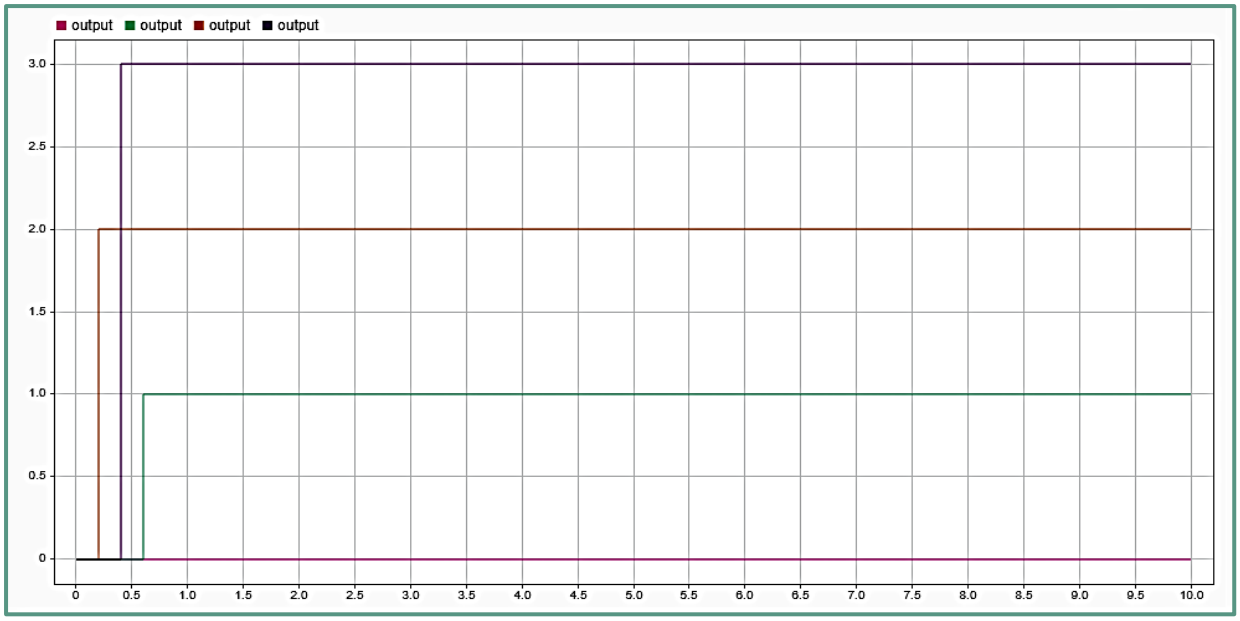


Figure 5.10: Results obtained using output-based approach

The four output equivalent classes are obtained using this approach, as we check the top scroll of the graphical representation, we can see that the four equivalent classes are obtained only using four test cases and those four results are represented using different colors. The below image represents the decision coverage, condition coverage and the MCDC coverage obtained for the system under test using the output-based approach.

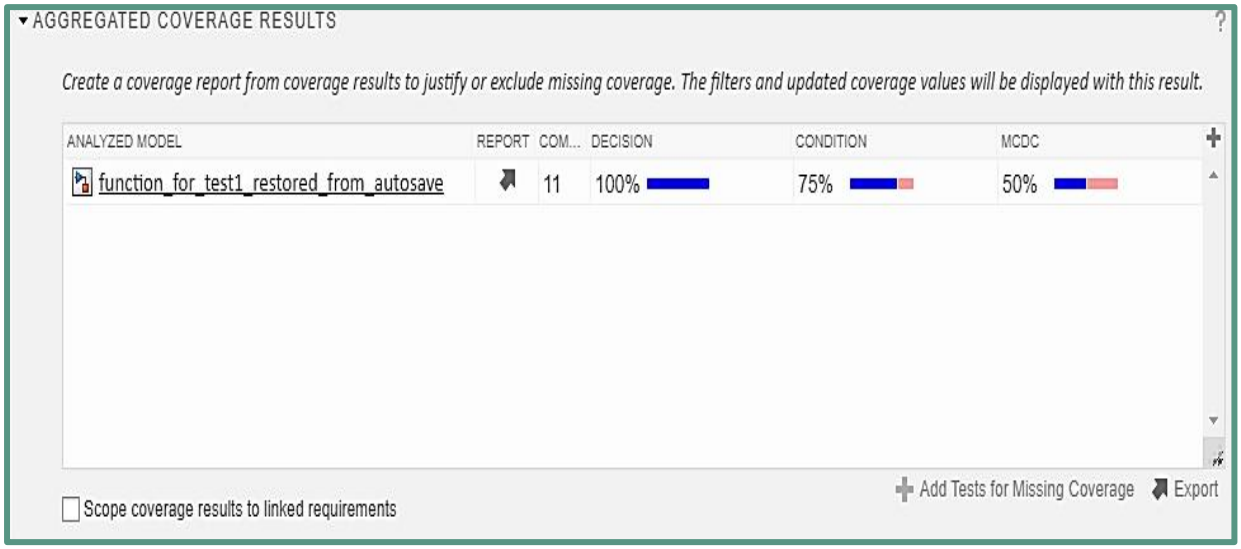


Figure *5.11: Coverage results for output-based approach*

Thus, using the output-based approach the test engineer could attain hundred percentage coverage for Decision parameter, 75 percent coverage for the condition and 50 percent for the MCDC.

Now, we need to focus on designing the test suite the input-based approach for comparing both the approaches and prove that our theoretical analysis holds true for every system under test. The system has two inputs x and y and the two inputs have four equivalent classes. After finding all the equivalent classes of the system and eliminating the redundancy of the test scenarios, we have found nine test cases which will result in hundred percentage coverage for the system in all the three matrices of coverage settings.

The test cases are designed as follows, when the value of x is greater than 10 and value of y is greater than 15 it exceeds the range of inputs. Therefore, the first test scenario will be of values that exceeds the range of the inputs. The second equivalent class of inputs that provide a different output is when the value of x and y are equal to the reference values. The third equivalent class of inputs is when the value of x is less than seven and the value of y is less than 9.

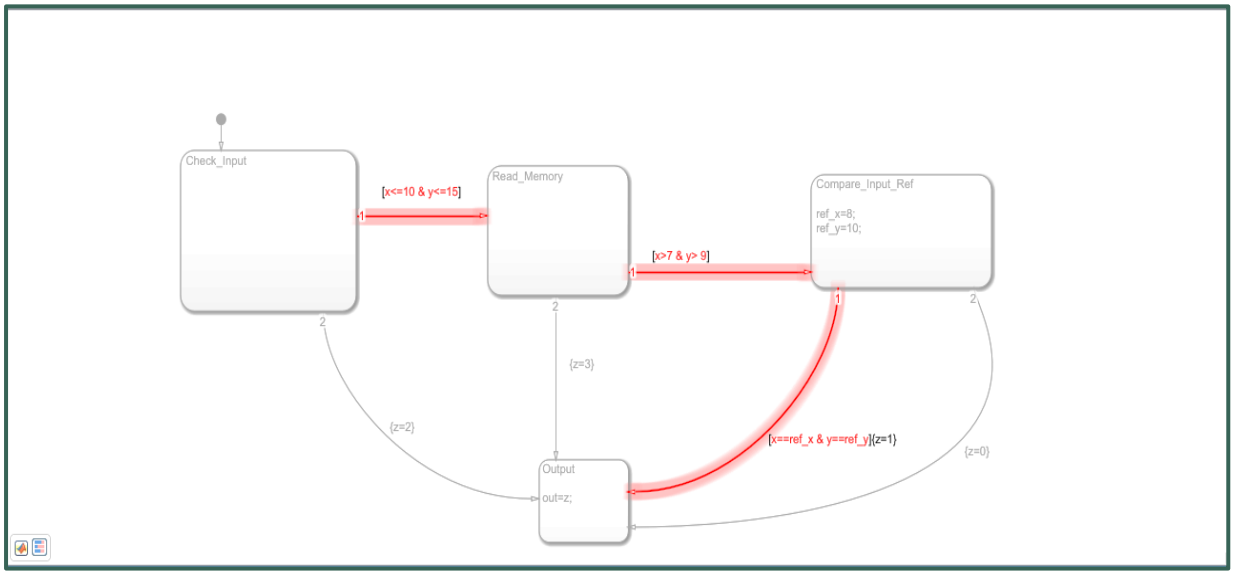


Figure 5.12: A branch being executed inside the state flow diagram

The fourth equivalent class of inputs is when the value of x is greater than 6 but not equal to the reference value and the value of y is greater than 8 but not equal to the reference value. On analyzing all the combinations, we will receive a total of nine test scenarios. Note that we have to choose the values of x and y such that the redundancy in test cases is eliminated.

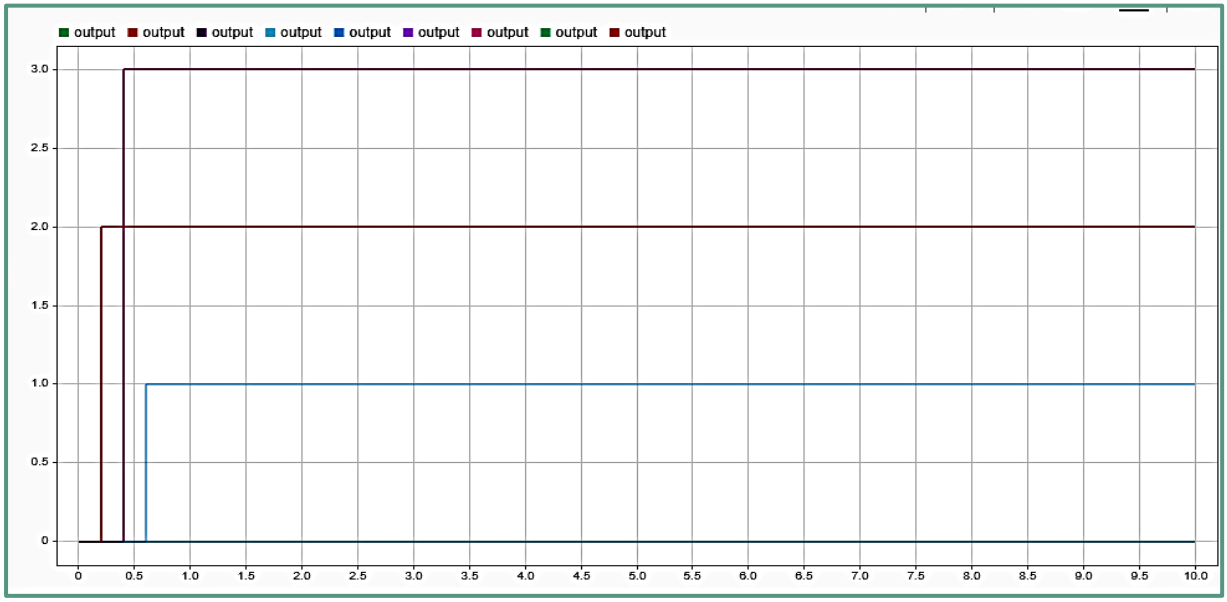


Figure 5.13: Outputs obtained with input-based approach

The above diagram shows the results obtained using the nine test scenarios in input-based approach. The outputs obtained are displayed in different colors and we can see the nine output results obtained have some redundancies even after eliminating all possible redundant test cases on concentrating on inputs and distinct inputs are applied to the system.

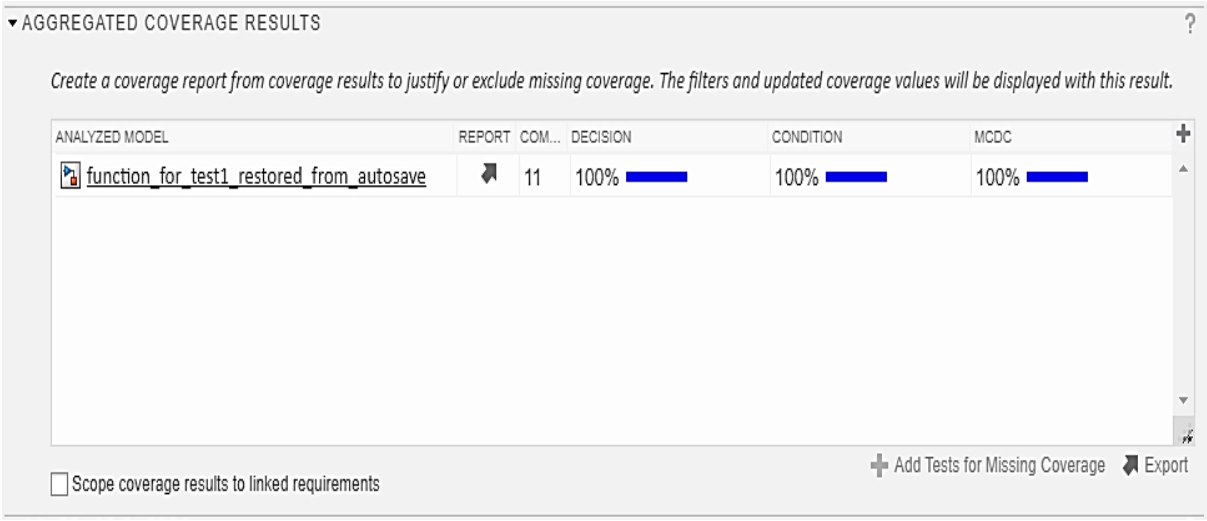


Figure 5.14: Coverage results obtained with input-based approach

Thus, on comparing the input-based and output-based approach we can see that our theoretical analysis holds true for both the systems. As mentioned in case of the first system (Figure 5.1) the test engineer can choose whether to follow the input-based approach or the output-based approach to design the test suite depending upon the complexity and importance of the project. If an intermediate release version is available and only regression is to be performed the test engineer can choose the output-based approach if the number of output equivalent classes are less than number of input equivalent classes.

Till now we have considered different systems with only one output. Now for checking the reliability of our theoretical analysis, we are designing a system with multiple outputs and output equivalent classes. A real function which provides multiple equivalent classes is chosen and the state flow diagram of the function is designed with use of MATLAB. For this, as explained in the previous chapter first the function written in C language is converted to a flow chart and then the flow chart is converted to the state flow diagram.

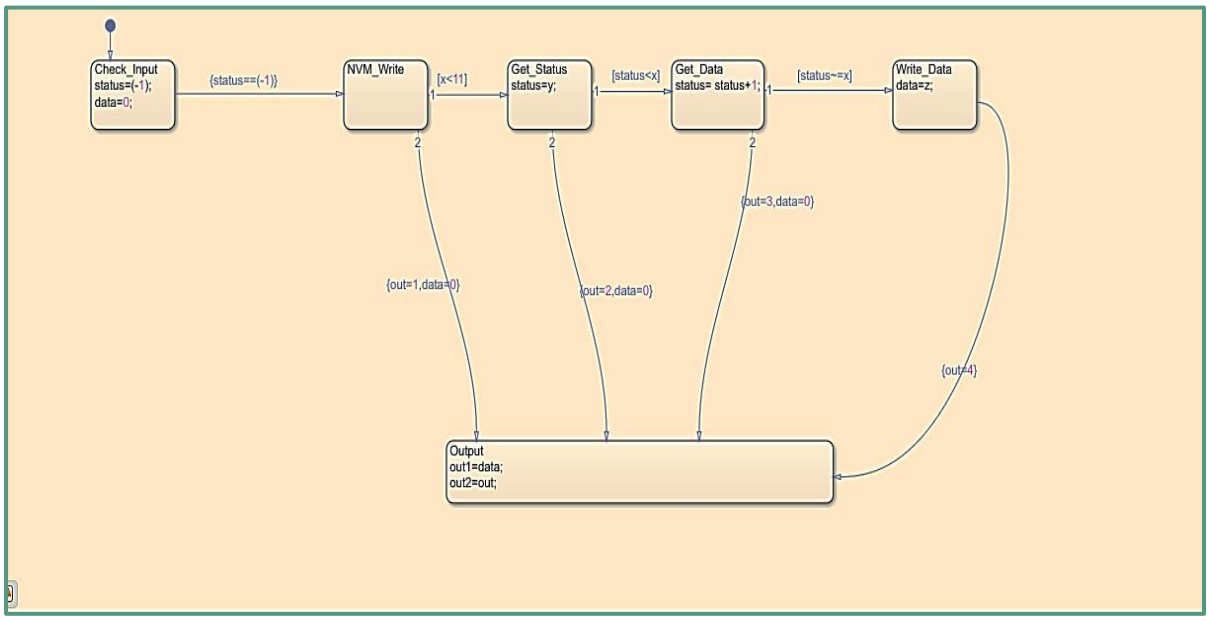


Figure 5.15: System with multiple outputs

The system writes the input data and the input data is displayed at the output through the data port. The second output port out shows the error messages at each stage of execution and at the final stage if the data is written successfully a success message is displayed. For each error and the success message in the function, we have used numerical values such as 1, 2, 3, and 4 here while designing it in MATLAB for the convenience.

Similar with the previous two systems, we are proceeding with designing test suite with both the input-based and output-based approaches. Here, the system has three inputs the address, data and the length of the data, x is the address, y is the length and z the data. As z is directly displayed at the output port if certain decisions are taken based on the address and length only x and y have effect on output and the equivalent classes are available only for x and y.

The data is set to zero initially and if there are error scenarios occurred the value of data will always be displayed as zero along with the error message. In case of the actual data given at the input port as zero, the output will be zero for data with a success message that is output 1 zero with output 2 four.

In the output-based approach, Output two has four equivalent classes, either the error messages or the success message and output one has two equivalent classes, either value of the input data or zero. Therefore, we need to design four test cases to obtain the four equivalent classes of output 1 and two test cases to obtain the two equivalent classes of output two. As the outputs are dependent, we can avoid the redundancies and in total we need to design four test cases.

When the value of x is greater than 10, output 1 gives the error message 1 and the output two will be zero. When the value of x is less than 11, the Get_Status block acquires the value of y and checks whether the value of y is less than x, if the value of y is greater than x then output 1 gives the error message 2, if not then the system execution proceeds to next block. At the Get_Data block the system checks if the value of y+1 is not equal to x, if it is equal to x then the output 1 receives the error message as 3. In both these cases the value of output 2 will be zero. If all these conditions are true then the final block is executed and the data is written and displayed at the output port with the success message. These are the four test cases that can be developed using the output approach.

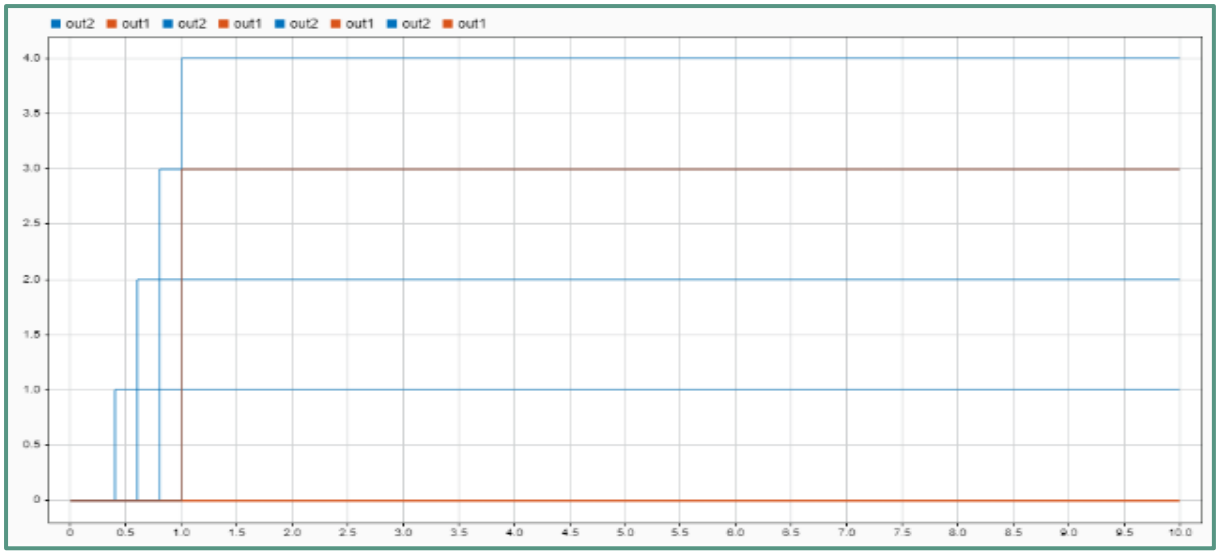


Figure *5.16: The results of outputs obtained with output-based approach*

The above diagram shows the results obtained using the output-based approach. Here also hundred percentage coverage is obtained for decision parameter. As the entire execution of the system depends on comparing the values of x and there are only decision points and no negative condition points available in the branches, even though we have designed the test suite to evaluate the condition and MCDC coverages, the condition and MCDC are not evaluated by MATLAB for this system.

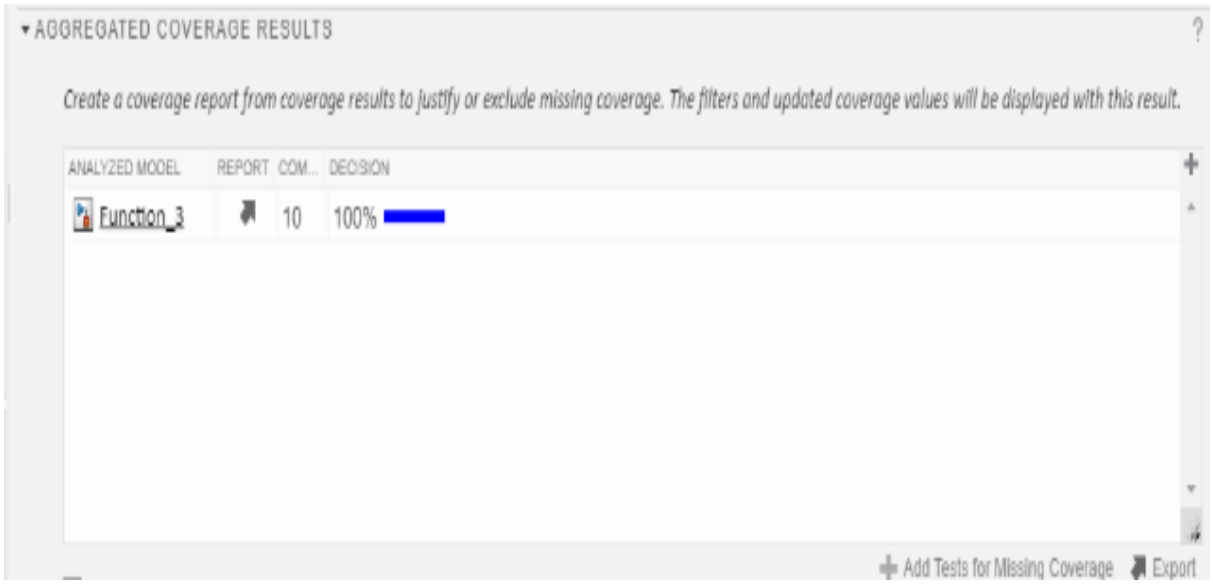


Figure 5.17: Coverage results for output-based approach

Now focusing on the input-based approach, we have two equivalent classes for input x that is x<11, x>10, x > y and x != y+1 . For input y, there is no limit mentioned and the further

execution of the system depends on the value of x every instance. Therefore, we can say that only x has equivalent classes and there are four equivalent classes for x. In this case concentrating on the inputs also, we have to design four test cases.

Figure 5.18: Results obtained using input-based approach

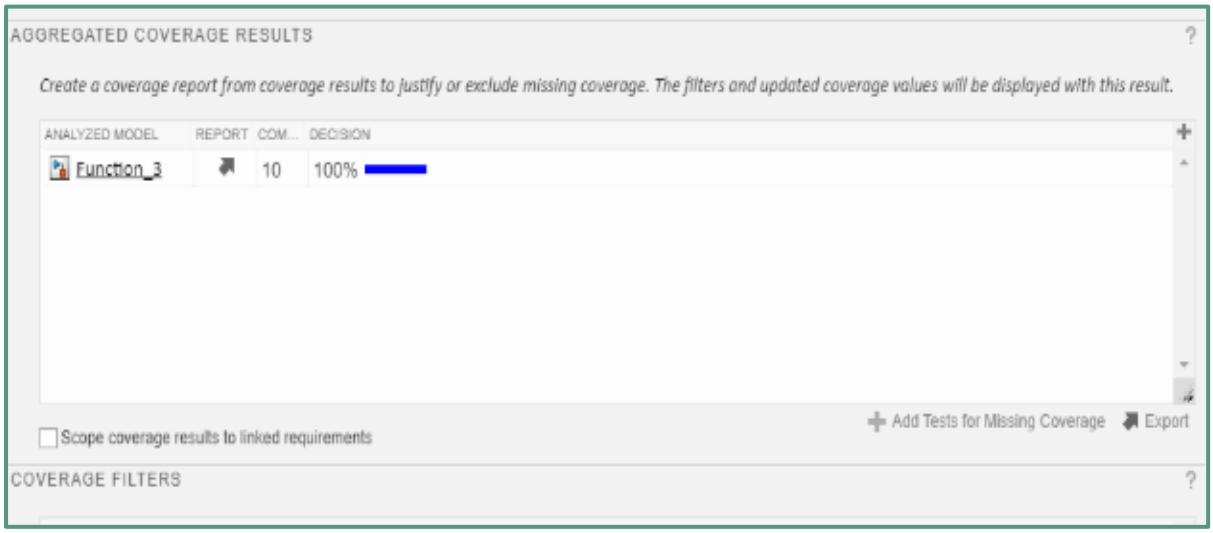


Figure 5.19: Coverage results obtained using input-based approach

Our theoretical equations hold true in both cases for the number of test cases in both the approaches. If we calculate the total number of test cases for the system under test using our equations for the output-based approach we will receive a total of six test cases, here for our system also we can obtain six test cases, but due to redundancy we have eliminated those two test cases while designing the test suite and those two test cases are not displayed in the final test results only because both the outputs are inter-dependent.

Thus, with the output-based approach we have 6 test cases for the system and with the input-based approach we have 4 test cases. We have also attained hundred percentage coverage for decision parameter in both the cases. Thus, it is the choice of the test engineer now to finalize which is the best approach to design the test suite on considering all the factors affecting the execution and the coverage matrices required for each project.

All the three systems that we have verified proves that the theoretical equations that we have obtained by our analysis holds true for each system under test and is reliable for the test engineer to consider these formulas while designing a test suite to find the best way to approach a system under test.

One of the main goals of the project has been achieved in this stage and as mentioned earlier there are no known previous evaluations or researches performed on this topic till date. In the upcoming chapter we will be approaching the Product Architecture of the automotive microcontroller on having all these concepts in mind and the test suite will be designed considering these formulas.

CHAPTER 6: TEST DESIGN FOR PRODUCT ARCHITECTURE

Chapter two explains the necessity of the early-stage product architecture verification of an automotive micro-controller. We have been using the term Product Architecture frequently in previous chapters, however this chapter explains what the product architecture of an automotive micro-controller is, the verification plan and the automation of test suite for the continuous evaluation.

6.1: PRODUCT ARCHITECTURE

The structuring (or chunking) of a product's functional components is known as product architecture. It is the interactions between these components, or chunks. It has a big impact on how to create, produce, market, use, and maintain a new product. linking system-level design to system engineering principles.

Figure 6.1: An automotive microcontroller manufactured by Infineon

There are two different types of product architecture development named modular and integral product architectures in the industry nowadays. A family of automotive microcontrollers developed inside Infineon uses the same product architecture unless there is a main functional change. In some scenarios, the product architecture or some features or blocks in the product architecture of one product in a product line is adapted partially or completely for another variants.

The product architecture development and evaluation are mainly carried in the Enterprise Architect software. Infineon uses UML for the development purposes of the automotive microcontrollers. The Enterprise Architect model of the product architecture compliments with the MATLAB Simulink state flow diagrams. Migration of the product architecture to MATLAB is the easier method as Enterprise Architect does not provide the features of developing a test suite and the automation of it inside the software.

Most of the automotive microcontrollers developed inside Infineon uses Infineon's coreless transformer technology and galvanic isolation. The pin configuration of Infineon's automotive IGBT is displayed below.

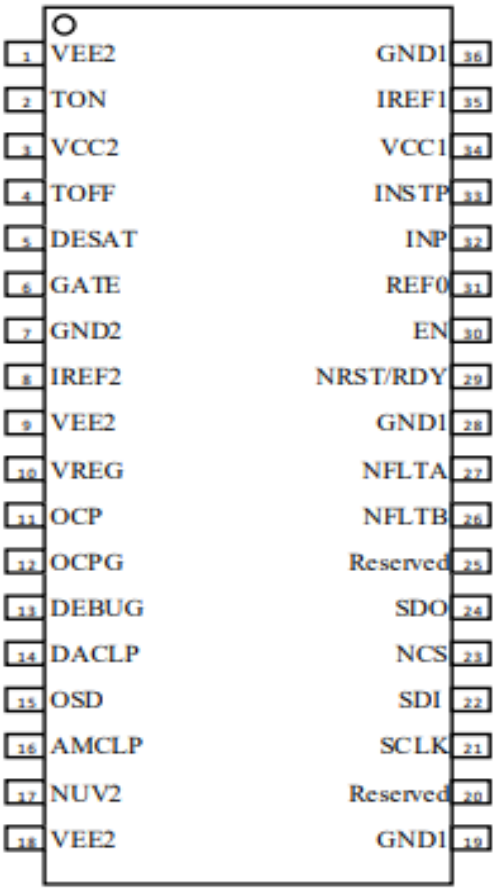


Figure 6.2: Pin configuration of a high voltage automotive IGBT driver

The product architecture of an automotive microcontroller is developed in MATLAB Simulink using the state flow diagram feature. The functionalities, the terminals and the input blocks of the product architecture are not explained in detail in this chapter so as to follow the confidentiality agreement with the organization. The general outline of the product architecture and the test suite designed is explained in detail.

The Simulink model below displays the product architecture model. This Product Architecture comprises of 16 input and 16 output ports.

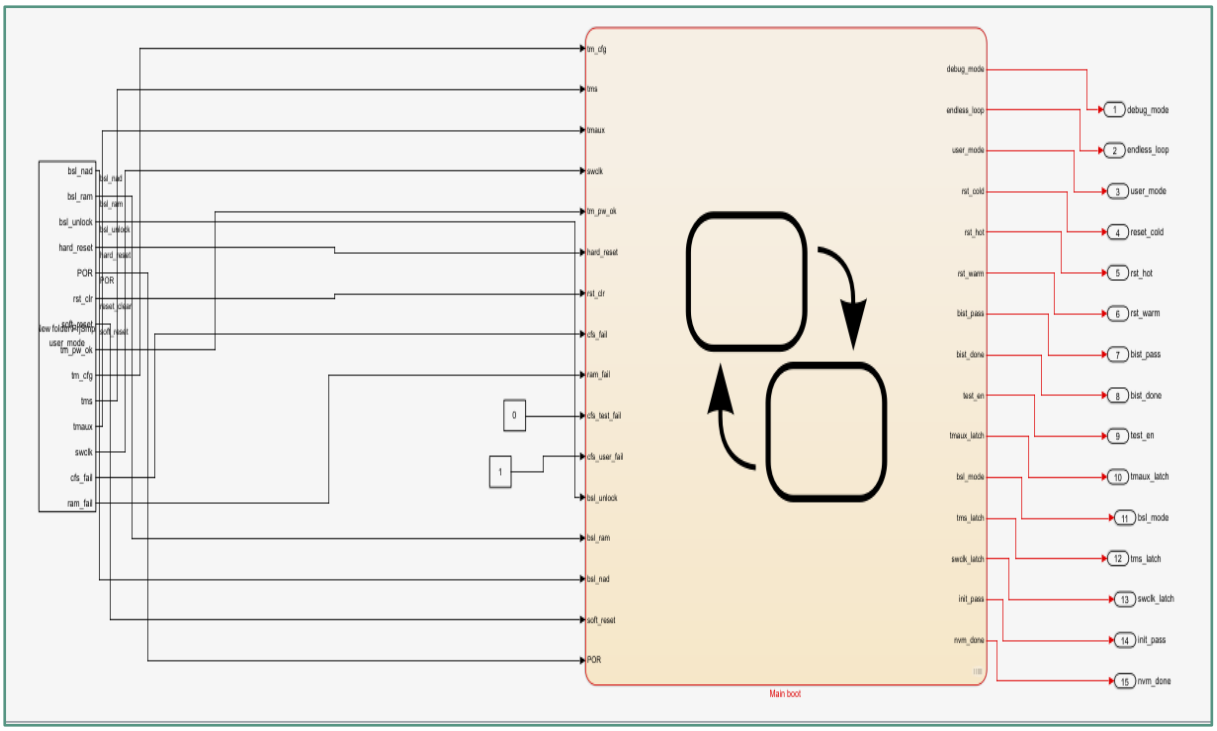


Figure 6.3: Product Architecture Model in Simulink

The system is designed in such a way that the inputs are fed to the system through a spreadsheet. The spreadsheet contains the input signals with different values for different time instances. The distinct input signals are fed to the system with a time gap of 0.1 second. We are validating the state diagrams and the time specified is only a parameter to feed different states into the system. The time range varies from 0.1 second to 10 seconds. After executing different values of inputs for the first ten seconds the system stops the execution and the output signals are obtained. Sixteen output signals along different Sim Output that is whether each block is executed or not in each time instance can be visualized using the data inspector feature of the Simulink.

6.2. TEST HARNESS CREATION FOR PRODUCT ARCHITECTURE

Before developing a test suite for the Product Architecture, we need to create a Simulink test harness for the system. A test harness is a model with inputs, outputs, and verification blocks set up for various testing scenarios that isolates the component being tested. A test harness can be made for a single model element or for the entire model. We can test a model or a model component in different environments using the test harness.

Here, the product architecture operates in four different modes namely the User mode, Debug mode, BSL mode, and BSL far mode. The BSL mode and BSL far modes are the

bootloader modes of the automotive microcontroller. BSL stands for the Boot Strap Loader in automotive industry inside the firmware ROM.

Only when the Boot ROM is running on the device can a BSL connection be made. The device must be reset, either by a power-up-reset (POR) or by a PIN reset, in order for the device to execute the BSL in the Boot ROM.

The BSL connection acceptance window opens after the reset release and certain basic initializations of the device have been completed. This time window's duration is determined by the None-Activity-Counter (NAC). Once the BSL connection window has been opened, a BSL connection attempt must be made, and the attempt must be completed before the BSL connection window shuts with the expiration of the NAC.

Thus, to operate the system in different modes, we need to create different test cases that overwrites the inputs fed into the system through the spreadsheet. For this purpose, we need to develop a test harness which can called inside the test suite.

The test harness created for the product architecture using the MATLAB's test harness feature is displayed below.

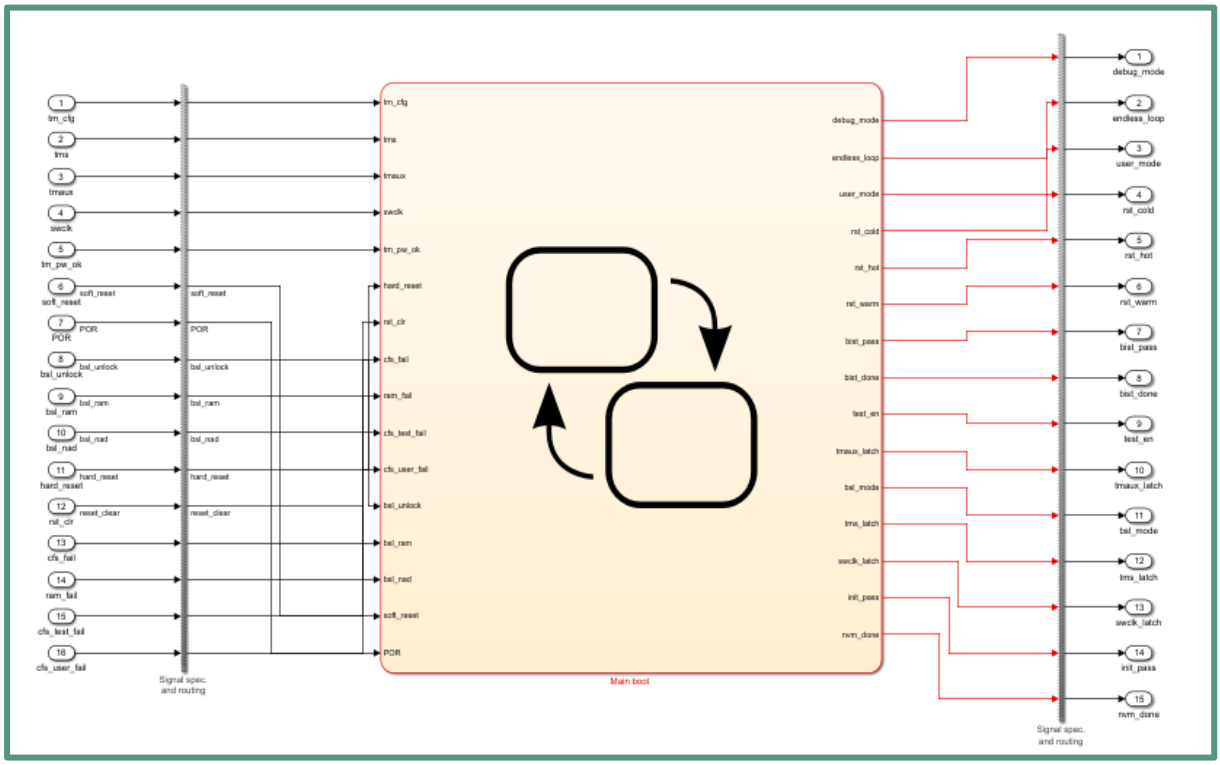


Figure 6.4: Test Harness generated for Product Architecture

As explained in the fifth chapter, here also we are using the test sequence feature of the MATLAB for feeding the inputs into the system. The generated test harness is tested for the reliability and the coverage analysis of the test harness is also performed for verifying whether the harness is functioning properly.

The outputs signals generated through the execution of harness is tested and compared with the output signals of the Product Architecture execution that we have performed for the model before creating the test harness. The output signals obtained by executing the harness is displayed below. There are different output signals as well as a state flow diagram of the product architecture is also displayed below.

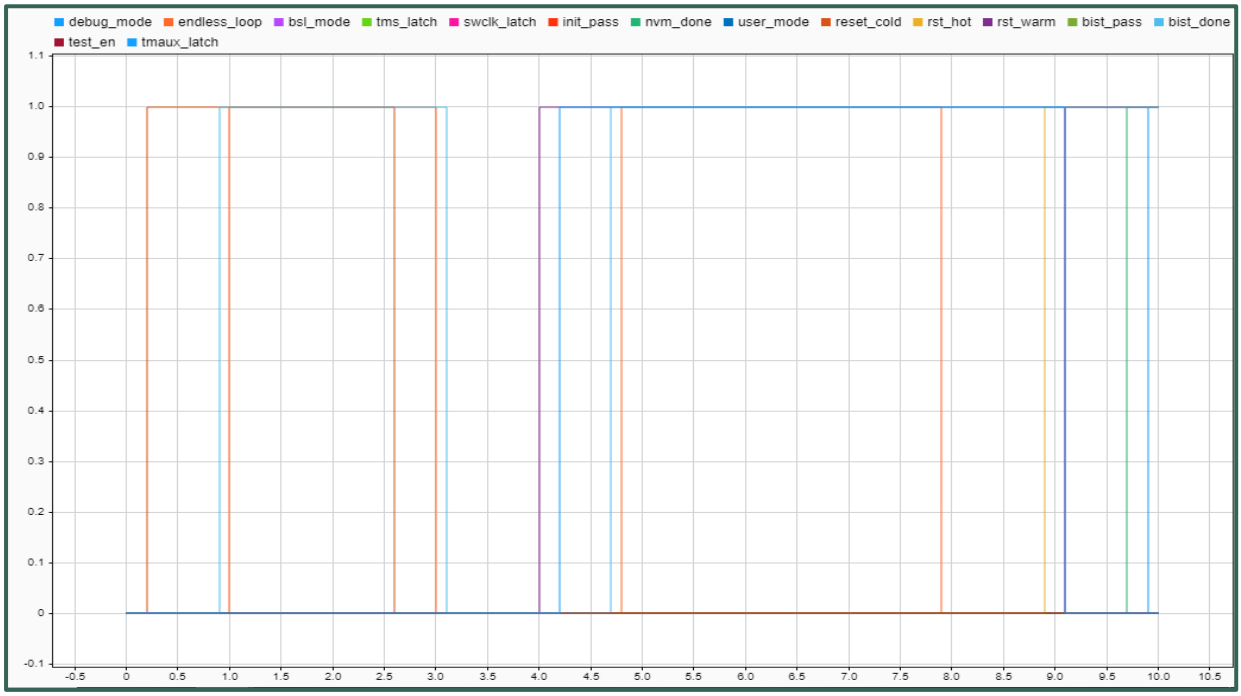


Figure 6.5: Outputs obtained on executing the Product Architecture

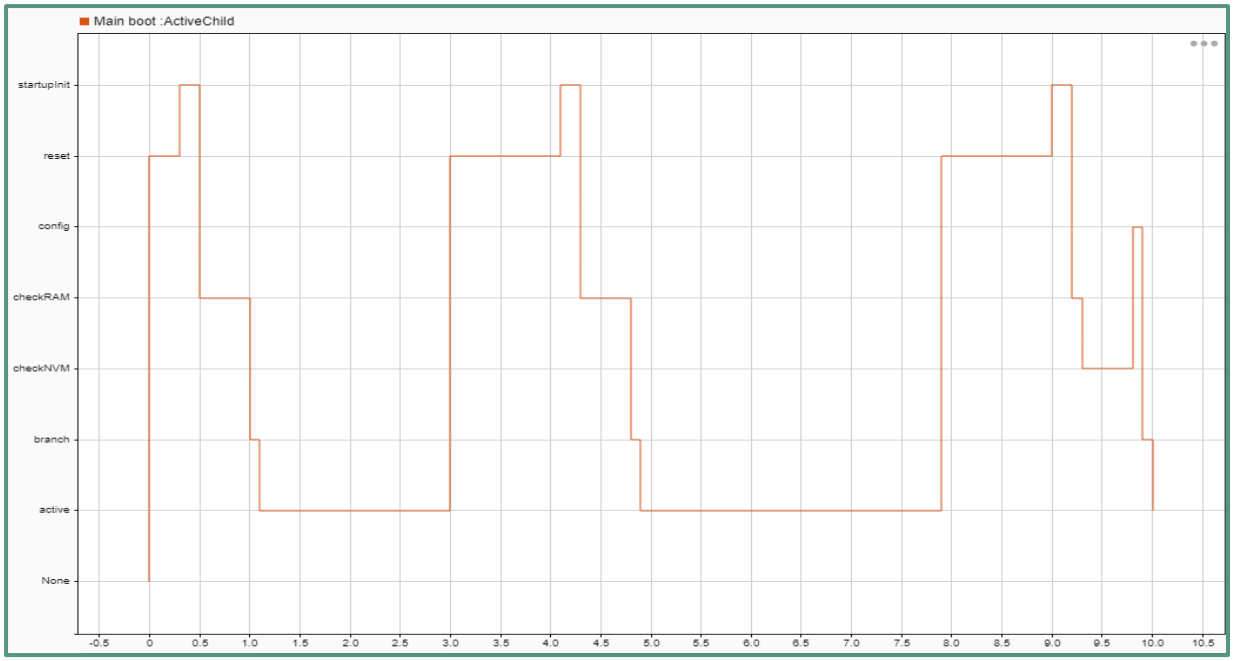


Figure 6.6: State flow diagram of the product architecture

The state flow diagram explains which all functional blocks of the product architecture are evaluated in each second of time. On checking the state flow diagram, we will be able to trace the path of execution. However, we need to keep these values as a reference for executing the test harness to check whether there is any error occurred during the creation of harness or is there any change in output occurred on executing the harness. The product architecture is executed in user mode throughout the procedure. The initial mode is set to user mode, there is no preference to set the modes of operation.

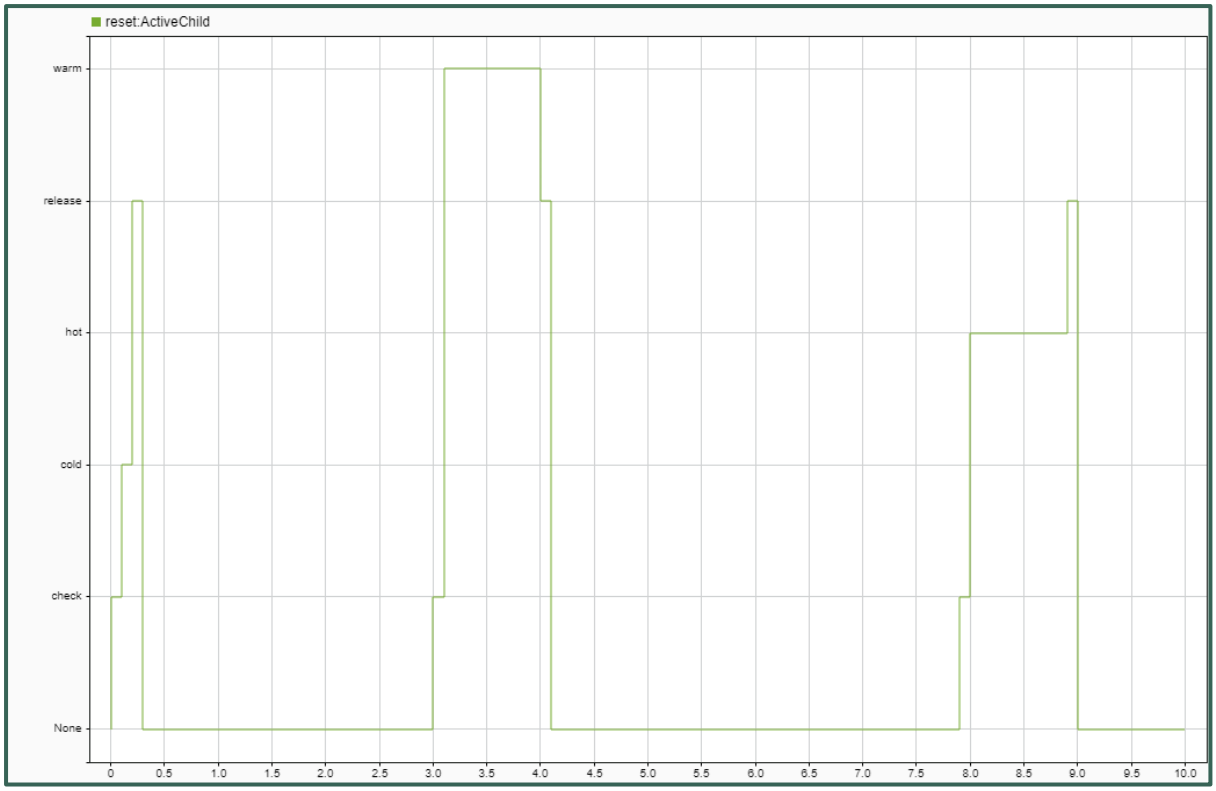


Figure 6.7: State flow diagram of reset block of the Product Architecture

The above diagram depicts the hard and soft resets happening inside the system when the inputs are applied in user mode. Hot, warm and cold resets are fluctuating in accordance with the input values applied. With respect to the type of reset occurred the further blocks are executed in the product architecture.

The coverage of the system is analyzed for the test harness in user mode. Hundred percentage execution coverage with 53 percentage of Decision and 52 percentage of condition coverage for the system is obtained in user mode alone.

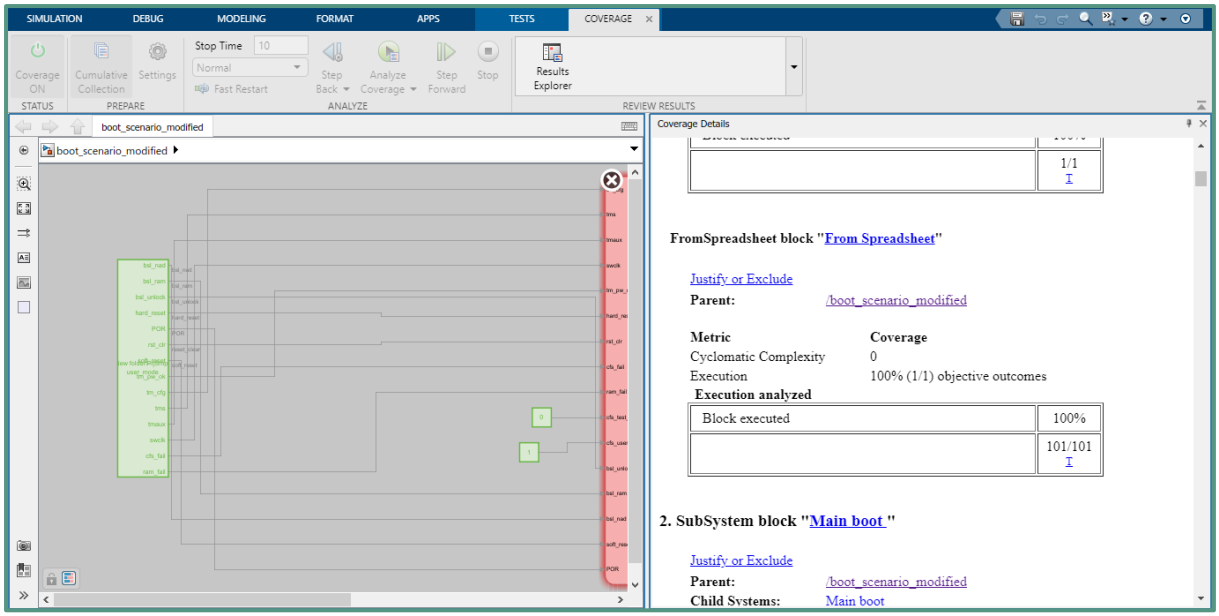


Figure 6.8: Coverage analysis of the Product Architecture in user mode

6.3. TEST SUITE CREATION AND TEST CASE DEVELOPMENT

We have already discussed about the different approaches and the reliability of the theoretical approaches to find the best way to verify a system under test in chapter four. Here, our system is having 16 inputs and 16 outputs. Thus, the number of inputs is equal to the number of outputs, so our next comparison is the equivalent classes of inputs and outputs.

We have already mentioned about the different modes of operation of the system. There are four different modes in which the system will be operated and these modes give different set of outputs. Therefore, we are classifying the system based on these modes of operation and the concentration is given to the four modes of operation to develop the test cases.

A test suite is created inside the test manager, recall the procedure of creating a test suite and test cases inside the test suite explained in chapter four. Unlike for the functions we have evaluated in chapter four, we are using a spreadsheet to feed the inputs into the system. The reason to use a spreadsheet is that as there are 16 inputs and 65536 possible combinations are available with the inputs, it is not convenient to manually add and edit the inputs inside the sequence editor. Therefore, we are not using the test sequence editor for the evaluation. Inside the test suite, create four test cases corresponding to each mode. Create a spread sheet with four sheets having the name user mode, debug mode, bsl mode and bsl far mode.

Now, we have to add values to the inputs in this spread sheet. We have filled the spreadsheet for first ten seconds with a time interval of 0.1 seconds. Choose the first test case inside the test suite and inside the system under test choose the product architecture model and the corresponding test harness.

Inside the Inputs option choose the “include input data in test result” option. Unlike the functions in chapter four, we are not selecting the test sequence block or override with scenario options here. The inputs block has a table for adding external inputs where we can add an external input file or create a file with signal generator.

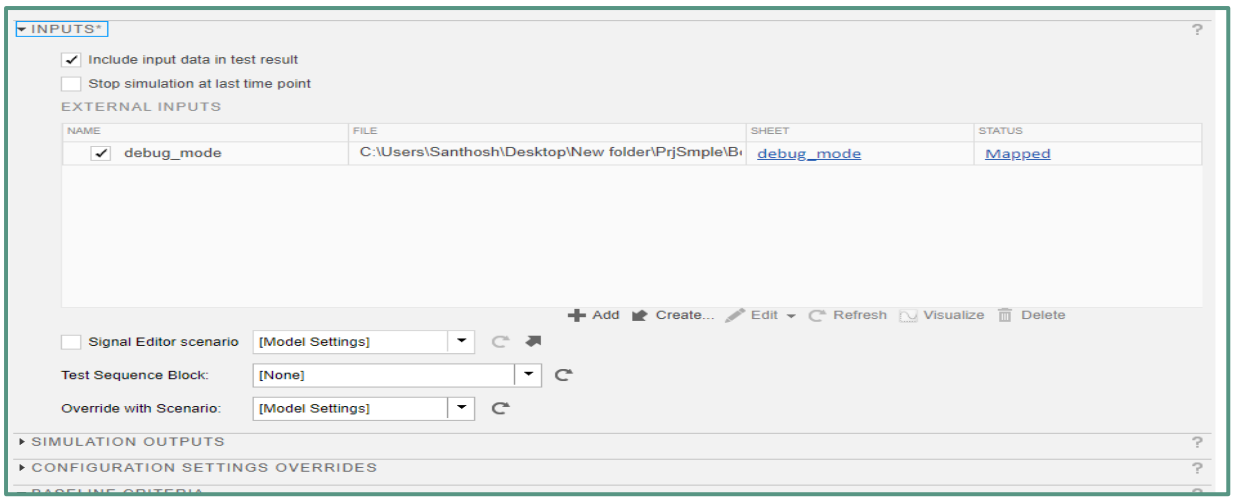


Figure 6.9: Input block inside the test case

As we have already created a spreadsheet for the inputs, we are choosing the add option to load the file into the test case. After loading the file into the test case select the Map inputs option to map the inputs inside the spread sheet to the system.

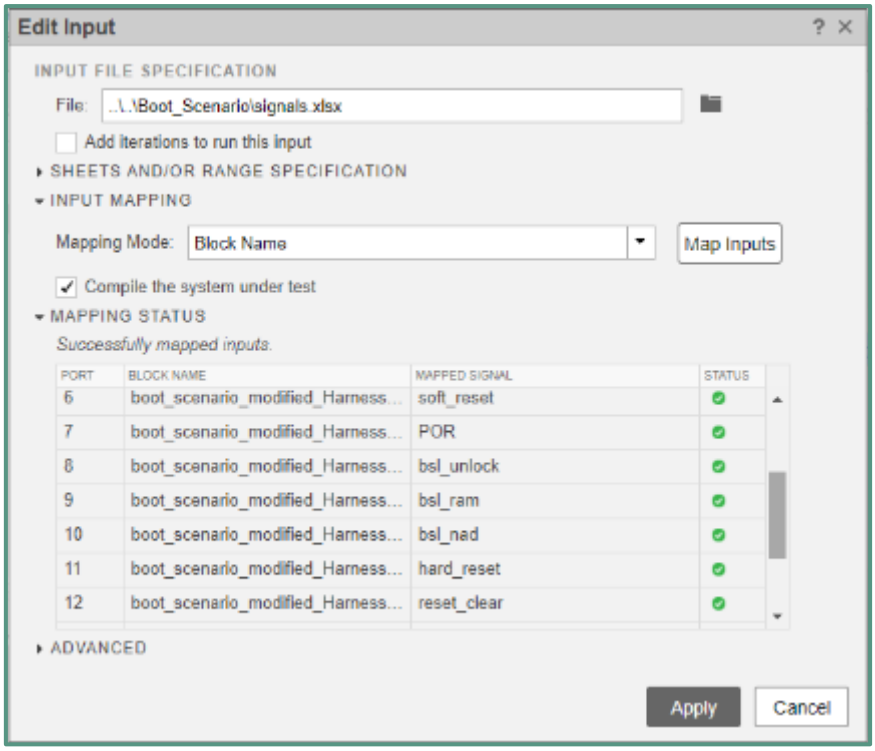


Figure 6.10: Mapping of inputs inside the input block after loading inputs from spreadsheet

As we have four different modes of operation, we are creating four different test cases each with 102 distinct input combinations for each mode. All the four test cases have been created inside the test suite and on executing the test cases we are receiving the following outputs and coverage results. Note that using the same set of inputs gives the same outputs always.

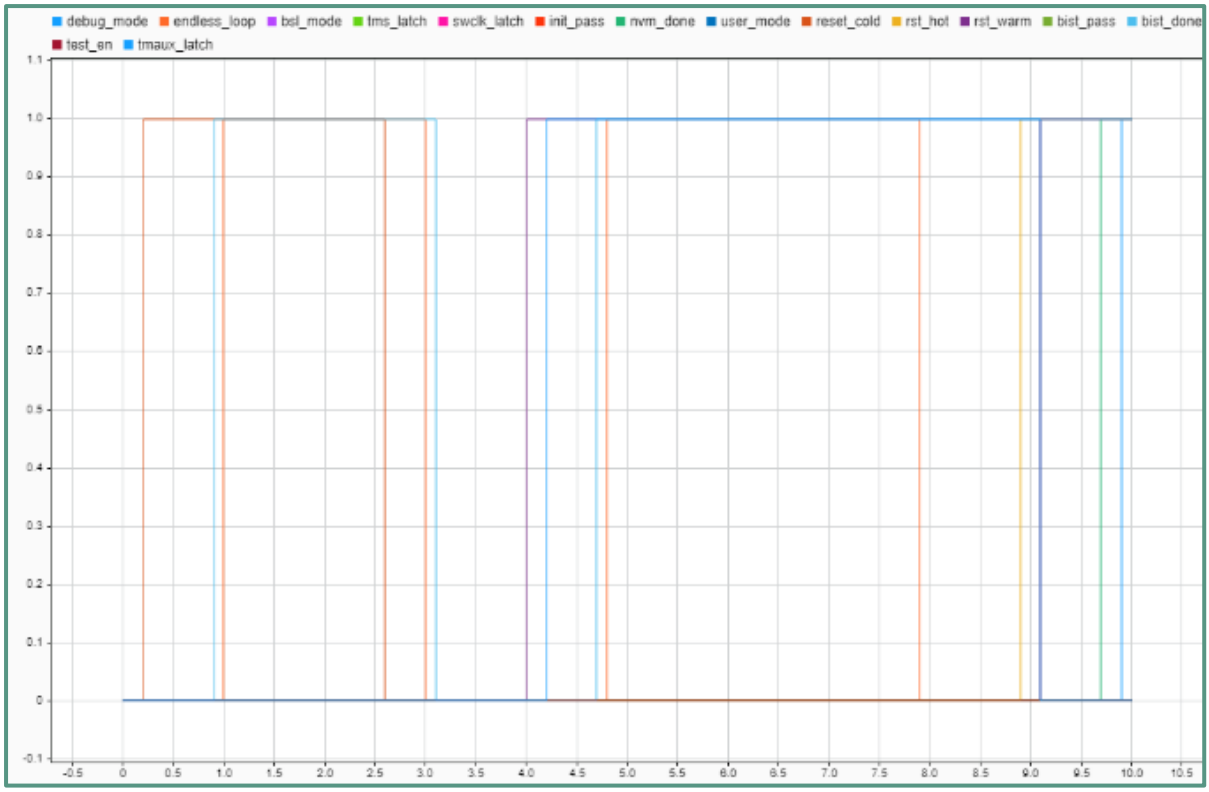


Figure 6.11: Output signals for Product Architecture in user mode.

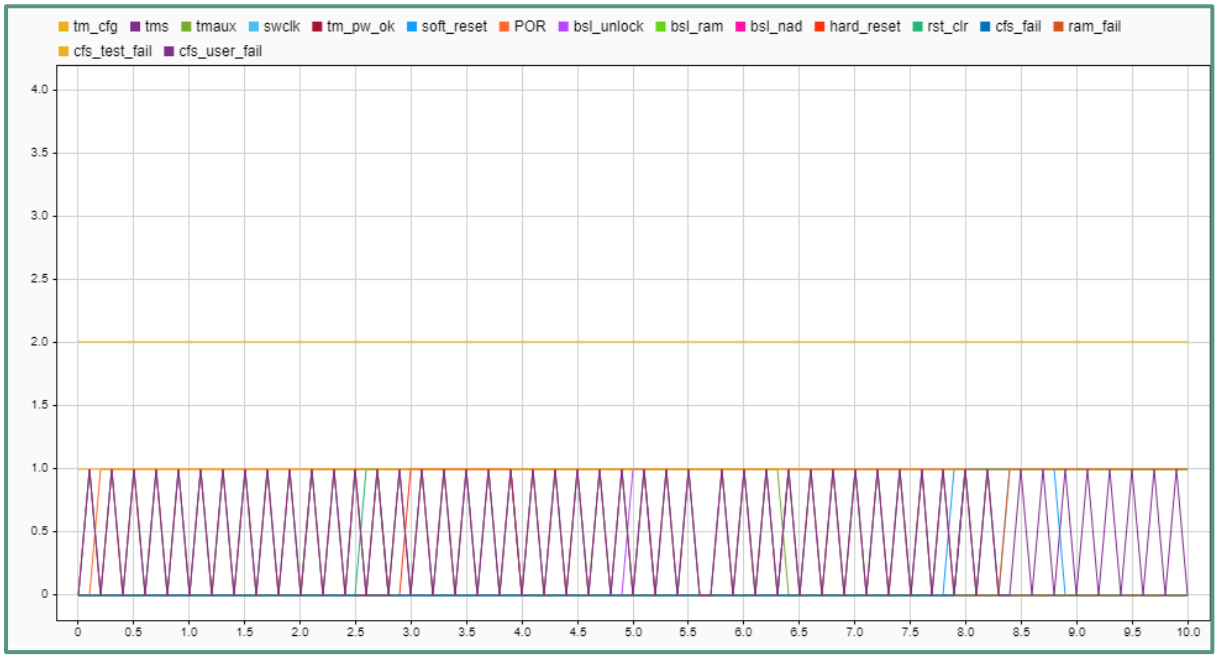


Figure 6.12: Input signals loaded to the system in user mode

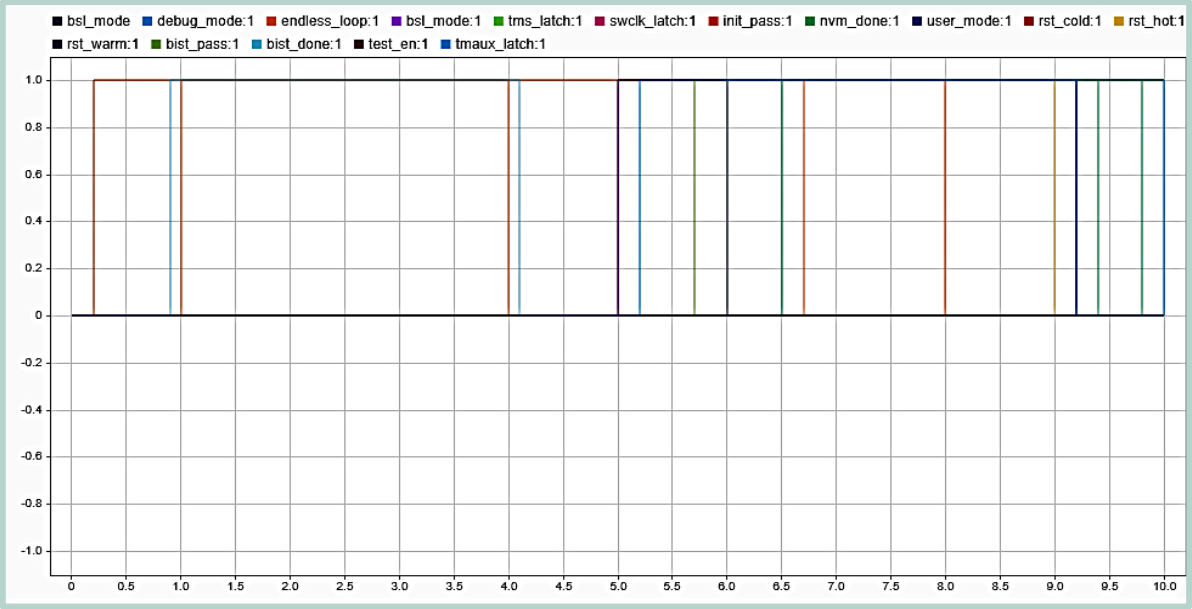
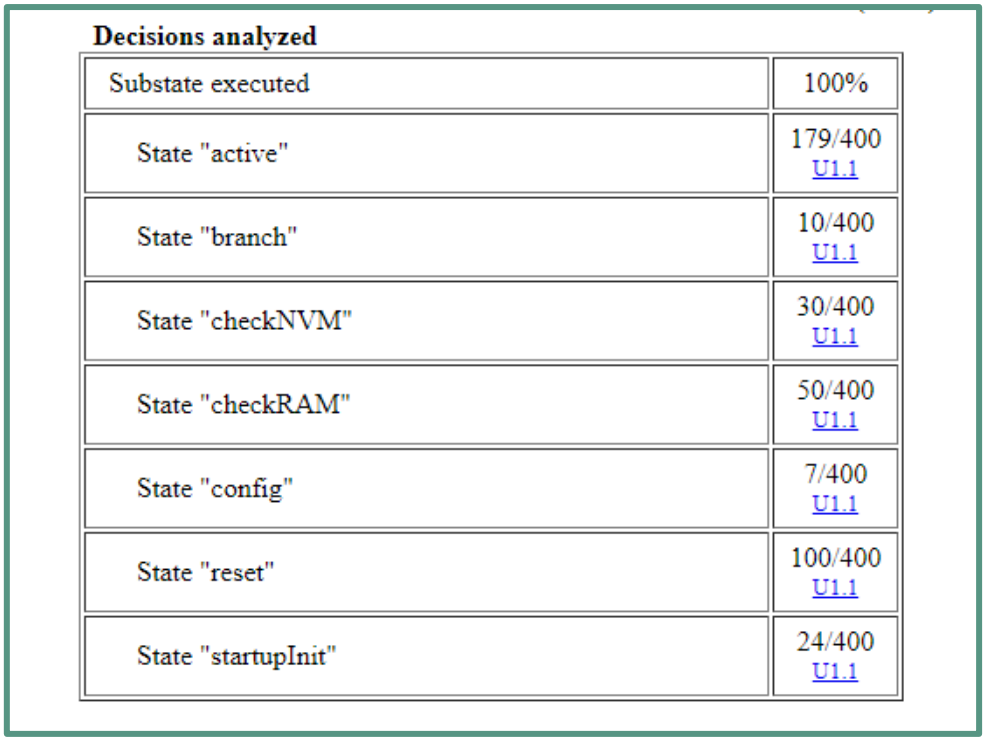


Figure 6.13: Output signals received on operating the system under debug mode

Now, we have created four test cases with four different modes of operation. If we observe the product architecture carefully, we have given the cfs_fail and cfs_test fail inputs as constants and they are not fed through the spreadsheet. This is done to achieve the maximum coverage for the system on testing the different modes. Our main aim is to achieve maximum coverage with minimum test cases. So, we have created two test cases which will feed all possible combinations of cfs_fail and cfs_test fail signals. These inputs are fed by creating the signals using the signal editor.

Thus, our test suite comprises of six test cases now. With these test cases we are analyzing the three different coverage matrices, namely condition, decision and MCDC. We have already found out that the individual operation in each mode gives more than fifty percentage of coverage for the system.

Now we need to generate the report with coverage results. Inside the test manager there is an option to generate reports for the test suite. Configure the test suite such that a report is generated in PDF format after executing the test suite and is saved in a location in the local system as per the preference. Note that we are using this option initially to generate and format the report with required contents. The automation of the test suite in the coming section explains the generation and sending the report to the test engineer from a remote system.



Decisions analyzed

Substate executed	100%
State "active"	179/400 U1.1
State "branch"	10/400 U1.1
State "checkNVM"	30/400 U1.1
State "checkRAM"	50/400 U1.1
State "config"	7/400 U1.1
State "reset"	100/400 U1.1
State "startupInit"	24/400 U1.1

Figure 6.14: Sub-states executed with the test suite

The above table displays each sub-state block evaluated during the execution with the test suite and we can see that each block is executed and every state is executed inside the block. There are 400 test evaluations are done in total and the number of times each state is executed out of 400 is marked in the table corresponding to each state.

Metric	Coverage
Cyclomatic Complexity	3
Condition	83% (5/6) condition outcomes
Decision	100% (2/2) decision outcomes
MCDC	67% (2/3) conditions reversed the outcome

Figure *6.15: Coverage result of the execution of test suite*

The Figure 6.15 displays the decision, condition and MCDC coverage obtained after evaluating the test suite. The coverage report generated by executing the test suite gives a detailed explanation of which all blocks are executed, what all conditions and decisions are executed and which all conditions are not executed.

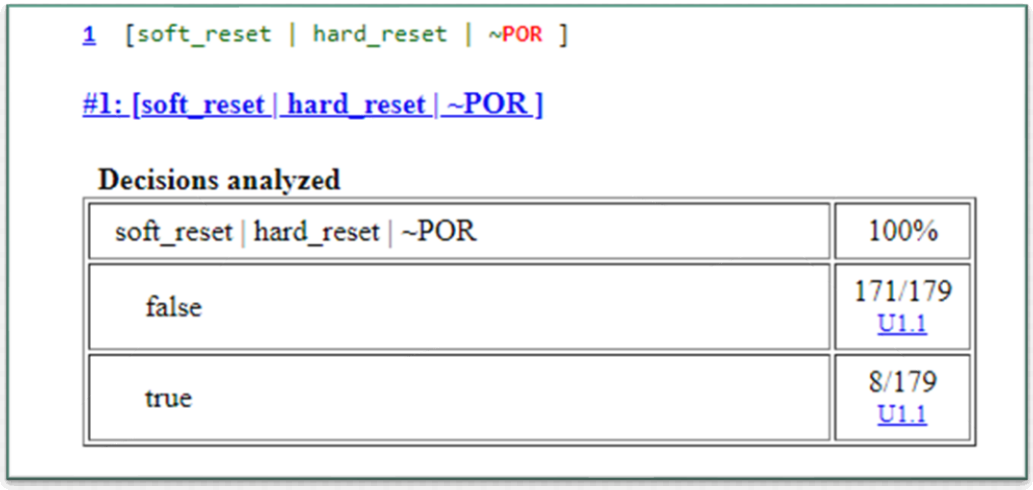


Figure 6.16: Snapshot of a coverage report of a decision analyzed

At the end of the coverage report generated, we are able to track which all branches are executed with full coverage of three coverage matrics and which all branches are executed only with hundred percent coverage for decision matrics. The Figure below shows the branches with the corresponding coverage results.

Figure *6.17: Branches of Product Architecture with Coverage Results*

The transitions of input signals from each block to the other for the set of inputs. Each transition will be evaluated for the Decision, condition and MCDC coverage. From figure 6.17, we can understand that there are transitions that has been completed with hundred percentage or full coverage for Decision, Condition and MCDC. There are transitions that only has hundred percent coverage for Decision parameter. Thus, in the final coverage report shown in Figure 6.15 only Decision coverage is hundred percentage.

6.4. AUTOMATION OF TEST SUITE FOR CONTINUOUS EVALUATION

As mentioned in the first section of this chapter, the Product Architecture can be the same for a product line. In all cases, the product architecture may not undergo many changes for a product under development. Each regression should be tested and it is mandatory with ASPICE standard that we must verify the product architecture. Thus, in continuous evaluation it is always convenient to automate the test suite.

Infineon uses Jenkins and Bitbucket inside the organization for the automation of the verification of the products. Therefore, we are also integrating the MATLAB with Jenkins and Bitbucket. Our first step towards the automation is installing the MATLAB in a remote node. A remote node is already available which is running inside the department of automotive Body P0.1ower for the verification purpose.

A folder is created inside Jenkins for in the name of Product Architecture verification. The MATLAB plugin is installed in Jenkins. A repository is created inside the Bitbucket and the MATLAB project created is pushed to the Bitbucket using the git extension software.

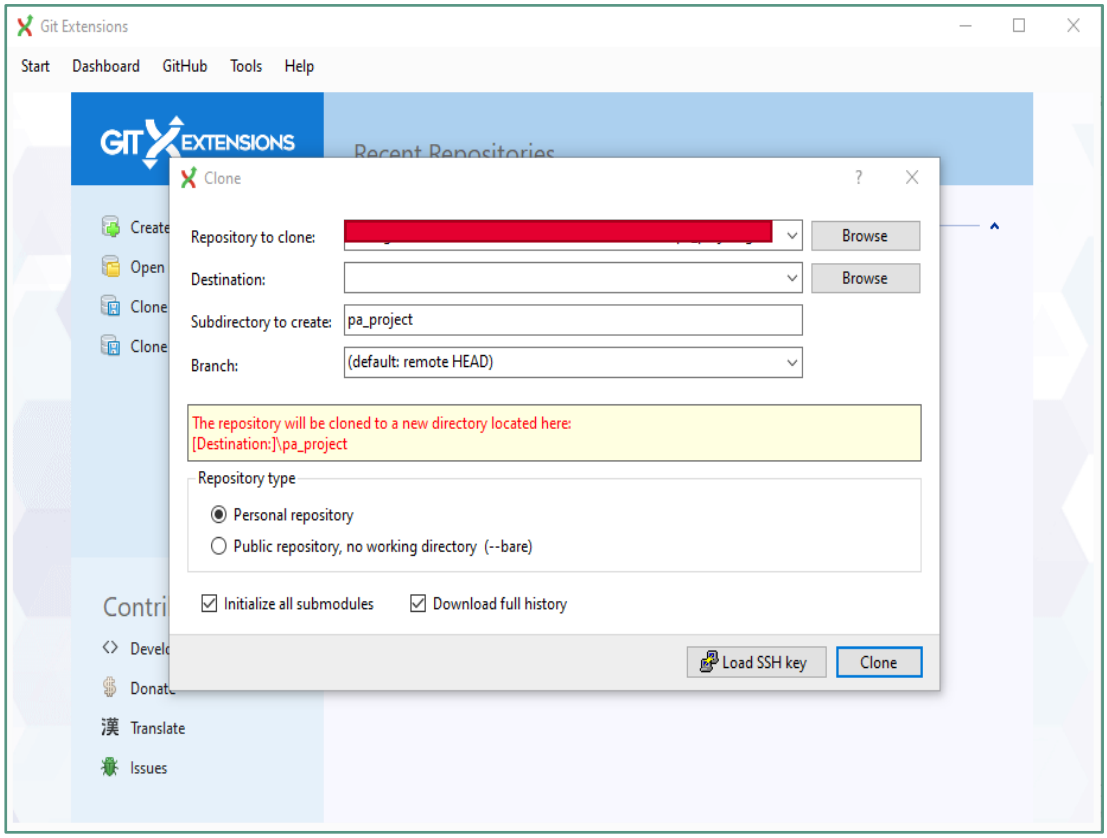


Figure 6.18: Files pushed to the Bitbucket using the Git extension software

Now we know that we are manually running the MATLAB execution and when we are pushing the files to Bitbucket and executing using the Jenkins there must be a script that fetches all the required files and performs the execution and gives the results back to the Jenkins.

For this purpose, we are using the unit test framework of MATLAB. A script is written in MATLAB with some inbuilt headers of MATLAB unit test framework that fetches the specified test suite and executes all the test cases inside the test suite.

```
import matlab.unittest.TestRunner.*
import matlab.unittest.TestSuite.*
import sltest.plugins.ModelCoveragePlugin.*
import sltest.plugins.coverage.CoverageMetrics.*
warning off Stateflow:Runtime:TestVerificationFailed;
warning off Stateflow:cdr:VerifyDangerousComparison;
%%Coverage parametres
%mcdcMet = CoverageMetrics('Decision',true,'Condition',true,'MCDC',true);

%covSettings = ModelCoveragePlugin('RecordModelReferenceCoverage',true,...
    %'Collecting',mcdcMet);
%%Open test suite containing the required test harness
pasuite = testsuite("function_test.mldatx");
runner = TestRunner.withTextOutput;
%addPlugin(runner,covSettings);

%%Generate PDF report
import matlab.unittest.plugins.TestReportPlugin.*
pdfFile = 'Report.pdf';
trp = TestReportPlugin.producingPDF(pdfFile);
addPlugin(runner,trp)
%%Add Test Manager results to report
import sltest.plugins.TestManagerResultsPlugin.*
```

Figure 6.19: Snapshot of the MATLAB script developed for the automation of the test suite

The script explains the system to fetch the test suite to be execute, then which all coverage parameters have to be verified, generate a coverage result in addition to the general test report created by the test suite. It also specifies the type of document to be created for the report. Here, we have used the PDF format for generating the report as the report will be portable.

Jenkins is configured in such a way that for every new successful commit in Bitbucket it will create a pull request. Now we have to push this script to Bitbucket and whenever a pull request is created inside the Bitbucket, the Jenkins will execute this script file. For this to happen we have created a Jenkins file written in Groovy language that will select which node to be executed when a pull request is generated and who all should be notified at the end of execution of the project. The execution and the stages of execution that are happening inside the MATLAB console are visible from the Jenkins console.

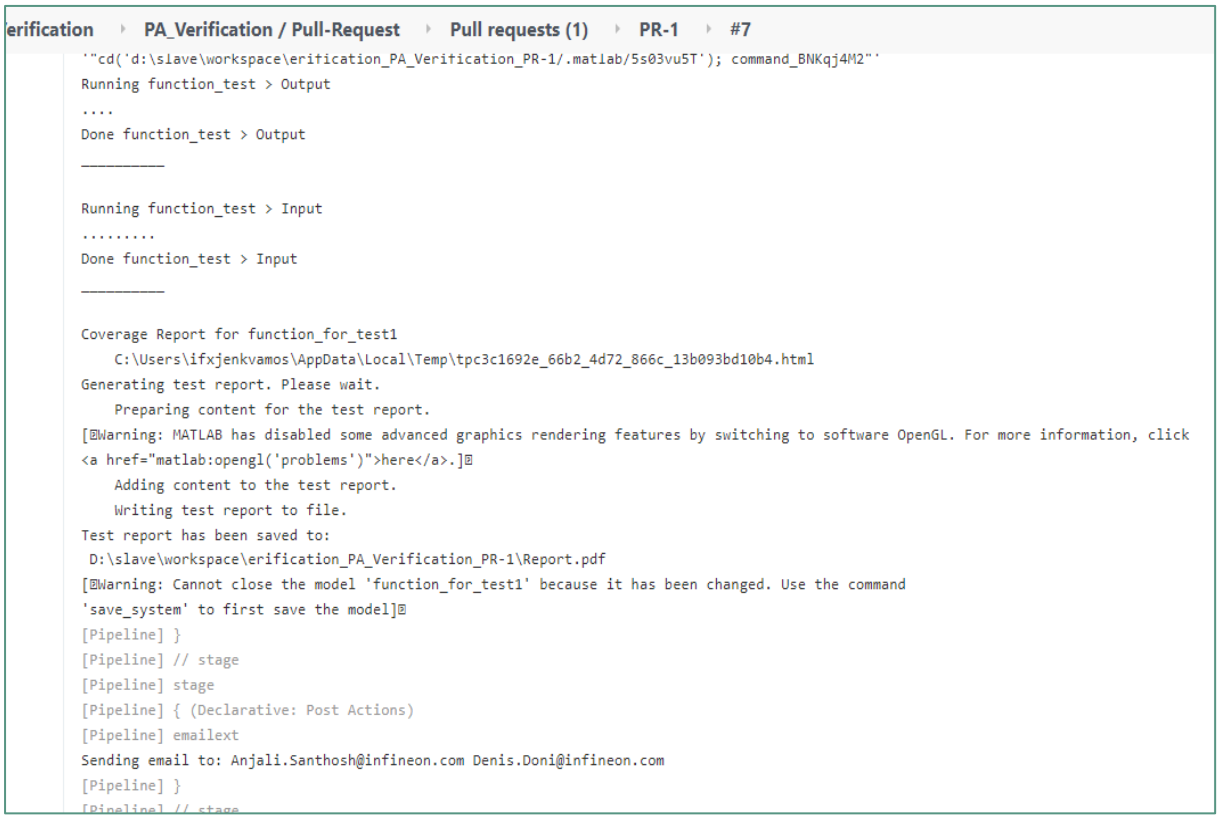


Figure 6.20: Jenkins Console Output showing a MATLAB execution

With this setup, whenever a new commit is made to the bitbucket a pull request will be created and Jenkins will run the verification of the product architecture and provide the report. There is no manual effort required for this from the test engineer unless there is a change in the product architecture. In most of the cases, regressions do not require the change in the product architecture or change in model.

6.5 MATLAB INBUILT APPROACH V/S DESIGNED APPROACH

We have designed the test suite with six test cases and we claim that it is the best approach. We have also proved that we are able to obtain hundred percent coverage with this approach. Now to prove that our approach is the best as compared to the inbuilt approach in

MATLAB we are performing the development of test cases with the MATLAB Design verifier and evaluating both approaches.

MATLAB has an inbuilt feature named Design verifier which will generate all the possible test cases for a system after verifying the system. Therefore, we are generating the test cases for the system with the Design verifier. The Design verifier of MATLAB for the product architecture shows that there are 437 outcomes to be satisfied for the design verifier to generate the test cases in which there are outcomes that cannot be satisfied by the design verifier and there are outcomes that are unsatisfied due to run time errors. Thus, we were able to obtain the following coverage for the same product architecture using the design verifier method.

Model Hierarchy/Complexity	Test 1		
		Decision	Condition
1. MainBoot0	209	60%	65%
2. . . . Main boot	208	60%	65%
3. . . . . SF: Main boot	207	60%	65%
4. . . . . SF: active	73	52%	57%
5. . . . . SF: activeModes	59	45%	50%
6. . . . . SF: bsl	29	53%	63%
7. . . . . SF: debug	7	83%	100%
8. . . . . SF: loopROM	10	14%	38%
9. . . . . SF: user	5	0%	NA
10. . . . . SF: protMem	11	71%	70%
11. . . . . SF: resetStatus	3	100%	NA
12. . . . . SF: branch	29	46%	52%
13. . . . . SF: checkNVM	10	50%	75%
14. . . . . SF: checkRAM	14	72%	88%
15. . . . . SF: config	22	50%	50%
16. . . . . SF: reset	20	75%	78%
17. . . . . SF: startupInit	7	60%	75%

Figure *6.21: Coverage report for each block using the design verifier approach*

As the Figure 6.21 shows, the design verifier approach was able to generate 65 percent coverage for Condition and 60% coverage for the decision matrics for the product architecture. Here, the main boot corresponds to the product architecture. Now we are comparing each block

of the product architecture coverage obtained using the best approach with the results obtained using the design verifier. The graphical representation below explains the results.

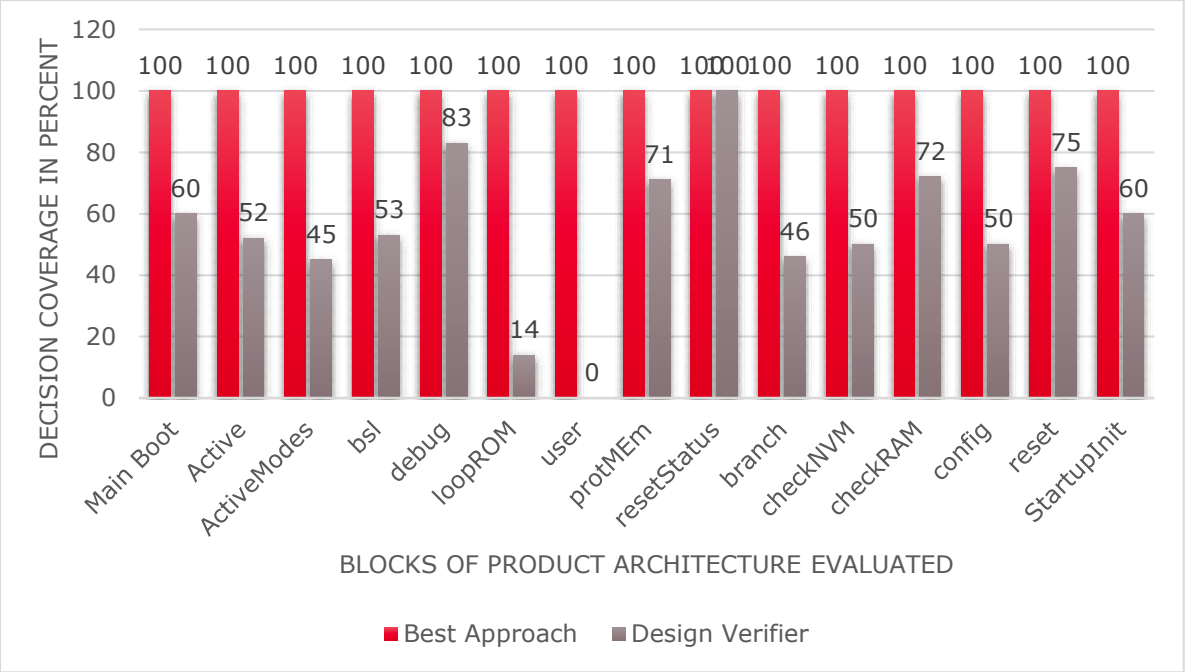


Figure 6.22: Decision Coverage obtained for the Product Architecture Verification using the Designed approach (Best Approach) and the MATLAB design verifier

On comparing the data, we can conveniently prove that the test suite designed by following the results of theoretical evaluation is the best method to approach a system under test and where the coverage parameters as well as the time consumption to design the test cases are a concern.

The Design verifier has taken 60000 seconds to generate and execute the test cases for the product architecture. In the mean-time, a test engineer will be able to generate the desired results within a time frame of fifteen to twenty minutes using the derived approach. We can also compare the condition coverage obtained for the model using both the approaches for the sake of understanding.

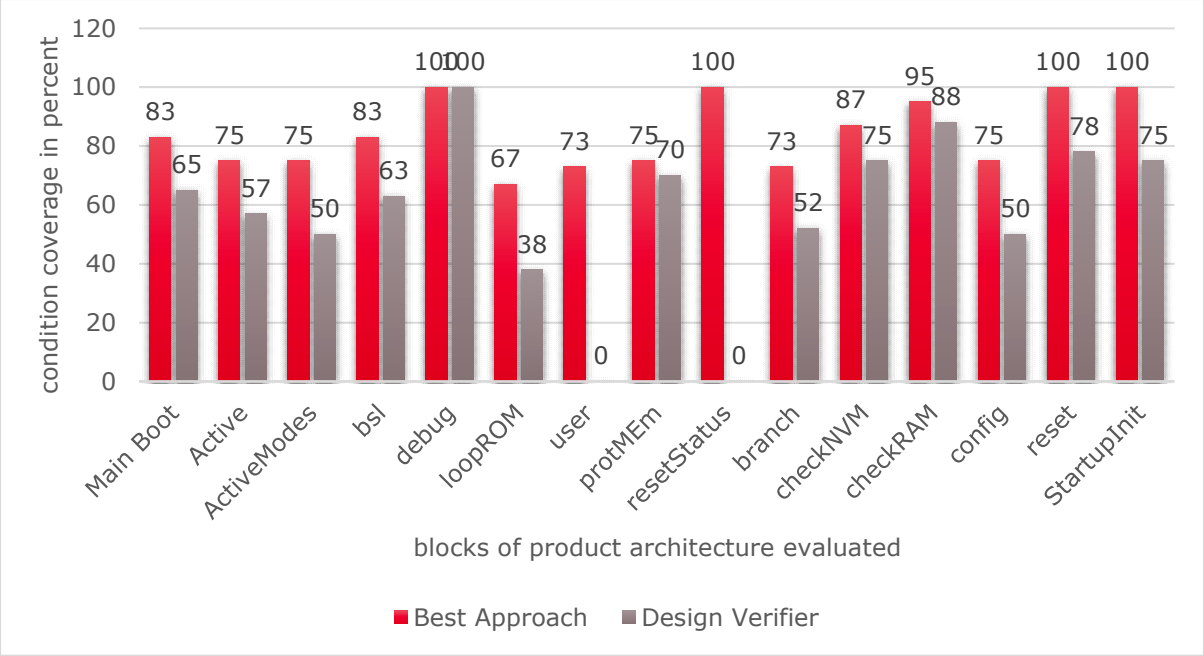


Figure 6.23: Condition Coverage obtained for the Product Architecture Verification using the Designed approach (Best Approach) and the MATLAB design verifier

Thus, from both the plots of Condition and Decision coverages obtained for the system we are able to achieve best coverage results for the system with minimum number of test cases and less time as compared to the inbuilt approach in MATLAB. This comparison is made in order to prove the reliability and efficiency of the new approach designed as compared to the already available and known approach.

To develop the test scenarios using the design verifier, to run the system in four different modes we need to generate the test suite four times by loading the inputs to the system from the spreadsheet independently. Each run will be having 439 outcomes to be satisfied and 439 test cases to be generated. Thus, with the Design verifier approach there will be 1756 test cases to be generated.

With the developed best approach, we will be having one test case in each mode and two test cases for the constant inputs to be fluctuated. Thus, with our approach we will be satisfying 179 outcomes using 6 test cases.

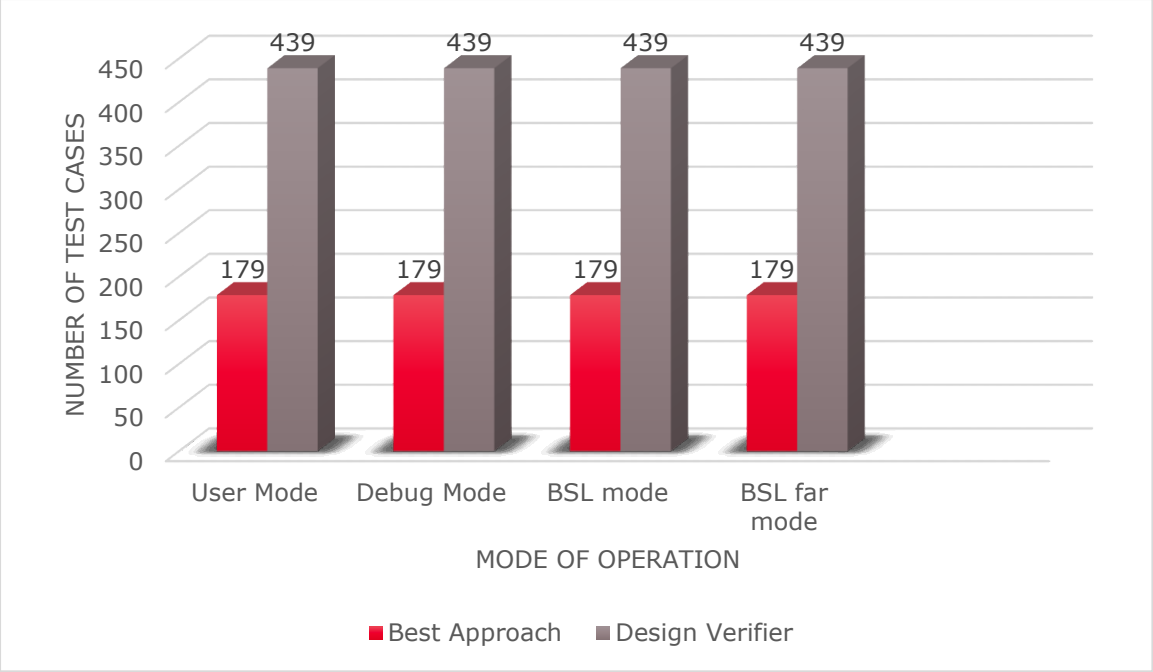


Figure 6.24: Number of test cases generated in each mode of operation

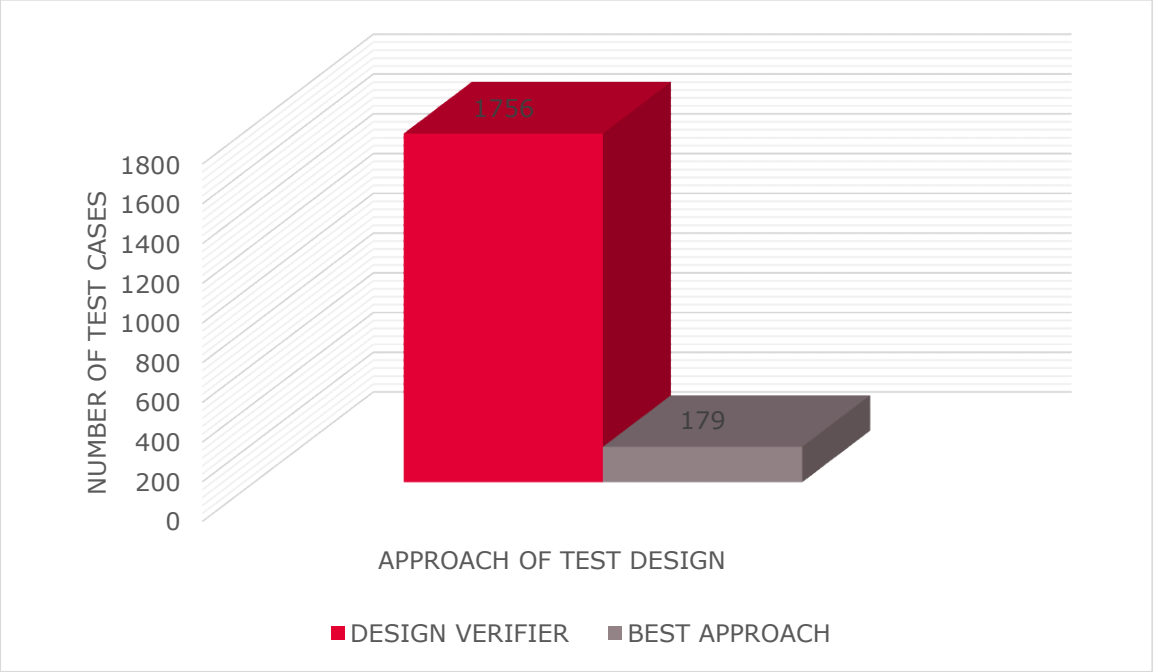


Figure *6.25: Total number of test cases generated using design verifier and proposed approach*

6.6. EXPECTED RESULTS VS ACHIEVED RESULTS

Even though the main aim of the verification plan is to achieve hundred percent coverage for the system under test in defined matrices of coverage, we need to verify whether we are receiving the correct set of outputs for our verification. As mentioned in the earlier

paragraphs, we are executing the system under test without creating a test harness and the initial results obtained for the execution are taken as a base line for the comparison.

One another commonly used method for verifying the results is that we manually predict all the possible expected results and load that values as the base line for the execution. These values are then compared with the obtained results and hence the outputs are verified.

The results obtained by running the model manually with feeding inputs individually to each mode of operation are noted and are set for the comparison. Once we have received the outputs from the test suite execution the comparison between both outputs are made for each mode of operation. The verification of outputs is done for any changes occurred.

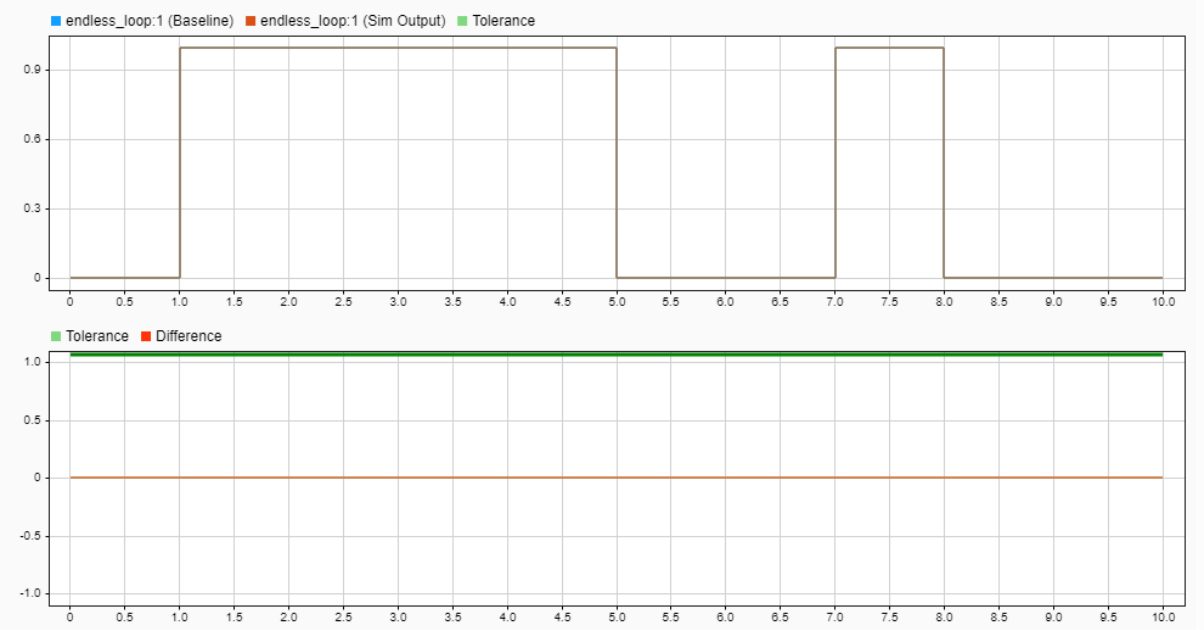


Figure 6.26: Comparison of expected result for endless_loop signal base line with the system simulation output

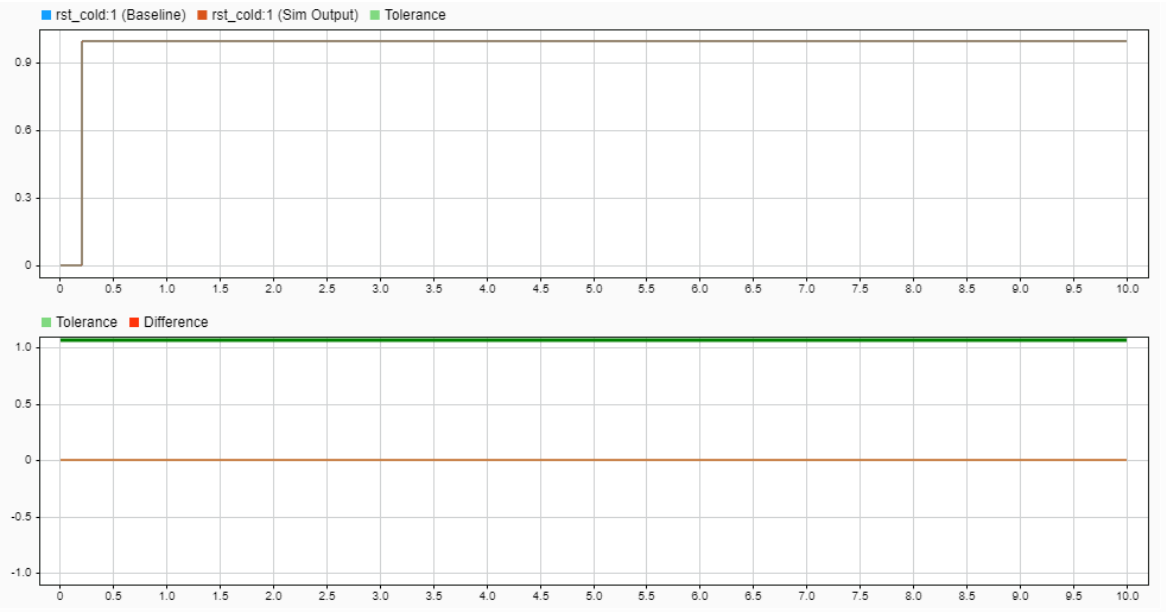


Figure 6.27: Comparison of rst_cold signals base line and simulation output

The difference achieved for all the signals during the simulation with the baseline signals are zero and the tolerance with the two output signals as one. Thus, with these comparison values we can understand that we have received the expected results at the end of execution.

CHAPTER 7: CONCLUSION AND FUTURE WORK

The project is based on the product architecture verification of an automotive microcontroller developed by Infineon Technologies Italia, conducted at the Infineon site Padova.

The early-stage product architecture verification is an important step to be performed at the initial stage of the product development as per the ASPICE standard which follows the Model Based Software Engineering. The project focuses on achieving two main goals such as:

- To find the best technique or method to approach a system under test with minimum effort and maximum coverage and,
- Design a test suite for the product architecture verification of an automotive microcontroller using the MATLAB Simulink Test Harness.

The initial chapters of the project deals with the theoretical evaluation of a system under test to derive formulas with which a test engineer can decide whether to go ahead with the input-based approach or output-based approach to test a system to receive the maximum coverage. Three formulas were derived successfully with which the test engineer is able to decide the same.

The second part of the thesis deals with the development of test suite in MATLAB for verifying the product architecture of the automotive microcontroller. A Simulink test harness is developed and with six test cases inside the test suite we are able to achieve hundred percentage decision coverage for the system.

We have followed the input-based approach to design the test suite and the number of test cases required to test such a complex product architecture is comparatively really small when comparing to the MATLAB default approach. The MATLAB's design verifier enables the user to generate test cases for a system under test. On using the design verifier, the design verifier took 60000 seconds to generate a test suite with 439 outcomes in which only 60% of the decision and 65% of the conditions were satisfiable.

Through the proposed approach we are able to achieve hundred percentage coverage for the decision matrix and 83 percent coverage for the condition matrix for the same product architecture with very a smaller number of test cases that is 179 outcomes in six test cases as we have explained in chapter six.

The future scope of the work conducted will be the environment created for the verification will be used in future for the verification of product architecture inside the organization. The theoretical equations developed will be used inside the organization to understand the workload on verification activities and optimize the procedure. As a future work this analysis approach can be applied to other metrics of coverage, for the theoretical evaluation of best approaches for other metrics of coverage [37].

From the analysis we are able to conclude that the proposed input-based approach based on the theoretical evaluation of the system is an efficient and the best-known way to approach a system under test especially in the early-stage verification of product architecture.

The automation of these test cases or the test suite with MATLAB unit test framework makes the task effortless for the test engineers during the regression testing as once the model is committed to the git repository for every update made an automatic pull request is created and the test reports will be available for the test engineer.

BIBLIOGRAPHY

[1]. R. Cloutier, B. Sauser, M. Bone and A. Taylor, "Transitioning Systems Thinking to Model-Based Systems Engineering: Systemigrams to SysML Models," in IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 45, no. 4, pp. 662-674, April 2015, doi:.

[2]. J. Holt, S. Perry, M. Brownsword, D. Cancila, S. Hallerstede and F. O. Hansen, "Model-based requirements engineering for system of systems," 2012 7th International Conference on System of Systems Engineering (SoSE), 2012, pp. 561-566, doi: 10.1109/SYSoSE.

[3]. V. Stoico, "A Model-Driven Approach for Early Verification and Validation of Embedded Systems," 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2021, pp. 684-688, doi: 10.1109/MODELS-C53483.20.

[4]. I. Traore and D. B. Aredo, "Enhancing structured review with model-based verification," in IEEE Transactions on Software Engineering, vol. 30, no. 11, pp. 736-753, Nov. 2004, doi: 10.1109/TSE.2004.86.

[5]. "Simulation-based verification of system requirements: An integrated solution," 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC. F. Aiello, A. Garro, Y. Lemmens and S. Dutré,),. 2017, pp. 726-731, doi: 10.1109/ICNSC.2017.80.

[6]. T. Gerlitz and S. Kowalewski, "Architectural Analysis of MATLAB/Simulink Models with Artshop," 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2016, pp. 307-310, doi: 10.1109/WICSA.2016.54.

[7]. K. Barber, M. Ghaniyoun, Y. Zhang and R. Teodorescu, "A Pre-Silicon Approach to Discovering Microarchitectural Vulnerabilities in Security Critical Applications," in IEEE Computer Architecture Letters, vol. 21, no. 1, pp. 9-12, 1 Jan.-June 2022, doi: 10.1.

[8]. FAQ, Model Based Engineering Forum. Model Based Engineering Forum. [Online] PivotPoint Technology Corp. https://modelbasedengineering.com/faq/.

[9]. R. Maschotta, M. Hammer, T. Jungebloud, M. Khan and A. Zimmermann, "Model-Driven Aspect-Specific Systems Engineering in the Automotive Domain," 2021 IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE), 2021, pp. 1-8.

[10]. Rethinking Software Testing Based on Software Architecture. Juan Jin, Fei Xue. Beijing, China : IEEE, 2011. ISBN:978-1-4577-1323-1.

[11]. Architecture evaluation in continuous development. S Magnus Ågren, Eric Knauss,Rogardt Heldal,Patrizio Pelliccione,Anders Alminger,Magnus Antonsson,. C, United States : Journal of Systems and Software,Elsevier Science Inc., 2022, Vol. 184. ISSN:0164-1212.

[12]. R. Baddour, A. Paspaliaris and D. S. Herrera, "SCV2: A model-based validation and verification approach to system-of-systems engineering," 2015 10th System of Systems Engineering Conference (SoSE), 2015, pp. 422-427, doi: 10.1109/SYSOSE.2015.7151960.

[13]. K. Kim, H. Kim, S. Kim and G. Chang, "A Case Study on SW Product Line Architecture Evaluation: Experience in the Consumer Electronics Domain," 2008 The Third International Conference on Software Engineering Advances, 2008, pp. 192-197, doi: 10.1109/ICSEA.

[14]. X. Xia, K. Qu, J. Shi, Z. Fan and L. Xu, "The construction of effectiveness evaluation model based on system architecture," 2017 IEEE International Systems Engineering Symposium (ISSE), 2017, pp. 1-4, doi: 10.1109/SysEng.2017.8088319.

[15]. Taeho Kim, In-young Ko, Sung-won Kang and Dan-hyung Lee, "Extending ATAM to assess product line architecture," 2008 8th IEEE International Conference on Computer and Information Technology, 2008, pp. 790-797, doi: 10.1109/CIT.2008.4594775.

[16]. F. G. Olumofin and V. B. Misic, "Extending the ATAM Architecture Evaluation to Product Line Architectures," 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005, pp. 45-56, doi: 10.1109/WICSA.2005.33.

[17]. DSP programming using MATLAB Simulink embedded coder: Techniques and advancements,". A. Elrajoubi, S. S. Ang and A. Abushaiba, ". 2017, 2017 IEEE 18th Workshop on Control and Modeling for Power Electronics (COMPEL),, Vol. doi: 10.1109/COMPEL., pp. , pp. 1-7, .

[18]. J. Börcsök, W. Chaaban, M. Schwarz, H. Sheng, O. Sheleh and B. Batchuluun, "An automated software verification tool for model-based development of embedded systems with simulink®," 2009 XXII International Symposium on Information, Communication and Automa.

[19]. N. He, V. Oke and G. Allen, "Model-based verification of PLC programs using Simulink design," 2016 IEEE International Conference on Electro Information Technology (EIT), 2016, pp. 0211-0216, doi: 10.1109/EIT.2016.7535242.

[20]. "Version control and patch management of protection and automation systems,. D. Jenkins, J. Arnaud, S. Thompson, M. Yau and J. Wright,". 12th IET International Conference on Developments in Power System Protection (DPSP 2014), 2014, pp. 1-4, doi: 10.1049/cp.

[21]. Y. Yu, H. Wang, V. Filkov, P. Devanbu and B. Vasilescu, "Wait for It: Determinants of Pull Request Evaluation Latency on GitHub," 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 367-371, doi: 10.1109/MSR.2015.42.

[22]. U. Michieli and L. Badia, "Game Theoretic Analysis of Road User Safety Scenarios Involving Autonomous Vehicles," 2018 IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), 2018, pp. 1377-1381, doi: 10.1109/P.

[23]. E. Gindullina, S. Mortag, M. Dudin and L. Badia, "Multi-Agent Navigation of a Multi-Storey Parking Garage via Game Theory," 2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2021, pp. 280-285, doi: 10.

[24]. V. B. Kleeberger, S. Rutkowski and R. Coppens, "Design & verification of automotive SoC firmware," 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015, pp. 1-6, doi: 10.1145/2744769.2747918.

[25]. R. Letor, G. Di Caro, A. Russo and N. Liporace, "Mixed system integration simplifies the design and the architecture of automotive power actuators," 38th IAS Annual Meeting on Conference Record of the Industry Applications Conference, 2003., 2003, pp. 125.

[26]. D. Sargsyan, "Firmware Generation Architecture for Memory BIST," 2018 IEEE East-West Design & Test Symposium (EWDTS), 2018, pp. 1-4, doi: 10.1109/EWDTS.2018.8524853.

[27]. D. Bilyi and D. Gerling, "Modeling of automotive power network for analysis of power electronics and losses calculation and verification by measurements on claw-pole alternator," 2016 IEEE 8th International Power Electronics and Motion Control Conference.

[28]. G. S. Harinarayan, M. Rana, N. Pant, M. Bansal, S. Sharma and N. Kaundal, "Automated Full Chip SPICE simulations with self-checking assertions for last mile verification & first pass Silicon of mixed signal SoCs," 2016 29th IEEE International System-on-Ch.

[29]. B. Schlich, F. Salewski and S. Kowalewski, "Applying Model Checking to an Automotive Microcontroller Application," 2007 International Symposium on Industrial Embedded Systems, 2007, pp. 209-216, doi: 10.1109/SIES.2007.4297337.

[30]. H. Kaindl, F. Lukasch, M. Heigl, S. Kavaldjian, C. Luckeneder and S. Rausch, "Verification of Cyber-Physical Automotive Systems-of-Systems: Test Environment Assignment," 2018 IEEE International Conference on Software Testing, Verification and Validation W.

[31]. G. Bahig and A. El-Kadi, "Formal Verification of Automotive Design in Compliance With ISO 26262 Design Verification Guidelines," in IEEE Access, vol. 5, pp. 4505-4516, 2017, doi: 10.1109/ACCESS.2017.2683508.

[32]. T. Nguyen and A. -D. Basa, "Verification methodology of sophisticated automotive sensor interfaces integrated in modern system-on-chip airbag system," IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society, 2013, pp. 2335-2340, doi.

[33]. "IEEE Standard for Software Verification and Validation Plans, " in IEEE Std 1012-1986 , vol., no., pp.1-87, 14 Nov. 1986.

[34] ASPICE Documentation, Official document available on ASPICE website https://www.automotivespice.com/fileadmin/software-download/AutomotiveSPICE_PAM_31.pdf

[35] Juan Jin, Fei Xue, Re-thinking software testing based on software architecture, 2011 Seventh International Conference on Semantics, Knowledge and Grids

[36] A. Haghighatkhah, M. Oivo, A. Banijamali and P. Kuvaja, "Improving the State of Automotive Software Engineering," in IEEE Software, vol. 34, no. 5, pp. 82-86, 2017, doi: 10.1109/MS.2017.3571571.

[37]. Cumulative coverage analysis, Mathworks website, https://uk.mathworks.com/help/slcoverage/ug/cumulative-coverage-analysis.html