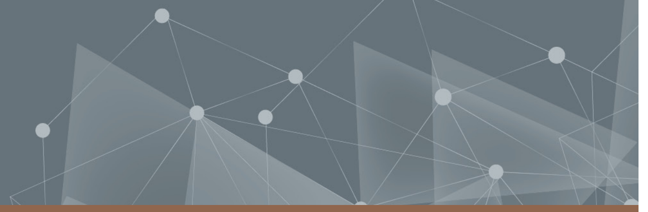

Automated testing of Simulink models for real-time test environments

Master thesis report

Master's thesis in Automotive Engineering

DINESHKUMAR BALAKRISHNAN ASOKAN

DEPARTMENT OF MECHANICS AND MARINETIME SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Automated testing of Simulink models for real-time test environments

Master thesis report

Dineshkumar Balakrishnan Asokan

Department of Mechanics and Marinetime Sciences
Division of Automotive Engineering
Chalmers University of Technology
Gothenburg, Sweden 2023

Automated testing of Simulink models for real-time test environments
Master thesis report
Dineshkumar Balakrishnan Asokan

© Dineshkumar Balakrishnan Asokan, 2023.

Supervisor: Fabien Desvignes, HIL Development, VolvoCarsCorporation
Examiner: David Sedarsky, Associate Professor, Energy Conversion and Propulsion Systems

Master's Thesis 2023
Department of Mechanics and Marinetime Sciences
Division of Automotive Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Automated testing of Simulink models for real-time test environments
Dineshkumar Balakrishnan Asokan
Department of Mechanics and Marinetime Sciences
Chalmers University of Technology

Abstract

During the development phase of next-generation electric vehicles, which combine embedded systems and hardware with hundreds of thousands of lines of code, continuous testing and fast feedback are essential. To assure the highest possible quality, we utilize Hardware-In-the-Loop (HIL) simulators which enable software testing on the target electronic control unit (ECU) before there even is a car.This testing requires high-quality and reliable HIL models that simulate the ECU interfaces, as in the car, and live up to the requirements for SW testing. An ever-growing amount of different vehicle and drivetrain variants also impacts our real-time simulation models, so they become more complex and require better and faster testing

This thesis focuses to automate the design and verification of simulation environments modelled in Simulink, and to create an automated process that generates tests and verifies our Simulink models. This will be an enabler to increase the pace of our deliveries and improve the quality of our real-time testing environments, crucial for electric vehicle software development.

Keywords: Hardware-In-the-Loop

Acknowledgements

I would like to take a moment to express my deep appreciation to Carol and Peter for their exceptional support and guidance during my thesis work at Volvo. Their encouragement and feedback were invaluable in keeping me motivated and focused throughout the project, and I consider myself fortunate to have had the opportunity to work with such exceptional managers.

I would also like to extend my sincere thanks to Axleander for offering me the chance to work on my thesis under the HIL development and framework and to Fabien for his invaluable mentorship and support. His guidance and expertise were instrumental in shaping my research and bringing it to fruition.

Additionally, I would like to express my gratitude to my college supervisor, David, whose invaluable supervision and feedback were crucial in helping me achieve my research goals and for answering all my queries quickly, helping me with all the administrative processes and constantly checking on the thesis work. Moreover, I am grateful to my colleagues at Nexer for their cooperation and support, which made the research work a lot smoother.

A special word of thanks should also go out to my parents and grandparents, whose unflinching faith in me and support got me through the tough times. I credit their love and support for helping me succeed to their dreams and hopes that I would graduate with a master's degree. I would want to express my sincere gratitude to my close friends, whose steadfast support and direction significantly aided in my decision to seek my goal.

I am truly grateful for all the help and support provided by the whole team throughout my thesis. It has been an excellent learning experience, and I feel privileged to have had the opportunity to work with such an exceptional team. Thank you once again for your invaluable contribution to my academic success.

Dineshkumar Balakrishnan Asokan, Gothenburg, August 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ADK	Advace Development Kit
BEV	Battery Electric Vehicle
CD	Continious Deployment
CI	Continious Integration
CAN	Controller Area Network
E2E	End to End
GPA	Global Producable Architectue
HEV	Hybrid Electic Vehicle
HIL	Hardware In Loop
HV	High Voltage
HW	Hardware
JSON	JavaScript Object Notation
LV	Low Voltage

Contents

List of Acronyms **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Problem description . . . 1
1.2 Aim and objectives . . . 2
1.3 Systematic Investigation . . . 2

2 Theory **5**
2.1 Simulink Model Extraction and Libraries . . . 5
2.1.1 Plant . . . 6
2.1.2 Application . . . 6
2.1.3 IO . . . 6
2.1.4 HIL . . . 7
2.2 Version Control . . . 7
2.3 Pipeline . . . 8

3 Methods **11**
3.1 Prerequisites . . . 11
3.2 Work Flow . . . 12
3.3 Modeling Guidelines . . . 13
3.4 Approach 1 . . . 15
3.4.1 Feedback . . . 17
3.5 Approach 2 . . . 18

4 Results and conclusions **21**
4.1 Initial observations . . . 21
4.2 Simulation work . . . 21
4.3 Summary . . . 23

5 Discussions and future scope **25**
5.1 Limitations . . . 25
5.2 Future Research Directions . . . 25

Bibliography 29

A Appendix: Source Code Implementation I

A.1 MATLAB Code Synopsis I
A.2 Model Configuration YAML IV
A.3 Python Code Synopsis VII

List of Figures

2.1 Layer of Simulink Block 5
2.2 Zuul Status 8

3.1 Focus Area on Modeling 13

5.1 Pick and Placing of Complete Simulink Models 26

List of Tables

1.1 Modeling Development Issues 2

1
Introduction

This section explains the problem in focus, the research performed in the field and the objectives aimed to reach in this project and outlines the methodology employed to attain the desired outcomes.

1.1 Problem description

In the revolution of Automotive Engineering and emergence of Software Defined Vehicles, software is been played as a key role for every new car released in the market. The new era of software defined vehicles refers to the idea that a vehicle's operation and behavior are managed and defined by software. Software defined vehicles are a product of modern car's growing integration of cutting-edge software and electronics. There has been a drastic advancement in the evolution of automotive market among the years, speaking of which software has outgrown the industrial growth of this market. When we take a look closer we know that there are space for further enhancements to be taken care in the software as well in order to deliver a car and in this fast pace next generation electric cars it has a lot more advancements to be done.

The more it involves software the more it can be outgrown, in recent days the fast pace software deliveries has been a major focus in order to have a updates with shorter development cycles and frequent releases. This involves complexity in developing and testing the software in a short duration of time and reiterating the whole process. The framework needs an Continuous Integration and Continuous Deployment (CI/CD) culture, The development, testing, and deployment procedures are automated using CI/CD framework methods, enabling quicker and more frequent releases[1]. Software updates may be swiftly integrated and deployed to production settings by developers merging their code changes into a common repository, which starts automated builds, tests, and deployments [2]. The more complex and various are the embedded activities and applications, the more complex, time-consuming and error-prone is the software [3]. My focus of study in this thesis is to automatically test the Simulink models for each software release, to lower the risk of manual errors we can automate as much as possible. Yet another purpose behind this work is to develop reliable software that fulfil system requirements and to test its behaviour during real- time hardware simulation, in order to achieve the validation step [3].

Based on the statistics in the table 1.1, it is clear that the vast majority of the problems are the result of poor implementation. Proactive steps are required to

Category	Count	Percentage %
Wrong implementation	14	33
User error or SW issue	5	12
HW error (HIL and Harness)	3	7
Wrong data, parameter or function due to missing information	9	21
CAN Implementation	7	17
E2E	4	10

Table 1.1: Modeling Development Issues

improve the overall quality and efficiency of our procedures. Conducting extensive code reviews to discover implementation issues early, giving comprehensive training to the development team, developing strong testing processes, and promoting a culture of cooperation and information sharing are examples of these strategies[4]. We can reduce mistakes, improve product reliability, and optimize our development workflows by addressing the core cause of wrong implementations, eventually leading to higher quality outputs. Currently there is very little way to enforce our models to follow the existing modeling guidelines such as having naming standards and model structures[5], lack of these would lead to a challenging task and have an automating framework and testing flow for the models. However, if such standards are established, it will lead to better re-usability and reliability of the models, while also enabling a faster pace for model on demand[6]. Once this is in place we can make sure the respective models are gated through the pipeline and verify if the model has met the requirements. This helps to identify and fix issues related to modeling style, syntax errors, and other potential bugs in model in the initial phase.

1.2 Aim and objectives

The primary aim of this study is to have simulink models been tested upon certain criteria even before reaching the build stage and have it tested on a real-time simulators. The main focus on those criteria are to have standardized hierarchy of models, define a standardized Bus name and Block Name. All of these define to have a MAAB which stands for MathWorks Automotive Advisory Board. This introduces best practices for modeling, simulating, and verifying embedded systems, with a focus on safety-critical systems in vehicles also help engineers improve the quality and reliability of their embedded systems by providing a standardized approach to development that is consistent with industry standards and regulations.

1.3 Systematic Investigation

Any new software development follows a certain procedure. The sequential and systematic process is prioritized by the V-shaped development and testing strategy, which is a software development methodology. It features a downward-facing "V" form[7], with the testing and validation phases shown on the right side and the planning, design, and coding phases on the left. The initial phase would be de-

pending on the needs, developers work on unique Simulink models. To guarantee code quality and traceability, they adhere to coding standards, best practices, and version control. This makes life easier to have versions for different releases[8], the integrators combine the many Simulink models created by various developers into a complete system. They make that the models interact and interface as intended by checking their interactions and interfaces. Unit testing is done before the complete software is released and tested on the hardware. To further verify the Simulink models behavior, SIL testing entails running the models in a simulated setting[9]. A more thorough verification of the system results from the validation of a certain degree of functionality and integration throughout each testing step.

Later to iterate the process of building HIL models before testing on real-time systems, the primary goal is to set certain standards and make sure the build stage is bypassed after the introduction of the matlab scripts which are linked to the Zuul pipeline to give a verdict in order to proceed to the build stage to complete the HIL model[9]. Which is then tested on the real-time systems. Hence this increases the reliability of the models built for multiple projects. Later the few test scenarios are also taken through the Box-car where it is similar to the HIL setup but some of the environmental factors are for real. All these testing process ensures higher quality, better traceability, and increased efficiency in the development process.

Each of the individual stages of testing in various environment are been run over an automated framework, Once the integration of scripts are done they are fed to pipeline and respective test scenarios are performed, the results obtained are visualised and analysed. Git/Gerrit are used for version controlling, Jenkins and Zuul is used as an CI/CD tool to have seemless deployment of the software. Having an simulink models built error free and compliant to our guidelines before build is the major goal in this thesis study.

2

Theory

This section is to provide a clear understanding of the underlying principles and concepts that are essential for comprehending the thesis work. The section includes the fundamental ideas and models that are been used to study throughout[10]. Each layer of the model is been over-viewed in accordance to Volvo's upcoming architectural soft ECU platforms. A software module can be only belong to a single layer, currently the HIL model are been developed for all project nodes and software seire.

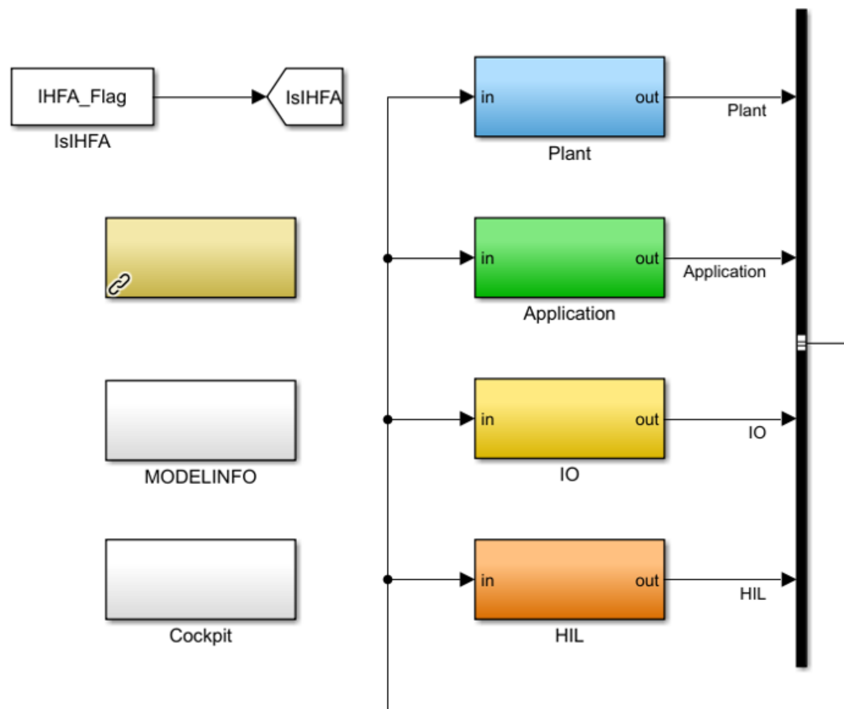

Figure 2.1: Layer of Simulink Block

2.1 Simulink Model Extraction and Libraries

Model consists of many different subsystems which can be linked to a library, in some cases used by libraries. Simulink libraries encourage standardization and modularity, which facilitate effective modeling. This enables users to use built-in Simulink li-

braries like the Simulink block library and the Simulink Coder library or to develop their own unique custom libraries. Libraries offer a central location for reusable parts[10], which speeds up development, encourages uniformity, and lowers error rates.

This leads to have enough libraries to create a soft ECU which aims to be more generic when used in various different ECU's when having dependencies. This is commonly done to ease communication between many team members working on different portions of the model, modularize complicated systems, improve reusability, improve readability, or all of the above. The Simulink model can be made easier to comprehend, maintain, and adapt by isolating and arranging particular components. The Model consists of similar main subsytems which are as mentioned.

2.1.1 Plant

The Plant has the physical systems around the vehicle. It consists of physical entities like electricity, speed, vehicle motion, vehicle stability and many more.In the context of a vehicle, refers to the collection of physical systems that are directly associated with or connected to the vehicle. The Electronic Control Unit (ECU) is meant to believe that it is functioning within a real automobile, and Plant as a system is intended to emulate the physical systems necessary to achieve that illusion. It offers a digital depiction of the elements and capabilities required to faithfully replicate the operation of a real vehicle. One such example is the HighVoltage DC Link is modelled so that when simulating closing contactors (to energize the DC link and supply all loads) the Electric Drive ECU can measure a simulated High voltage allowing it to activate its main functionalities.

2.1.2 Application

Application in general is a soft ECU implementation which intents to simulate the software in the other nodes that interact with our ECU under test via various communication protocols such as CAN, LIN. A generic model of the entire SWF of the car is used to make it easily reusable and platform independent. This subsystem's key goals within the project are to create good communication between the many nodes involved and to closely monitor and replicate the functionality of these ECUs. The subsystem is crucial in maintaining the efficient functioning and coordination of the entire system by combining the software ECUs and providing seamless communication.

2.1.3 IO

IO as the name expands Input/Output it refers to the communication and interaction between electronic devices or systems. The computers, communication networks, and micro controllers. It enables interaction between electronic equipment and the outside world, the processing of inputs, and the production of outputs for control, display, or communication. The power supply is mainly connected to test

objects and controller of the simulator. Sensor simulation, power supply management, and monitoring are common uses for analog and digital inputs and outputs.. CAN networks between ECUs and more seldomly, LIN between sensor/actuators and control unit.

2.1.4 HIL

HIL is the control and the monitoring of the Simulator itself. The essential interfaces, I/O capabilities, and simulation tools are provided by HIL systems, enabling thorough testing by simulating the behavior of the real system. It has various different power supplies from various different sources. It has a response over the voltage and current over other systems which are interlinked to the HIL environment.

2.2 Version Control

Version control is a key element of continuous integration (CI), version control facilitates efficient teamwork, code synchronization, automated builds, version labeling, and issue resolution[8]. It offers a trustworthy and organized method of managing code changes and supporting continuous integration and delivery processes, it also supports software quality. The idea behind version control is that to have a seamless working environment and have no overlaps or conflicts in one's development[11]. The thesis study involves version control of certain tools and framework for enabling the simulink model to match its requirement.

The version control is also done over binaries on the .slx files over different projects after every new development phase. In the organisation they have various different branches for various reasons. One such example is a 'feature' branch of a project is for development and a 'master' branch is for deployment of the project as a product to the next phase during integration. Version control systems serve as the central code repository in CI. Developers commit their code changes to the version control system, which maintains a complete history of all modifications. This allows for easy collaboration and provides a single source of truth for the codebase, this also enables to trigger the pipeline which enhances the continous deployment and integration of a project.

One prime advantage over version control is that since everything is merged on the new changes, when an issue is been raised it can be tracked down to the exact chain of changes and it can be reverted to exactly to that point of instance looking at the command history and fix the specific changes responsibly[12]. This gives the developers a leverage on not to crash any system with a faulty code or model. This helps maintain the stability and quality of the software.

2.3 Pipeline

The pipeline is designed to automate and streamline the software delivery process, promoting frequent and consistent integration, testing, and deployment[13]. It ensures that code changes are validated early and often, leading to improved software quality, faster feedback cycles, and a more efficient development workflow. The pipeline can be set in accordance to one's need. In the organisation they tend to have two pipeline services, one is Jenkins [14] which is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. Another service based tool is Zuul which is also a program that drives continuous integration [15], delivery, and deployment systems with a focus on project gating and interrelated projects. In this thesis we work closely on the Zuul pipeline where we create pipelines on the project requirements.

The pipeline enables to have an automated build and test environment which helps to catch errors and bugs in early phase of testing and repetitive iterations of checks can make the product more viable and ensure safety on these highly engineered products which value more on safety and quality standards[16]. Apart from build and test it also enables on running the project on different set of pipelines such as deployment and release phase, which in turn enables the integration and functional tests.

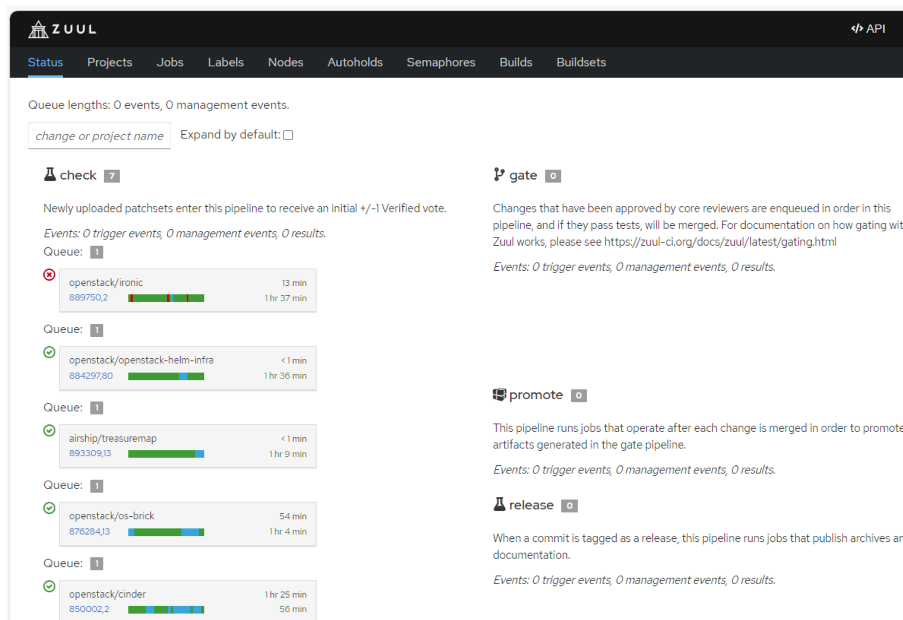

Figure 2.2: Zuul Status

In this thesis work, for enforcement guidelines of simulink models we place the logic in an separate repository which follows the Electric Propulsion Software architecture. These repositories are tracked down by zuul which helps to trigger the build environment when a new software is been built from a different repository. Zuul uses

Ansible playbook which automate and manage the deployment and configuration of software applications [17]. The various phases of the software delivery process are coordinated by Zuul, who serves as the orchestrator for the CI/CD pipeline. It organizes code reviews, engages version control systems, and starts the execution of jobs or tasks[18]. In contrast, Ansible is a potent automation tool that enables the provisioning, setting up, and deployment of infrastructure and applications.

Ansible playbooks are used in the context of Zuul to specify the intended state of the target system or environment[19]. Each job in these playbooks specifies a specific action that has to be taken, such as installing dependencies, setting up services, or deploying application code[20]. To suit the particular needs of the software project, the playbooks can be altered. In regards with the thesis work the jobs are defined in a experimental pipeline which then calls upon a playbook which is made sure to run the logic of the enforcement when ever a new model is been set for build. Zuul triggers a task, which then calls the appropriate Ansible playbook, supplying any required variables or arguments. The playbook's tasks are subsequently carried out on the target system by Ansible, ensuring the deployment and configuration goals are met.

Listing 2.1: Example of a zuul playbook

```
- job:
    name: example-job
    parent: base
    description: |
        This is an example job in the zuulplaybook.
    run:
        playbook: playbooks/example.yml

- project:
    check:
        jobs:
            - example-job

    gate:
        jobs:
            - example-job
```

3
Methods

In this chapter the focus will be on how to approach towards the goal, the above discussed architecture in chapter 2 have one common project for one node and many different software series, so we have to support many diff nodes and for various software series. This inturn becomes complex to maintain many different software series and store as many as binaries and support for older versions within HIL Teams. Refactoring and maintaining models would be so tedious that each software series has different variants involved. Hence, the thesis would focus on to create a test framework to check our own modeling guidelines. Based on user selection on selection for a variety of software series, project and nodes. This is to introduce recipe for various projects which results in a build (HV Battery system , LV Battery System etc) specific repository use the framework which we develop to have a strict enforcing on checks for all different models which are been created.

The idea of Model on demand needs a strong foundation and requires having a generic interface using the modeling guidelines and error free models which can achieve this for based upon the user selection, this in-turn also helps the user to can get rid of many duplicates and main unique HIL models per project[5]. Avoid cockpit updates manually for each projects and variants. Complexity of the Variant handling will be reduced. This will ensure we always have the latest model content. Having these guidelines also do not require to have complex understanding for each project rather to disrupt the workflow of developers for each project instead it would be a one time target. Reduces the maintenance of models and parallel development would be easier this would help to achieve the work on smaller units and to develop many more models as outcome[5].

3.1 Prerequisites

Acquiring access to the repositories related to Hardware-in-the-Loop (HIL) integration and framework teams marked a crucial milestone in the development process. Gaining entry to these repositories provided the necessary foundation for collaboration and ensured that the project could progress efficiently. With access granted, the next logical step was to establish the essential tools and software solutions that would facilitate the daily tasks and operations. The main tool selected for the project was Matlab2021a, which gave a full range of capabilities and features adequate for the project's needs. It was supplemented by dSpace licenses.

There are various instances to be followed in order to achieve our final goal, it is to have generic names to minimize the confusion among the 3 layers. Common (Vehicle, Platform, Solutions), Brain (Energy, Movement, Body and infotainment) Component (closed to hardware HV Battery), and External (Charging to charging stations). Use library linked subsystem, this helps in reusing everywhere possible, and being a generic library it will be a little less hassle to add a new signal if there are any needs in the project[2]. It is important that these links are been enabled in order to have a track over. This in future will help in to achieve parsing the signal name and following the convention and do the automation and like to pipeline to verify the model before build.

This gives an opener for loading the projects and models which are to be examined and performed the tasks on. To get to know indetail functionalities and properties of each class described I had to go through some of the generic functions used across various projects, Canblock, CockpitBlock, CockpitList, HILModel 2.1 and more helped to gain much more understanding on then implementation of each constructor class in each of the repositories.This allowed to understand the in-depth functionalities and properties of each class described. Examine the way you write your code and make sure your files are in the correct format and location so that any new implementations don't mess with how the project is being developed under the software framework guidelines.

The idea is also to have a unique configuration file for each project which maintains the hierarchy, naming standards and library links to be noted[18]. Software applications and systems must maintain an expandable configuration file as they develop and face new needs. Flexibility, modifiable, and adaptability are all guaranteed by a well-managed configuration file without affecting current settings. A consistent structure and naming standards in the configuration file is crucial for readability. Give settings names that are simple and descriptive, and group them logically. Developers will find it simpler to comprehend and edit the file as a result and finally to utilize version control tools for each project, such as Git/Gerrit[21], to keep track of configuration file modifications. Because it makes it simple to revert to earlier versions and leaves a record of all changes, it improves developer responsibility and cooperation.

3.2 Work Flow

The existing software development process in the organisation has a sequential flow from requirements and modeling through the build stage, when software is produced and tested for flaws. While this method has accomplished its function, a strategic shift is on the horizon. The goal is to include Zuul, a robust and adaptable gating system, into the process before proceeding to the build step. Furthermore, there is a concentrated attempt to apply internal modeling rules. From the figure 3.1 we can say that by introduction of Zuul early in the process represents a shift toward Continuous Integration (CI) and Continuous Deployment (CD) approaches. Code changes will be subjected to automated testing and validation earlier in the develop-

ment cycle, ensuring that they match with project goals and guidelines, with Zuul acting as a gatekeeper[15]. This method improves code quality by detecting errors earlier and decreasing the risk of bugs finding their way into the final result.

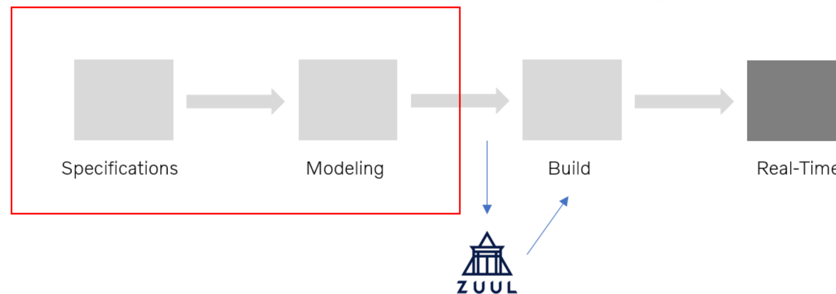

Figure 3.1: Focus Area on Modeling

Incorporating internal modeling guidelines at the same time emphasizes the necessity of uniformity and consistency throughout the development life cycle. These standards serve as a foundation for developing, documenting, and validating models. By following to these standards from the start, it becomes simpler to spot any flaws and inconsistencies early on, hence improving the software's quality. By implementing these improvements, it not only streamlines the development process, but also fosters a culture of quality and efficiency. It encourages to eliminates the need for major bug-fixing activities late in the development cycle. Ultimately, the objective is to provide more dependable software to real-time settings with more speed and confidence, fulfilling the needs of modern software development in a dynamic and competitive world.

3.3 Modeling Guidelines

Simulink has certain standards of modeling guidelines which are customary for all models being developed, one such majorly followed guidelines are MAAB. Mathworks Automotive Advisory Board (MAAB) is used while developing models and systems. Modeling guidelines are a collection of principles and best practices that direct the Model-Based Development (MBD) approach's process for developing and designing models. A special set of modeling rules created by MathWorks for the automobile sector is called Modeling automobile Adaptive Behavior. Simulink and Stateflow, software tools created by MathWorks for model-based design and simulation, are the primary modeling tools used in the MAAB recommendations for modeling automotive systems.

Signals shall flow from left to right[22], The direction of signal flow in automobile diagrams is now specified under this standard. As many cultures traditionally read from left to right, signals that flow from left to right consistently enhance readability and comprehension. Additionally, it facilitates a greater understanding of the behavior of the system and makes the mental process of following information

flow simpler. **Post of labels lines and blocks shall not overlap**[22], overlapping parts can create visual chaos and confusion. The model stays clear and unambiguous by making sure that labels, lines, and blocks do not cross over, which makes it simpler for users to recognize and differentiate between various components. A clean diagram improves the overall visual appeal and makes it easier to communicate the model's design. **Signal lines and bus lables shall be positioned below signal lines**[22], A consistent visual hierarchy is maintained in the model by placing bus labels and signal lines beneath the signals they represent. This layout option offers a clear and organized depiction of the system's data flow, making it simpler for readers to understand the connections between signals and their corresponding buses.**Signal line connection not cross other lines use 'Line hop'**[22], signal lines that cross in complicated models can create a crowded and aesthetically misleading graphic. Utilizing a symbol like a bridge to make the connections obvious without utilizing overlapping lines, the "Line hop" approach helps avoid these junctions. As a result, the model is presented in a clear and orderly manner, improving readability in general.**Signal lines shall not be split into more than two sub-lines**[22], signal line splitting too often might result in an extremely complex and difficult-to-understand model. The complexity of the model is efficiently controlled by limiting signal lines to divide into two sub-lines at most, making it simpler for readers to comprehend the data flow.**Unconnected signal lines and subsytems shall be removed or terminated**[22], disconnected components might result in mistakes and incorrect interpretations. The danger of possible problems during simulation and implementation is decreased by eliminating disconnected signal lines and subsystems and ensuring an accurate representation of the system. This routine aids in keeping the model succinct and effective.**Orient blocks with output to the right**[22], The model is more coherent and consistent visually when blocks with outputs are arranged consistently to the right. Readers may readily follow the data flow since it follows the defined signal direction, moving from left to right. This configuration also makes it easier to spot system dependencies and behavior.**Adhere to Simulink guidelines for usable characters in lines and block names**[22], Cross-platform compatibility and portability are guaranteed by adhering to Simulink's rules for permitted characters in line and block names. Using just permitted characters reduces the possibility of compatibility problems, enabling seamless sharing and cooperation on models between various people and systems.

In conclusion, the goals of the MathWorks Automotive Advisory Board are to provide well-structured, standardized, and communicable automotive models. By adhering to these recommendations, modelers may increase the productivity and accuracy of their work while ensuring that their models continue to be compatible and simple to share across many contexts and platforms. It is compressed to binaries at the end so its much easier when pushed to gerrit and used to develop further upon when version controlled, so its better to have enforced checks to software developers so its better to have Guidelines to be followed.

Apart from MAAB, When building and maintaining MATLAB models, we must adhere to a set of rules, standards, and best practices known as internal modeling

guidelines. These rules are essential for ensuring that your models are reliable, readable, and consistent. The names of the blocks inside your MATLAB models describe the various systems and subsytems. The consistency, cooperation, and model maintenance are all facilitated by using standardized block name conventions. In the future this can also be an enabler to create a simulink model from the yaml file. Bus name guidelines guarantee that the names are appropriate and adhere to that of the block and have a standard structure. When working with complicated signal flows, clear bus naming helps avoid misunderstanding. The models' structure, including how various components are arranged and connected, is defined by the model hierarchy. Model hierarchy guidelines guarantee that your models are well-organized, modular, and user-friendly. And lastly, MATLAB models' libraries offer a mechanism to exchange and reuse parts amongst several models. The management of library connections is covered, this entails checking that all required libraries are appropriately connected as well as locating and any broken or inoperable library links are filled upon in Non-library links.

3.4 Approach 1

Having a good understanding of the project requirements and following some useful standards to follow was to have code formatting, variable and function naming and comments and many more. All of these improves for better readable, clean, and maintainable code. A thorough workflow is created in the early stages of the development process to effectively manage model information and follow modeling best practices. There are various phases in this process, each of which improves the development process overall effectiveness and dependability. In the first iteration of development loading the appropriate model into the MATLAB environment starts the procedure. Since MATLAB offers a strong framework for dealing with complex systems, this model most certainly depicts a complicated system. After loading and initialising the dSpace dependencies, the emphasis is on obtaining important data from the model. These specifics include the model's hierarchical structure, block names inside sublayers, bus names, bus output line names, and bus names. Understanding the structure and elements of the model clearly requires the extraction procedure.

Especially the `get_param` function, are used to pull data, this function is essential for retrieving different qualities and details on the parts of your models. You may specify the precise data you need for analysis and validation by giving specified parameters to the `get_param` function throughout the extraction process. You may get the "LinkStatus" attribute of blocks using `get_param` to determine the state of library links. A block that has a value of "resolved" is one that has been correctly connected, but values like "broken" or "unresolved" denote problems. Use the `find_system` method to go across the model hierarchy[23]. Using different criteria, such as block type or name, you may use this method to search for blocks or subsystems, function may be used to list and filter block names according to certain requirements. You can look for blocks of a specified kind, in a particular subsystem,

or that match a particular name pattern. The retrieved data is then arranged and saved in a JSON (JavaScript Object Notation) file format. JSON is a compact, legible data format that works well for storing structured data. This data may be easily accessed for further processing and analysis by being saved as JSON files. The process is run ideally in which the MATLAB environment adds another level of control and efficiency by integrating the data extraction from the specified model and dSPACE initialization procedures. With this method, you may not only gather and save important data in JSON format but also add extra dependencies.

A configuration file is created in tandem with the JSON data extraction to specify modeling principles. The configuration file's format is set to YAML. YAML provides for the formulation of guidelines and standards that direct the development process and is human-readable. These rules may address name conventions, structural specifications, and coding standards, among other model-related standards. The skeleton of the configuration file is as below

Listing 3.1: Example of ModelValidatorConfig

```
- Block: ""
  Modelstructure:
    Linked: ""
    NotLinked:
        - "{␣root␣}/Application"
        - "{␣root␣}/Plant"
  Subsystem: ""
    - Block: Application
      Modelstructure:
        Linked: ""
        NotLinked:
          - "{␣root␣}/Application/Brain"
      LibraryLink: ""
      Subsystem:
        - Block: Brain
          Modelstructure:
            Linked:
              - "{␣root␣}/Application/Brain/CoolantControl"
              - "{␣root␣}/Application/Brain/ElectricDrivetrainControl"
            NotLinked: ""
          Subsystem: ""
        - Block: Common
          Modelstructure:
            Linked:
              - "{␣root␣}/Application/Common/Diagnostics"
            NotLinked: ""
          Subsystem: ""
```

This yaml file can be modularised and can be used across various models in a single project and have different configurations for different projects. Adhering to well-established coding standards is not just a suggestion in the field of software development; it is a crucial pillar of developing strong and maintainable code. It facilitates developer collaboration, improves code readability, and maintains consistency between projects. In this context, we have a Python package that strictly adheres to the core Google code standards, aligning itself with industry best practices. However, it is not only about adhering to code standards. We are committed to the complete software development ecosystem. This involves selecting tools and procedures that promote efficient and trouble-free development. Poetry is one such essential component.

Poetry is a Python dependency management and packaging tool that interacts neatly with the development workflow. It makes expressing and maintaining project dependencies easier[24]. Poetry provides an elegant and effective way to manually managing dependencies and their versions. You specify the libraries on which your project depends in a single, easy-to-read file, and Poetry handles the rest. It installs, updates, and maintains these dependencies to ensure that your project always has the necessary libraries in the correct versions[25]. Our Python package exhibits a dedication to current and efficient development processes by using Poetry. It represents our intention to streamline and improve the development process, ensuring that our code is not just well-written but also well-packaged, manageable, and consistently stable.

This python package is then run, once the JSON file is safely in its hands, our Python module begins a path of investigation and validation. It meticulously compares the contents of this JSON file to those of a YAML file. However, this is not a casual comparison; it is a detailed examination against a set of internal modeling principles.These internal modeling rules demonstrate our dedication to best practices and quality assurance. The Python module does not only execute this comparison silently; it does it transparently. It tracks the outcomes and generates a clear and informative report. This log acts as a truth document, a record of whether or not our program adheres to the set rules. In essence, our Python package is the gatekeeper of our quality standards, the sentinel that guarantees our software's internal structure is consistent with our best practices. It symbolizes our commitment to quality assurance by offering openness, accountability, and a plan for ongoing development. It is an essential part of our development process, ensuring the integrity and perfection of our product.

3.4.1 Feedback

A deliberate change in your modeling and validation method may be seen in your choice to forgo the JSON file in favor of employing numerous files to manage the data retrieved from your MATLAB models. Adaptability and constant improvement are critical in the dynamic field of software development. Following several iterative assessments of our development process, our supervisor suggested a major

modification. The modeling and validation process, which has formed the foundation of our workflow, is the subject of this modification. It entails a departure from the traditional practice of organizing the data extracted from our MATLAB models into a single JSON file. The shift from a single JSON file may have broad effects on how we handle, process, and validate data in addition to how we store and maintain it. Making this choice comes with both possibilities and difficulties.

Additionally, a remarkable observation regarding code readability and code coverage is also a result of the iterative evaluation process. It is critical that our code be both understandable and clear. Error rates are lower and development times are sped up when there is less complicated code. It encourages teamwork and makes sure that everyone is able to comprehend and utilize the code efficiently. Contrarily, code coverage indicates how well our code-base is tested. We can be more assured of the dependability and caliber of our software if we do tests that are more thorough. It's an essential component of quality control that strengthens the development process.

We are dedicated to improving the overall efficacy of our development process as we accept these modifications and pay attention to our supervisor's criticism. We are aware that change and adaptation are not indications of weakness but rather of resiliency and a commitment to producing high-caliber software. Our goal is to raise the bar for our coding standards such that our software not only meets industry standards, but continuously surpasses them through the promotion of code readability and increased code coverage. These modifications provide witness to our unshakable dedication to the highest standards of software development and our never-ending search of perfection. This is covered in the section that follows.

3.5 Approach 2

In our pursuit of software excellence, the second iteration of our project was a watershed moment of transition. We proceeded on an intensive path of refining, driven by a commitment to clarity and excellence. One of the key goals was the crystalline clarity of our codebase. We worked hard to improve the readability and comprehensibility of our code, making sure that every line was a testimonial to precision and logic.Concurrently, our dedication to dependability and robustness gained concrete form with the deployment of rigorous unit tests and broad code coverages. Each part of our program was thoroughly examined, confirming its functioning and guaranteeing that it met the highest quality requirements. This rigorous testing strategy not only strengthened our code against possible flaws, but also fostered trust in its dependability.

Before the focus was to shift towards the unittets and code coverage in the thesis work I try to mitigate the multiple file handles. The decision to remove the JSON file from the system extraction procedure was a watershed moment in our development path. In its stead, we used the power of the MATLAB Engine, a Python program [26]. This clever approach expanded our options by allowing us to run MATLAB

environments straight from our Python ecosystem via API calls. To get this system up and running, we needed to setup initialization files and set up paths to libraries, allowing the MATLAB environment to run seamlessly within the Python environment[25]. This was a critical step in ensuring a seamless and connected process. After finishing the setup, we were ready to run MATLAB scripts straight from our Python environment. This connection not only eased the development process but also the data transition. We took a more efficient way than depending on JSON files. When the MATLAB programs were run, they produced results that were recorded and arranged into Python dictionaries. This move simplified file handling, reducing the requirement for intermediary storage.This shift aided our growth process in various ways. It decreased the expense of managing JSON files and lowered the danger of data damage during transitions. The structured data, which was now held within Python dictionaries, flowed effortlessly into our Python programs as parameters, making it more accessible and simple to deal with. Furthermore, this method improved data integrity by guaranteeing that information remained intact and consistent throughout its journey from the MATLAB environment to the Python ecosystem.

As code matures, unit tests act as a safety net, spotting regressions and unwanted side effects. It let's us to test that each element of our program works properly in isolation, guaranteeing that it completes its unique purpose without mistakes. Unit tests, in addition to confirming code correctness, help with code description and maintainability[1]. A well-written unit test serves as a living example of how to utilize a given piece of code, making it easier for developers to comprehend and work with the code. It is to enable to have for all the functions and classes defined by mocking the inter dependecies and having an code coverage aggregate of 70% on a whole, but there are few limitations which will be discussed later session for the drop in the unit test to reach above 100%. Code coverage is a statistic that gauges how much our unit tests work the codebase. It counts the lines of code, branches, and conditions that are run by tests. Understanding the completeness of our testing efforts is dependent on code coverage.High code coverage implies that a large amount of our code has been tested, lowering the possibility of hidden errors. It provides a practical degree of assurance in the dependability of our program. We may detect untested or under-tested sections of our code and target testing efforts appropriately by measuring code coverage. High coverage necessitates careful consideration of edge situations, mistake scenarios, and extraordinary conditions, resulting in more resilient software.

Finally, using a Zuul pipeline matches our development process with contemporary CI/CD best practices. By automating the deployment process, it speeds up the delivery of new features and problem fixes to end users. This means shorter development cycles, faster issue resolution, and a more responsive software development methodology. The addition of a Zuul pipeline in our second iteration illustrates our dedication to efficiency and quality in the development process. It is a huge step in embracing modern development processes, ensuring that our product is delivered in a reliable, timely, and high-quality manner.

4
Results and conclusions

From the uncertainty analysis performed, This section of the master thesis places us at the crossroads of creativity and achievement. This section demonstrates how, after months of hard study, development, and testing, we not only tackled a crucial obstacle, but also plotted a course into a more efficient and dependable future for our area. As we move through this section, we'll look at the real situations created from the research, debate the interpretations, and consider the larger ramifications. Let us go through the facts and the conclusions they create via the lens of critical thinking and scientific curiosity, a journey that not only answers but also generates new questions, fuelling the never-ending quest for knowledge.

4.1 Initial observations

A significant discovery arose during the early stages of our thesis study, when our proposal took shape. We diligently planned a series of actions to assure the flawless startup of our MATLAB engine. specifying startup files, defining accurate library locations, and specifying the key DSPACE needs were all part of this procedure. With this foundation in place, we easily executed our MATLAB code, sending it on a mission to collect critical data from the given model.The Model Configurator YAML file, a major component of our project, played an important role in the unraveling event to act as a place holder for the internal guidelines. This YAML file, which had been meticulously prepared to incorporate our modeling requirements, was put into effect. It performed a thorough dance with the data obtained throughout the MATLAB run, it was a complicated symphony in which the contents of the file were harmonized with the acquired data, shaping the log that would follow us later on.

4.2 Simulation work

The final stages of our software development process ushered in a streamlined and automated workflow, marked by precision and efficiency. Once the meticulously crafted package reached its designated repository, a crucial phase commenced. Developers, armed with the latest version of our software solution, were tasked with running the code. This step was not merely routine; it was a pivotal moment where the theoretical met the practical, and the software's integrity faced real-world scrutiny. In this environment, our software was put to the test. The beauty of our approach lay in its proactive nature. Developers were not merely coding; they were engaging in a preemptive strike against potential errors. By running the code, they

initiated a diagnostic process that adhered closely to our internal guidelines. These guidelines, crafted with meticulous care, served as a shield against errors, ensuring that issues were identified and rectified well before they could infiltrate the core of our project.

The significance of this phase cannot be overstated. By detecting errors early in the development cycle, we preempted potential pitfalls at a stage when corrections were still straightforward. This strategic move not only saved time but also safeguarded the overall stability of our software. It represented a shift from reactive bug-fixing to proactive error prevention, aligning our development process with the industry's best practices. By putting these adjustments into practice, the number of cases of "wrong implementation" 1.1 would be significantly reduced, which would shorten the lead time for creating the HIL model for every iterative software release. Detecting and fixing issues early on reduces the need for significant debugging and troubleshooting once they enter the development process. This improves the overall efficiency of the software development process in addition to speeding it up. As a result, the software aligns with industry best practices and becomes more dependable, robust, and adaptive. This change establishes the groundwork for software development cycles that are more responsive and agile, and it is supported by systematic mistake avoidance. Crucially, this process unfolded seamlessly without the need for manual interaction with MATLAB. The burden of loading MATLAB prior to every check was eliminated, streamlining the workflow and ensuring a swift and uninterrupted development pace. In essence, our software had become an autonomous entity, capable of error detection and rectification without constant human intervention.

A key player in this phase was the Zuul pipeline, our sentinel against flawed code. Triggered automatically whenever a new model required validation, the Zuul pipeline ensured that every line of code adhered to our stringent guidelines. Its vigilance meant that no error, no matter how minor, escaped scrutiny.Our project's documentation is kept up to date and accurate with the most recent software versions thanks to Zuul's ability to draw upon playbooks created for document production with ease. In addition, Zuul expands its functionality to execute playbooks for our Python modules unit tests and code coverage analyses. By executing a series of unit tests to guarantee the stability of our code, these playbooks automate the testing procedure. In-depth code coverage reports are another thing they provide, and they offer insightful information about how thoroughly our code is tested.

Overall, this automated, guideline-driven phase epitomized the pinnacle of our software development journey. It represented a marriage of meticulous planning, innovative automation, and unwavering quality standards. By allowing our software to self-diagnose and correct, we not only ensured the delivery of a robust solution but also set a precedent for future software development endeavors. This phase was more than a process; it was a paradigm shift, transforming how errors are addressed in the realm of software development. Through these automated checks, our software not only met industry standards but exceeded them, emerging as a beacon of quality and efficiency in the ever-evolving landscape of technology.

4.3 Summary

In conclusion, this thesis has not only contributed to the academic conversation, but it has also generated a practical, real-world solution. Our effort has an influence that goes beyond this page, since our software is positioned to bring about beneficial changes in the industry. As our program emerges from the theoretical cocoon of this thesis, it develops its own personality. Its potential for beneficial industrial development is limitless. By tackling a serious issue, we've paved the way for enhanced efficiency, dependability, and creativity. It exemplifies the revolutionary potential of research, which is propelled by unshakable devotion, creative invention, and the synergy of teamwork.

Our comprehensive testing, including unit tests, code coverage assessments, and validation against internal modeling guidelines, has fortified our software's quality and reliability. The introduction of a Zuul pipeline into the development workflow has streamlined our processes, ensuring that code changes are rigorously assessed and seamlessly integrated into the software.

5

Discussions and future scope

5.1 Limitations

A significant limitation that warrants to completely integrate it with the Zuul pipeline is a noteworthy shortcoming that has to be given a lot of consideration in this thesis project. This restriction stems mostly from a fundamental need for our process, which is the utilization of the MATLAB engine via an API rather than the Python environment. But in order to fully integrate this, you will need MATLAB 2021b, which is an upgraded version that guarantees flawless compatibility.The crux of the issue is that the organization's virtual image does not have the necessary version of MATLAB 2021b. As a result, there is a significant backlog in the thesis work since this particular version of MATLAB is required to run our MATLAB engine alongside Python, which allows for smooth automation and error checking.

We are forced to run the process on static nodes, which are effectively our local computers, because this version isn't available. Because of the very inflexible environment this produces, our system cannot function on dynamic computers as it should in the context of the Zuul pipeline. This restriction significantly impairs our workflow's flexibility and dynamic character, which has an effect on the effectiveness, adaptability of our procedures, and full potential of automation and error checking through Zuul. To overcome this restriction, the organization must upgrade its virtual image as soon as possible to include MATLAB 2021b. This upgrade allows us to fully utilize automation and eliminates a significant bottleneck in our workflow while also guaranteeing that our thesis work complies with industry standards and best practices. All things considered, the lack of MATLAB 2021b in the company's virtual environment continues to be a major obstacle, preventing our thesis work from being seamlessly integrated with the Zuul pipeline and limiting the workflow's flexibility. For our research and development processes to reach their maximum potential, action must be made to overcome this barrier.

5.2 Future Research Directions

The future scope of this thesis presents an exciting trajectory for our research, with the potential to revolutionize the way modeling and development are conducted in our field. One of the central objectives on the horizon is the transformation of the YAML file, which currently serves as a reference for our modeling guidelines, into a primary input for generating models. This transition holds the promise of elimi-

nating the dependence on the Simulink environment, a substantial leap forward in terms of reducing issues associated with Simulink libraries and wrong implementations. We achieve new levels of speed, flexibility, and scalability by switching to a model generation process that is driven by the YAML file. Without being limited by the Simulink environment, models may be built, offering a more flexible and modular foundation for development. This guarantees the production of extremely precise models by streamlining the modeling process and lowering the possibility of mistakes and inconsistencies.

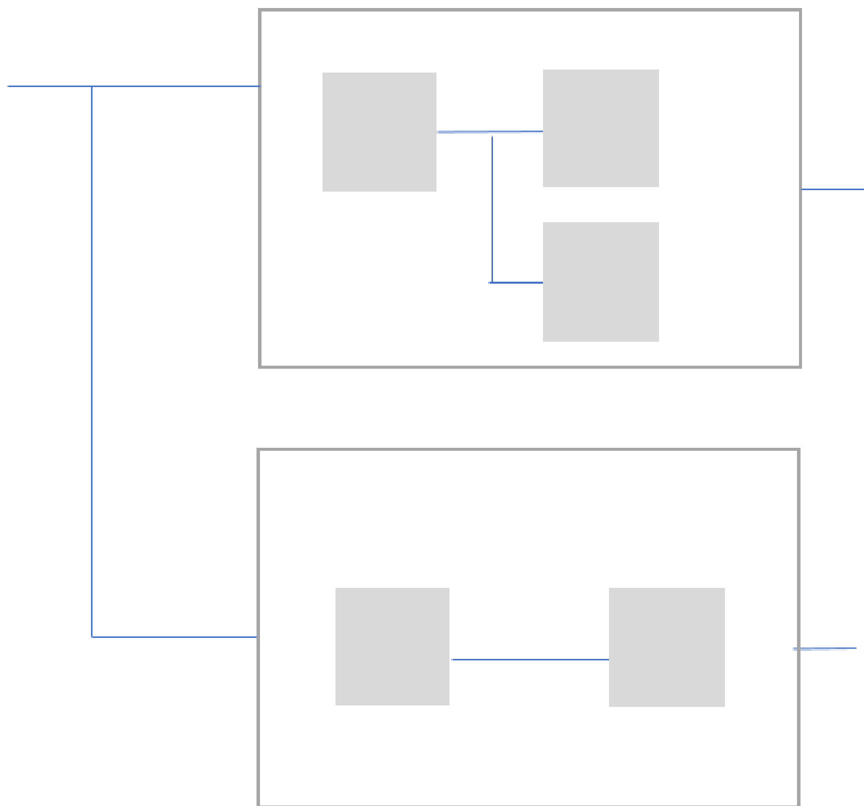

Figure 5.1: Pick and Placing of Complete Simulink Models

Moreover, the research's future scope goes beyond the parameters of the present thesis. Now that the produced models are independent of the Simulink environment, they may be easily included into the Advance Development Kit (ADK). This integration is a crucial step since it makes it possible to create models on demand using an intuitive graphical user interface (GUI). As we can see as an example from the picture 5.1 to pick and place the boxes which imply as subsytem of each model. It is a game-changing idea to be able to make models whenever needed, as long as they follow the project's guidelines and criteria. It is consistent with the ultimate objective of responsive and agile software development, where models are quickly changed and adjusted to meet changing requirements. This method guarantees that the software is always up to date with the most recent project requirements while also greatly speeding up the development process.

The ultimate goal of this thesis is to break free from Simulink's limitations and herald in a new era of precision and efficiency by switching to a YAML-driven model creation process. This makes it possible for a smooth connection with the ADK, where GUI-driven model generation is made possible and models that are exactly suited to the project's requirements may be developed. These advancements have the capacity to completely transform our industry's software development environment, promoting responsiveness, accuracy, and agility.

This effort has an influence that goes beyond the pages of this publication since it has the ability to improve the industry. It shows the ability to overcome obstacles and reimagine traditional procedures, and it is a monument to the strength of research, invention, and teamwork. We are still fully committed to excellence as we look to the future. We recognize that software is an ever-evolving entity, much like knowledge. As a result, we are ready to take on new challenges in the future, keep refining our solution, and lead the way in innovation in our industry. The software solution serves as our beacon as we embark on a new adventure that doesn't stop here.

Bibliography

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston: Addison-Wesley, 2007.

[2] C. C. Low, T. H. Lee, and M. C. Ong, "Automated testing of simulink models for real-time applications," *Journal of Simulink Testing*, vol. 12, no. 4, pp. 123–137, 2017.

[3] R. Kaushik, A. Shukla, and V. Garg, "Automated testing framework for simulink models," *Simulation and Testing Journal*, vol. 15, no. 2, pp. 45–62, 2018.

[4] J. H. Kim, H. D. Kim, and K. H. Lee, "Automated verification and testing of simulink models for real-time embedded systems," *Embedded Systems Journal*, vol. 18, no. 1, pp. 78–92, 2021.

[5] G. A. Salazar and M. F. Hassan, "Automated testing of simulink models using matlab unit testing framework," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 210–225, 2019.

[6] B. L. Welch and K. E. Kinsella, *Model-Based Design for Real-Time Systems: Simulink-Based Automated Testing*. New York: Springer, 2020.

[7] K. Popovici, *Real-Time Simulation Technologies: Principles, Methodologies, and Applications*. CRC Press, 2017.

[8] B. Appleton and S. Konieczka, *Effective Version Control for Agile Development*. 2006.

[9] H. Zhang and Y. Chen, "Hardware-in-the-loop testing of embedded systems using simulink and dspace," *Real-Time Testing Journal*, vol. 14, no. 3, pp. 105–120, 2018.

[10] L. Zamboni, *Getting Started with Simulink*. 2013.

[11] A. Mockus and L. G. Votta, "Version control tools for software maintenance," 2000.

[12] O. N. Computer, "Version control in software engineering: An overview," 2006.

[13] M. J. Muller, *Version Control for Engineers*. 2013.

[14] Jenkins project contributors, "Jenkins documentation." https://www.jenkins.io/doc/, 2011-2023.

[15] Zuul project contributors, "Zuul documentation." https://zuul-ci.org/docs/zuul/latest/index.html, 2012-2023.

[16] M. Soni, *Jenkins Essentials*. Packt Publishing Limited, 2015.

[17] Ansible project contributors, "Ansible best practices." https://docs.ansible.com/ansible/latest/index.htmll, Last updated on Oct 05, 2023.

[18] J. Loeliger and M. McCullough, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. 2012.

[19] J. Geerling, *Ansible for DevOps*. Self-published (available on Leanpub), 2015.
[20] N. Gehani, S. Suresh, and V. Thiagarajan, "Automating application deployments with ansible," Publication date may vary, refer to the IBM Developer website.
[21] R. Somasundaram and A. Kumar, *Git: Version Control for Everyone*. 2013.
[22] The MathWorks, Inc, "Matlab maab modeling guidelines." https://se.mathworks.com/solutions/mab-guidelines.html, 1994-2023.
[23] S. Attaway, *MATLAB: A Practical Introduction to Programming and Problem Solving*. 2009.
[24] Python Poetry Contributors, "Python poetry documentation." https://python-poetry.org/docs/, Last updated on 2023.
[25] M. Dawson, *Python Programming for the Absolute Beginner*. Cengage Learning, 2010.
[26] MathWorks, "Matlab engine for python documentation." https://se.mathworks.com/help/matlab/matlab_external/get-started-with-matlab-engine-for-python.html, Last accessed on 2023.

A

Appendix: Source Code Implementation

A.1 MATLAB Code Synopsis

```
function extract = model_extract (mdl_name)

    load_system(mdl_name)
    json_data = {};
    extract = struct('Model', [], 'SysPath', [], 'LibLinkPath', [], 'Bus_Path', [], 'Bus_BlockName', [], 'Bus_LineName', []);

    sys_info_paths = find_system(mdl_name,'SearchDepth',3, 'BlockType','SubSystem');
    library_links = find_system(mdl_name,'SearchDepth',3,'FollowLinks','on','LookUnderMasks','on','LinkStatus','resolved');
    for i=1:length(library_links) % inactive the library links for getting BusName and BusPath
        set_param(library_links{i,1},'LinkStatus','inactive')
    end

    bus_connector = find_system(mdl_name,'SearchDepth',4,'BlockType','BusCreator');
    line_handles = get_param(bus_connector,'LineHandles');
    % Replace the newline to '_' for readablity
    bus_connector = replace( bus_connector,char(10),'_');

    for i = 1:length(line_handles)
        % Get Source Block information from Bus Creator
        srcinfo.srcBlock{i,1} = get_param(line_handles{i}.Inport, 'SrcBlock');
        % Get Inport Name information to the Bus Creator
```

```
22          srcinfo.InportName{i,1} = get_param(line_handles{
               i}.Inport, 'Name');
23          if iscell(srcinfo.srcBlock{i})
24              for j=1:numel(srcinfo.srcBlock{i,1})
25                  % Get Bus Path information for respective
                        Bus Creator
26                  srcinfo.buspath{i,1}{j,1} = bus_connector
                       {i};
27              end
28              if cellfun(@(x) contains(x, 'Bus'), srcinfo.
                   srcBlock{i})
29                      srcinfo.srcBlock{i,1} = srcinfo.
                           InportName{i,1};
30              end
31          else
32              srcinfo.buspath{i,1} = bus_connector{i};
33              srcinfo.srcBlock{i,1} = {srcinfo.InportName{i
                   ,1}};
34          end
35      end
36      bus.Path = concatenateCellArray(srcinfo.buspath);
37      bus.Block = concatenateCellArray(srcinfo.srcBlock);
38      bus.Name = concatenateCellArray(srcinfo.InportName);
39      disp('')
40      extract.Model = mdl_name;
41      extract.SysPath = sys_info_paths;
42      extract.LibLinkPath = library_links;
43      extract.Bus_Path = bus.Path;
44      extract.Bus_BlockName = bus.Block;
45      extract.Bus_LineName = bus.Name;
46      j_data = jsonencode(containers.Map({'AModel','SysPath
           ','LibLinkPath','Bus_Path','Bus_BlockName','
           Bus_LineName'},{mdl_name,sys_info_paths,
           library_links,bus.Path,bus.Block,bus.Name}),
           PrettyPrint=true);
47      json_data{end+1} = j_data;
48      for i=1:length(library_links) % restore the library
           links
49          set_param(library_links{i,1},'LinkStatus','
               restore')
50      end
51      % Clear the struct variables
52      srcinfo = structfun(@(x) [], srcinfo, 'UniformOutput'
           , false);
53      bus = structfun(@(x) [], bus, 'UniformOutput', false)
           ;
```

```
54
55      bdclose('all')
56      % Concatenate all the JSON data in the cell array
57      json_all = ['[', strjoin(json_data, ','), ']'];
58      fileID = fopen('Model_extract_checks.json','w');
59      fprintf(fileID, '%s\n', json_all);
60      fclose(fileID);
61      %% Check cockpit revision
62  %     fprintf('--- Checking For Python File ---\n')
63  %     init_python_packages_path();
64  %     check_files = py.importlib.import_module('ci_check.
    check_files')
65  %     py.importlib.reload(check_files);
66  %     check_files.read_file()
67  end
68
69  function concatenated = concatenateCellArray(cellArray)
70      % Initialize an empty cell array for concatenation
71      concatenated = {};
72
73      % Loop through each element in the cell array
74      for i = 1:numel(cellArray)
75          % Check if the element is a cell
76          if iscell(cellArray{i})
77              % Get the inner elements of the cell
78              innerElements = cellArray{i};
79
80              % Loop through each inner element
81              for j = 1:numel(innerElements)
82                  % Check if the inner element is empty
83                  if isempty(innerElements{j})
84                      % If it is empty, fill in 'No Line
                          Name'
85                      concatenated = [concatenated, 'Empty␣
                          Name'];
86                  else
87                      % Otherwise, append the inner element
88                      concatenated = [concatenated,
                          innerElements{j}];
89                  end
90              end
91          else
92              if isempty(cellArray{i})
93                  cellArray{i} = 'Empty␣Name';
94              end
95              % If it is not a cell, simply append it
```

```
            concatenated = [concatenated, cellArray{i}];
        end
    end
    concatenated = concatenated';
end
```

Listing A.1: MATLAB Code Example

A.2 Model Configuration YAML

```
- Block: ""
  Modelstructure:
    Linked: ""
    NotLinked:
        - "{␣root␣}/Application"
        - "{␣root␣}/Plant"
  Subsystem:
    - Block: Application
      Modelstructure:
        Linked: ""
        NotLinked:
          - "{␣root␣}/Application/Brain"
          - "{␣root␣}/Application/Common"
          - "{␣root␣}/Application/Component"
      LibraryLink: ""
      Subsystem:
        - Block: Brain
          Modelstructure:
            Linked:
              - "{␣root␣}/Application/Brain/
                 CoolantControl"
              - "{␣root␣}/Application/Brain/
                 ElectricDrivetrainControl"
              - "{␣root␣}/Application/Brain/
                 ElectricalEnergyCoordinator"
              - "{␣root␣}/Application/Brain/
                 FrictionBrakeControl"
              - "{␣root␣}/Application/Brain/
                 HighVoltageEnergyControl"
              - "{␣root␣}/Application/Brain/
                 MovementEnabler"
              - "{␣root␣}/Application/Brain/StartManager"
              - "{␣root␣}/Application/Brain/
                 ThermalEnergyCoordinator"
              - "{␣root␣}/Application/Brain/
                 VehicleMotionAndRoadStateEstimation"
```

```
29         - "{␣root␣}/Application/Brain/VehicleMotionCoordination"
30         - "{␣root␣}/Application/Brain/VehicleStabilityControl"
31       NotLinked: ""
32     Subsystem: ""
33   - Block: Common
34     Modelstructure:
35       Linked:
36         - "{␣root␣}/Application/Common/Diagnostics"
37         - "{␣root␣}/Application/Common/NetworkManagement"
38         - "{␣root␣}/Application/Common/Time"
39         - "{␣root␣}/Application/Common/VehicleMode"
40       NotLinked: ""
41     Subsystem: ""
42   - Block: Component
43     Modelstructure:
44       Linked:
45         - "{␣root␣}/Application/Component/CrashControl"
46         - "{␣root␣}/Application/Component/DriverControls"
47         - "{␣root␣}/Application/Component/EFAD"
48         - "{␣root␣}/Application/Component/ERAD"
49         - "{␣root␣}/Application/Component/GearBox"
50         - "{␣root␣}/Application/Component/HighVoltageBattery"
51         - "{␣root␣}/Application/Component/LowVoltageBattery"
52         - "{␣root␣}/Application/Component/ThermalManagement"
53       NotLinked: ""
54     Subsystem: ""
55 - Block: Plant
56   Modelstructure:
57     Linked: ""
58     NotLinked:
59       - "{␣root␣}/Plant/Driveline"
60       - "{␣root␣}/Plant/External"
61       - "{␣root␣}/Plant/HvSystem"
62       - "{␣root␣}/Plant/LvSystem"
63       - "{␣root␣}/Plant/Vehicle"
64   Subsystem:
65     - Block: Driveline
66       Modelstructure:
```

```
        Linked: ""
        NotLinked:
          - "{␣root␣}/Plant/Driveline/DogClutch"
          - "{␣root␣}/Plant/Driveline/Engine"
          - "{␣root␣}/Plant/Driveline/FrontMachine"
          - "{␣root␣}/Plant/Driveline/ParkLock"
          - "{␣root␣}/Plant/Driveline/RearMachine"
      Subsystem: ""
    - Block: External
      Modelstructure:
        Linked: ""
        NotLinked:
          - "{␣root␣}/Plant/External/CcsChargingPole"
          - "{␣root␣}/Plant/External/ChademoChargingPole"
          - "{␣root␣}/Plant/External/GbtChargingPole"
          - "{␣root␣}/Plant/External/Grid"
      LibraryLink: ""
      Subsystem: ""
    - Block: HvSystem
      Modelstructure:
        Linked: ""
        NotLinked:
          - "{␣root␣}/Plant/HvSystem/ChargingInlet"
          - "{␣root␣}/Plant/HvSystem/DcLink"
          - "{␣root␣}/Plant/HvSystem/DcdcConverter"
          - "{␣root␣}/Plant/HvSystem/Elac"
          - "{␣root␣}/Plant/HvSystem/FrontInverter"
          - "{␣root␣}/Plant/HvSystem/HvBattery"
          - "{␣root␣}/Plant/HvSystem/Hvch"
          - "{␣root␣}/Plant/HvSystem/OnBoardCharger"
          - "{␣root␣}/Plant/HvSystem/RearInverter"
      Subsystem: ""
    - Block: LvSystem
      Modelstructure:
        Linked: ""
        NotLinked:
          - "{␣root␣}/Plant/LvSystem"
          - "{␣root␣}/Plant/LvSystem/DcLink"
          - "{␣root␣}/Plant/LvSystem/LvBattery"
          - "{␣root␣}/Plant/LvSystem/Relays"
      Subsystem: ""
    - Block: Vehicle
      Modelstructure:
        Linked: ""
        NotLinked:
```

```
112                 - "{␣root␣}/Plant/Vehicle/CarCockpit"
113                 - "{␣root␣}/Plant/Vehicle/Chassis"
114             Subsystem: ""
```

Listing A.2: YAML Configuration Example

A.3 Python Code Synopsis

```
1  import sys
2  import pathlib
3  import yaml
4  import matlab.engine
5  import os
6  import json
7  import psutil
8  import glob
9  import time
10
11 sys.path.insert(0, f'{pathlib.Path(__file__).parent.
      parent.parent}')
12 import init_python_paths
13 init_python_paths.add_children_to_path()
14 FRAMEWORK_REPO_DIR, INTEGRATION_REPO_DIR =
      init_python_paths.get_hilmdl_repo_dirs()
15
16
17 def run_matlab_engine():
18     """
19     To initilise matlab eninge add paths and  API for
          Python provides a package to
20     integrate MATLAB functionality directly with a Python
           application, creating an interface to call
          functions
21     from your MATLAB installation from Python code.
22     """
23     model_path = os.path.abspath(f"{INTEGRATION_REPO_DIR
          }\\projects\\GPA\\IHxA\\E2\\sclx")
24     ci_check_path = os.path.abspath(f"{FRAMEWORK_REPO_DIR
          }\\tools\\python\\ci-check\\ci_check")
25
26     PATHS_TO_ADD = [
27                     INTEGRATION_REPO_DIR,
                          FRAMEWORK_REPO_DIR, model_path,
                          ci_check_path,
28                     os.path.join(FRAMEWORK_REPO_DIR, '
                          callbacks'),
```

```
29                          os.path.join(FRAMEWORK_REPO_DIR, '
                                classes'),
30                          os.path.join(INTEGRATION_REPO_DIR, '
                                libraries'),
31                          os.path.join(FRAMEWORK_REPO_DIR, '
                                tools'),
32                          os.path.join(FRAMEWORK_REPO_DIR, '
                                tools', 'python'),
33                          os.path.join(FRAMEWORK_REPO_DIR, '
                                tools', 'ModelPortBlocks'),
34                          os.path.join(FRAMEWORK_REPO_DIR, '
                                tools', 'yamlparser'),
35                          os.path.join(INTEGRATION_REPO_DIR, '
                                plant', 'Common', 'tools')]
36
37      # Clean up before starting matlab engine
38      for proc in psutil.process_iter():
39          # check whether the process name matches
40          if 'matlab' in proc.name().lower() or '
                configurationdesk' in proc.name().lower():
41              proc.kill()
42      print('Done Config')
43      try:
44          # TODO : Fix once py version is resolved with
                matlabengine
45          eng = matlab.engine.start_matlab('-nosplash -sd
                "{}" -logfile output.log'.format(model_path))
                # noqa : F821
46      except Exception:
47          print('Failed to start matlab engine, trying
                again...')
48          time.sleep(5)
49          # TODO : Fix once py version is resolved with
                matlabengine
50          eng = matlab.engine.start_matlab('-nosplash -sd
                "{}" -logfile output.log'.format(model_path))
                # noqa : F821
51
52      print('Lauch Matlab')
53      eng.slCharacterEncoding('Windows-1252', nargout=0)
54
55      for path in PATHS_TO_ADD:
56          eng.addpath(path)
57
58      files = get_mdl_in_directory(model_path)
59
```

```
60      print("Launching 'model_extract.m' from {}".format(
            ci_check_path))
61      eng.init_path_GPA_IHxA(nargout=0)
62      eng.init_data_GPA_IHxA(nargout=0)
63      # collection of slx and mdl in directory
64      model_extracts = {}
65      for file in files:
66          if file.endswith('.mdl'):
67              base_name = os.path.basename(file)
68              mdl_name = os.path.splitext(base_name)[0]
69          elif file.endswith('.slx'):
70              base_name = os.path.basename(file)
71              mdl_name = os.path.splitext(base_name)[0]
72          # # Calling .m script
73          print("Calling 'model_extract.m' \n")
74          extract = eng.model_extract(mdl_name, nargout=1)
75          model_extracts.update({file: extract})
76
77      print('--------------- MATLAB LOG 
            --------------------:\n')
78      print('\n')
79      f = open('output.log', 'r')
80      file_content = f.read()
81      print(file_content)
82      f.close()
83
84      eng.exit()
85
86      return model_extracts
87
88
89  def parse_model_extract(confg_file, extract_info):
90      """
91      To initilise the report and generate the list of
            fault from the entire model
92      and traverse through the elements from the
            ModelValidatorConfig.yaml file and
93      pass through each of the the 'Block' Element. The
            Block elemenent determines
94      if there exist a sub layer in .yaml file
95      """
96      # Initilise Fault Report
97      report = {'Modelstructure': [], 'LibraryLinks': [], '
            BusPath': [], 'BusBlock': []}
98
99      for block in confg_file:
```

```
100            process_block(block, report, extract_info)
101
102        return report
103
104
105    def get_mdl_in_directory(directory):
106        """
107        To fetch the list of .slx or .mdl in the directory
108        """
109        files = []
110        slx_files = glob.glob(directory + "/*.slx")
111        mdl_files = glob.glob(directory + "/*.mdl")
112        files.extend(slx_files)
113        files.extend(mdl_files)
114
115        return files
116
117
118    def process_block(block, report, extract_info):
119        """
120        To traverse through the elements from the
              ModelValidatorConfig.yaml file,
121        and append each of the faulty Names/LibraryLink/
              ModelStructure/Busnames to the report
122        when compared against the matlab-engine dict variable
               of the models
123        """
124        block_name = block['Block']  # noqa: F841
125        model_structure = block['Modelstructure']
126        lib_link = block['Modelstructure']['Linked']
127        sub_systems = block.get('Subsystem')
128
129        # Append the Linked and NotLinked to one complete
              list for model structures
130        list_model_structure = []
131        for linked in model_structure['Linked']:
132            list_model_structure.append(linked)
133        for notlinked in model_structure['NotLinked']:
134            list_model_structure.append(notlinked)
135
136        for bus_struct in list_model_structure:
137            buspath, busblock = bus_check(bus_struct,
                  extract_info)
138            report['BusPath'].append(buspath)
139            report['BusBlock'].append(busblock)
140
```

```
141     # Loop over Model Hirerachy and Check for the
           structure
142     for model in list_model_structure:
143         check = model_struct_check(model, extract_info)
144         report['Modelstructure'].append(check)
145
146     # Loop over Library Links and Check for Disabled/
           Broken Libraries
147     for links in lib_link:
148         check = lib_link_check(links, extract_info)
149         report['LibraryLinks'].append(check)
150
151     # Loop over Subsytems
152     for sub_sys in sub_systems:
153         if 'Block' in sub_sys:
154             # Process the sub-block
155             # Recursive call to itterate inside subsytems
                   if 'Block' present
156             process_block(sub_sys, report, extract_info)
157
158
159 def bus_check(value, extract_info):
160     """
161     To check if the Block Name and Line Name is the same
           from the extracted info from the model through
162     matlab engine
163     """
164     faulty_struct_path = []
165     faulty_struct_block = []
166     for file_path, info in extract_info.items():
167         flag = False
168         bus_struct = value.replace('{␣root␣}', str(
               extract_info[file_path]['Model']))
169         rep_bus_string = bus_struct.rsplit('/', 1)[0] + '
               /Bus_Creator'
170         bus_name = bus_struct.rsplit('/', 1)[-1]
171         for index, extract_bus_path in enumerate(
               extract_info[file_path]['Bus_Path']):
172             if rep_bus_string in extract_bus_path:
173                 # check if element same as Bus_BlockName
                       and Bus_LineName
174                 if extract_info[file_path]['Bus_BlockName
                       '][index] == extract_info[file_path]['
                       Bus_LineName'][index]:
175                     # if above condition is True check
                           bus name, path and block in
```

```
                        accordance with config_file (.yaml
                        )
176                  if extract_info[file_path]['
                        Bus_BlockName'][index] == bus_name
                        :
177                      flag = True
178                      break
179         if not flag:
180             faulty_struct_path.append(bus_struct)
181             faulty_struct_block.append(bus_name)
182 
183     return faulty_struct_path, faulty_struct_block
184 
185 
186 def lib_link_check(value, extract_info):
187     """
188     To validate if the Library links are available as
           mentioned in .yaml file
189     and is the same from the extracted info from the
           model through
190     matlab engine
191     """
192     faulty_struct = []
193     for file_path, info in extract_info.items():
194         rep_lib_link = value.replace('{ root }', str(
               extract_info[file_path]['Model']))
195         flag = False
196         for liblink in extract_info[file_path]['
               LibLinkPath']:
197             if rep_lib_link == liblink:
198                 flag = True
199                 break  # Exit the loop if a match is
                       found in either of the models .slx
200         if not flag:
201             faulty_struct.append(rep_lib_link)
202     # Return Missmatched Links from YAML file
203     return faulty_struct
204 
205 
206 def model_struct_check(value, extract_info):
207     """
208     To validate if the Model structure are available as
           mentioned in .yaml file
209     and is the same from the extracted info from the
           model through
210     matlab engine
```

```
211     """
212     faulty_struct = []
213     for file_path, info in extract_info.items():
214         rep_model_struct = value.replace('{␣root␣}', str(
               extract_info[file_path]['Model']))
215         flag = False
216         for SysPath in extract_info[file_path]['SysPath'
               ]:
217             if rep_model_struct == SysPath:
218                 flag = True
219                 break  # Exit the loop if a match is
                       found in either of the models .slx
220         if not flag:
221             faulty_struct.append(rep_model_struct)
222     # Return Missmatched Structure from YAML file
223     return faulty_struct
224
225
226 def process_report(report, extract_info):
227     """
228     To itterate over each element in the error logs
229     """
230     logs = {'Modelstructure': [], 'LibraryLinks': [], '
           BusPath': [], 'BusBlock': []}   # Initilise Fault
           Report
231     for item in report:
232         logs[item] = itterate_process_report(item, report
               , extract_info)
233     return logs
234
235
236 def itterate_process_report(item, report, extract_info):
237     """
238     To consolidated the error logs from all elements and
           check if the error exist
239     """
240     processreport = []
241
242     def process_faults(faults):
243         for fault in faults:
244             # check if fault is type list --- [[master.
                   slx],[master1.slx],[master2.slx]]
245             if isinstance(fault, list):
246                 process_faults(fault)
247             # check if faults is more than in one model
                   --- [[master.slx]] | [[master.slx],[
```

```
                    master1.slx],[master2.slx]]
248                 elif len(faults) == len(extract_info):
249                     processreport.append(faults)
250                     break
251 
252         for faults in report[item]:
253             process_faults(faults)
254 
255         return processreport
256 
257 
258 def generated_report(ci_report):
259     """
260     To list the error logs from all elements and print
           them accordingly against each element
261     """
262     for item, value in ci_report.items():
263         if 'BusPath' in item:
264             print("\nBelow are the BusPath and BusBlocks 
                   which are faulty with LineNames or 
                   BlockName\n")
265             print("
                   ----------------------------------------------------
                   ")
266             bus_path_list = [bus_path for bus_path in
                   value]
267             bus_block_list = [bus_block for bus_block in
                   ci_report['BusBlock']]
268             for bus_path, bus_block in zip(bus_path_list,
                    bus_block_list):
269                 print("BusPath {} and its respective 
                       BusBlock is {}".format(bus_path,
                       bus_block))
270             break
271         else:
272             print("\nBelow are the {} which are faulty or
                    needs fix\n".format(item))
273             print("
                   ----------------------------------------------------
                   ")
274             items = [item for nested_list in value for
                   item in nested_list]
275             print('\n'.join(items))
276 
277 def main():
```

```
278     # # Reading the Config files in sections of each Main
           Block ( Application/Plant/IO/HIL )
279     # confg_file_path = os.path.abspath(
280     #                     f"{FRAMEWORK_REPO_DIR}\\tools\
          python\ci-check\ci_check\check_application_layer.
          yaml")
281     confg_file_path = os.path.abspath(os.path.join(os.
          path.dirname(os.path.realpath(__file__)),
282                                                   "
                                                      ModelValidatorConfig
                                                      .yaml"))
283     # Opening YAML file
284     with open(confg_file_path, "r") as confg_file:
285         confg_file = yaml.safe_load(confg_file)
286     # Call the function to run matlab engine
287     extract_info = run_matlab_engine()
288 
289     report = parse_model_extract(confg_file, extract_info
          )
290     error_report = process_report(report, extract_info)
          # Process the Report
291     # Save the error log as a json file for redablity
292     ci_output_log_path = os.path.abspath(os.path.join(os.
          path.dirname(os.path.realpath(__file__)),
293                                                     "ci_output_log.
                                                         json"))
294     with open(ci_output_log_path, 'w') as file:
295         json.dump(error_report, file)
296     # Generate CI-Report
297     generated_report(error_report)
298     return 0
299 
300 # Calling the main function
301 if __name__ == "__main__":
302     main()
```

Listing A.3: Python Code Example

DEPARTMENT OF MECHANICS AND MARINETIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se