DEGREE PROJECT IN COMMUNICATION SYSTEMS, SECOND LEVEL
STOCKHOLM, SWEDEN 2015

Investigation of a new integration test environment

Facilitating offline debugging of Hardware-in-the-Loop

DEKUN YANG

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

Investigation of a new integration test environment

Facilitating offline debugging of Hardware-in-the-Loop

Dekun Yang

2015-11-17

Master's Thesis

Examiner and Academic adviser
Gerald Q. Maguire Jr.

Industrial adviser
Thomas Gustafsson

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

Advanced automatic testing is very important in development and research within the vehicle industry. Hardware-in-the-loop (HIL) systems give the ability to validate Electronic Control Units (ECUs) based on software simulation ***without*** gathering all of the physical hardware. This enables testing by providing inputs and examining the corresponding outputs of the ECUs in a simpler and safer way than in traditional physical testing. HIL offers the advantage that we can verify and validate the functions of ECUs prior to full-scale hardware production.

On the contrary, because HIL systems are normally released as general-purpose test beds, it takes time to embed them into the current system. Additionally, the question of how to fill the gap between the HIL and the test environment is even more critical when the test bed is expected to be used for a long period of time without modifications. Furthermore, HIL systems are precious. It is not practical and will be considered as a waste of resource if it is used exclusively by testers. Scania's RESI group uses Client-Server architecture to make it more flexible. The HIL system is hosted at server side while the testers operate it at client side. This architecture enables different implementations of client and server as long as a same protocol is applied, but this still does not solve the problem that the HIL is not always accessible when the testers want to debug their scripts. The testers want to find a solution to achieve this goal offline (without servers).

To solve the problem, we first investigated which programming languages are used in the industry. Without doubt, there is no dominant language that ideally suits all situations, so secondly, we developed a new test environment. The new environment including "Dummy Mode" and "Mat Mode" is able to provide script validation service on basic and logic levels without servers. The result shows the Dummy mode is able to reach a higher detection rate (99.3%) on simple errors comparing to the current environment (81.3%). By reproducing and reusing the result of HIL system, Mat mode is able to identify logic errors and provide better assistance when the logic errors are found. In general, the proposed environment is able to show a better way of using HIL which makes the whole system more efficient and productive.

Keywords

Hardware in the Loop, test environment, Python, declarative test script, imperative test script, Simulink, MATLAB

Sammanfattning

I fordonsindustrin ställs stora krav på avancerad automatiserad testning. För att utvärdera Electronic Control Units (ECUs) används så kallade Hardware-In-the-Loop-system (HIL) för att simulera den omkringliggande hårdvaran. Detta möjliggör enklare samt säkrare testning av ECU-komponenterna än vid traditionell fysisk testning. Med hjälp av HIL kan ECUs testas innan en fullskalig produktion sätts igång. Då HIL-system vanligtvis utvecklas för ett brett användningsområde kan det ta tid att skräddarsy dem för ett specifikt system. Ett annat viktigt problem vi ställs inför är skillnaderna mellan HIL-systemet och testmiljön, då testfallen förväntas att användas en längre tid utan förändringar. Vidare är HIL-system kostsamma. Det anses vara varken praktiskt eller ekonomiskt att låta HIL-system enbart användas av testare.

Scanias RESI-grupp använder en klient-server-arkitektur för att åstadkomma flexibilitet HIL-systemet körs på serversidan medan testarna arbetar på klientsidan. Den här typen av arkitektur öppnar upp för olika implementationer på klient- samt serversida, förutsatt att samma kommunikationsprotokoll används. En nackdel med den nuvarande lösningen är att HIL-systemet inte alltid finns tillgängligt när testarna vill felsöka deras programskript. Testarna vill hitta en lösning där det går att utföra felsökningen lokalt, utan tillgång till servrar.

För att kunna lösa problemet undersöktes först vilka programmeringsspråk som används inom industrin. Undersökningen visar på att det finns inget programmeringsspråk som är idealt för alla ändamål. Vidare utvecklades en ny testmiljö som tillhandahåller testlägena "Dummy Mode" samt "Mat Mode". Testmiljön kan användas för att validera programskript på grund- och logiknivå utan att kommunicera mot servrar. Resultatet visar att "Dummy Mode" detekterar upp till 99.3% av enklare typ av fel än motsvarande 81.3% i nuvarande testmiljön. Genom att reproducera och återanvända resultat av HIL-systemet kan “Mat Mode” identifiera logikfel samt ge en bättre indikation om vad felen innebär. Generellt sätt kan den föreslagna testmiljön visa på ett bättre användande av HIL, som gör hela systemet mer effektivt och produktivt.

Nyckelord

Hardware In the Loop, Testmiljö, Python, Deklarativ programmering, Imperativ programmering, Simulink, MATLAB

Acknowledgments

I would like to thank my wonderful supervisor at Scania Thomas Gustafsson for his great support and valuable experience with all my questions.

From KTH Royal Institute of Technology, I have received innumerous help and suggestions from professor Gerald Q. Maguire Jr. I could not have finished this work without you. A thousand thanks.

Thanks to my friend Mattias Appelgren and Eddie Kämpe for the translation of the Abstract.

My beautiful girlfriend Ying Cai has provided great support to me during these six months. Thank you so much!

Stockholm, November 2015
Dekun Yang

Table of contents

Abstract i
Keywords i
Sammanfattning iii
Nyckelord iii
Acknowledgments v
Table of contents vii
List of Figures ix
List of Tables xi
List of acronyms and abbreviations xiii
1 Introduction 1
1.1 Background 1
1.2 Problem definition 2
1.3 Purpose 2
1.4 Goals 2
1.5 Delimitations 2
1.6 Structure of the thesis 2
2 Background 5
2.1 Evaluation of Language: Why is Python used as well? 5
2.1.1 Language Evaluation Criteria 5
2.1.2 Current Testing Environment 6
2.1.3 Conclusion 6
2.2 Test Environment 6
2.3 Symbolic execution 8
2.4 Brief introduction to test scripts 9
2.5 Independent Guarded Assertions 9
2.6 MAT-files 11
2.7 HIL in Scania 12
3 Method, methodology, and tools 13
3.1 Feedback meetings 13
3.2 Priority checklist 13
3.3 Case study and literature review 14
3.4 Architecture and algorithm design 15
3.5 Software tools 17
4 Implementation 19
4.1 Dummy Mode 19
4.1.1 Algorithm and Implementation 19
4.1.2 Optimizations 21
4.1.3 Analysis and Validation 23
4.1.4 Why not Symbolic execution 24
4.2 Mat Mode 24
4.2.1 Algorithm and Implementation 24

4.2.2 Analysis 25
4.2.3 Graphical User Interface (GUI) 27
4.2.4 A real case 30
5 Evaluation 31
5.1 Offline debugging 31
5.1.1 Run time 32
5.1.2 Error Detection Rate 33
5.1.3 Ease of use: when an error is detected........................... 34
5.1.4 Ease of use: debugging with GUI in Mat mode 35
5.2 Efficient static analysis tool 39
5.3 Be able to run automatically 40
6 Conclusions and Future work 41
6.1 Conclusions 41
6.2 Limitations 41
6.3 Future work 42
6.4 Reflections 42
References 45
Appendix A: Detailed results 47

List of Figures

Figure 1-1: Illustration of HIL Simulation 1
Figure 2-1: Work flow in our Client-Server architecture 7
Figure 2-2: Communication between Client and Servers 7
Figure 2-3: Structure of a test script in RESI 9
Figure 2-4: Matching of a course and a script 11
Figure 2-5: MATLAB level 5 MAT-file format 12
Figure 4-1: First Run of execution program 19
Figure 4-2: After the first trim program (second run of execution program) 20
Figure 4-3: Branch removed 20
Figure 4-4: New Branch created 21
Figure 4-5: Gather information and exceptions from the running program 21
Figure 4-6: Same Operation in two Scripts 26
Figure 4-7: General view of GUI, Mat mode 29
Figure 4-8: Zoom in 29
Figure 4-9: Plot with conflicts 30
Figure 5-1: Multiple curves plotted in the new environment 35
Figure 5-2: Multiple curves plotted in the old environment 36
Figure 5-3: Zoom in at a specific area-new environment 37
Figure 5-4: Zoom in to a specific area-old environment 37
Figure 5-5: Zoom in-new environment 38
Figure 5-6: Zoom in-old environment 38
Figure 5-7: Coefficient setting-old environment 39
Figure 6-1: Trigger sequence 42
Figure 6-2: Future continuous integration testing 42

List of Tables

Table 2-1: List of functions 10
Table 3-1: List of priorities (ordered by priority) 13
Table 5-1: Exceptions..32
Table 5-2: Running time comparison: HIL hardware and Dummy mode...33
Table 5-3: Errors Detected..34

List of acronyms and abbreviations

API	application programming interface
CAN	Controller Area Network
CI	Continuous Integration
COO	Coordinator
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
EMS	Engine Management System
EES	electronic error simulation
FIU	failure injection unit
GMS	gearbox management system
GPIB	general purpose interface bus
GUI	Graphical User Interface
HIL	Hardware-in-the-loop
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
I/O	Input/Output
LIN	local interconnect unit
MIL	Model-in-the-loop
RESI	Vehicle Electrical Integration and Chassis System Software
Scipy	Open Source Library of Scientific Tools
SESAMM	Scania Electrical System Architecture for Modularization and Maintenance
SIL	Software-in-the-loop
SOPS	Scania Onboard Product Specification
SUT	System Under Test
VEHIL	Vehicle Hardware-in-the-loop

1 Introduction

Vehicles are expected to always be more reliable and intelligent due to advanced and complicated systems, and at the same time vehicle manufacturers require faster and more efficient production and delivery. In order to achieve these objectives, an optimized automatic test environment is of great importance to the whole development process. This thesis aims to explain Scania's current automatic test environment and give a set of solutions that as a whole will improve the speed and efficiency of this testing.

1.1 Background

Scania uses MathWorks® MATLAB®/Simulink® [1] to model the advanced control system and Hardware-in-the-loop (HIL) to perform integration testing. HIL is a combination of software and hardware which helps to perform testing of embedded systems while achieving low cost, a repeatable test procedure, and high usability in a safer environment that traditional testing [2]. HIL is also used to perform tests that would be hard or very dangerous to test in a real vehicle. Furthermore, in Scania, HIL is used to complement real tests in vehicles in order to cover the large variation space due to many options that are available when configuring a specific instance of a vehicle.

In the HIL environment, components under tested believe that they are placed into a real environment, but they are actually connected with various signal sources that send exactly the same signals as the corresponding real component. Computers, instead of a physical plant (engine, brakes, and vehicle dynamics), feeds the stimulated signals to the object(s) under test [3].

Many development procedures can benefit from this HIL pattern. Function tests can be done at an earlier stage, thus accelerating the maturity of the products; especially when the product depends upon other hardware or software that has not yet been brought into existence. Reactions taken by Electronic Control Units (ECUs) of failures or dangerous situations can also be easily done at a lower cost in terms of money and time than when using traditional testing. Most importantly, HIL has the ability to automate all of these test cases. With an appropriate test configuration, testing can run 24 hours a day without human interaction [4]. See the illustration in Figure 1-1 of HIL being used to test an ECU.

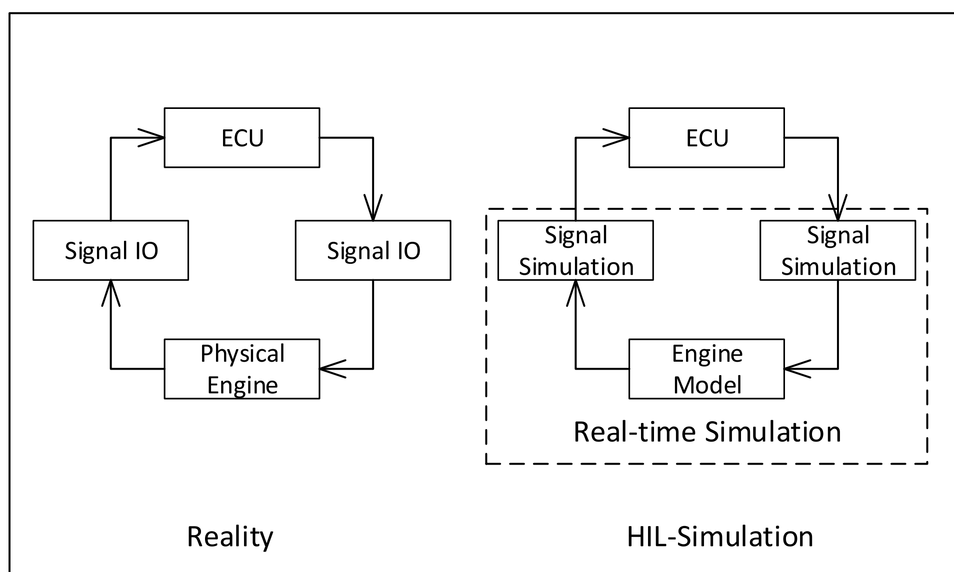

Figure 1-1: Illustration of HIL Simulation

Obviously, HIL is a relatively independent general-purpose environment, but there is still a large gap to fill to make it work perfectly inside Scania's continuous integration (CI) system.

1.2 Problem definition

In RESI department, HIL is not always accessible to all test script writers. To debug or validate their scripts, testers need to wait in queue. On the other hand, the scripts might contain very simple errors before they are tested against HIL, and debugging such kind of errors on HIL is considered a waste of time and resource. This reduces the efficiency of using HIL more seriously. Therefore, it is very important to find a new way to make the debug of the scripts easier and use the HIL more productive.

1.3 Purpose

The purpose of this degree project is to investigate a new testing environment to overcome the current problems and improve the using efficiency of HIL. The new testing environment should be able to do offline debugging in an easier way. Two debugging tools, Dummy and Mat are proposed and implemented in the project and evaluated.

1.4 Goals

The goal of this project is to investigate a new test script environment. The goal has been divided into the following three sub-goals:

1. Background research: which language is used in industry and what is the trade-offs if we switch to the new language.
2. Implementation of a new test script environment: use the chosen language to implement a new testing environment which is able to provide better offline debugging support and better user experience.
3. Evaluation of the new testing environment: proof is required to illustrate the new environment is better than the old one in terms of running time, bug detection rate and user experience.

1.5 Delimitations

This thesis does not discuss how to practically compose a test "course" because this is outside the scope of this thesis. Although we do not have a pre-designed course at hand; fortunately, such a course is completely independent of our environment, hence when we implement related functions we will simply assume that we have a suitable course. More details about test courses can be found in Sections 2.4 and 2.5.

The details of how to use HIL are also not part of this thesis because we use the well-known client-server architecture in our test environment. This enables us to focus on the client part, while ignoring the implementation of the server side (where the HIL is hosted).

1.6 Structure of the thesis

Chapter 2 presents relevant background information about test environment. Chapter 3 presents the methodology and method used to solve the problem. Chapter 4 presents a detailed implementation of the test environment in a systematic fashion. Chapter 5 compares the new and old environment and gives an evaluation of the new one. Finally, the thesis concludes with Chapter

6 that offers some conclusions, suggestions for future work, and some reflections on the relationship of this thesis project with society.

2 Background

This chapter introduces the Python programming language (used in our test environment), the current test environment, and declarative scripts. Section 2.1 explains why Python is still used as the main language in the test environment. Section 2.2 describes the architecture of the test environment and Section 2.4 gives a brief introduction to test script. Section 2.5 introduces the concept of "independent guarded assertions". This concept is very important because it is used to do matching between a script and the corresponding MATLAB (mat) files. This chapter also introduces some additional aspects relevant to the thesis.

2.1 Evaluation of Language: Why is Python used as well?

This section explains why Python is still used in the test environment from two different aspects: a comparison with other languages and the tradeoffs of moving to a new language.

2.1.1 Language Evaluation Criteria

It is very hard to evaluate any programming language in isolation because when we believe one language is better than another, we make this judgement based on our own understanding of and background in the two languages. Moreover, this conclusion might not hold for others in the same team. This means, we cannot simply give each programming language a score and choose the language with the highest score. Additionally, it is pointless to talk about the merits of a single language without considering its application environment. As a result, we need to fully understand the requirements and only then can we identify a language that would satisfy as these requirements. Requirements that cannot be met by the language itself will need to be addressed by tools, either available tools or our own tools.

Ordinarily, before we do a detailed comparison, some languages can be easily removed from our list, such as low-level programming languages (machine languages and assembly languages) and web programming languages (Javascript, Hypertext Markup Language - HTML, and so on).

Generally speaking, programming language evaluation criteria includes four aspects: readability, write-ability, reliability, and cost [5].

Readability is the capability required for a reader to understand the purpose of a text. It includes many aspects such as overall simplicity, data types, control statements, syntax considerations, and so on. Write-ability includes simplicity, support for abstraction, and expressivity [5]. These latter two factors determine if it is easy to implement a certain function in a shorter length of code and whether the result code can be easily and correctly understood by other readers within a shorter period. Reliability involves aspects such as type checking, exception handling, and aliasing (different presentation of the same memory block, for example by pointers, object names, and reference to the same object in C). Cost includes more general aspects, such as the time spent training programmers, writing programs, compiling time, execution time, maintainability, and so on [5].

By implementing a phone-code function, Lutz Prechelt [6, 7] provides a very good example describing programming languages in a context which is quite close to us. Lutz sent the requirements to programmers giving each of them the same requirements and input. The collected result shows that the length of script languages such as Python and Perl was only half the length of non-script languages such as C, C++, and Java, but the reliability of the program shows no observable difference. Within the script language group, Python and Perl were faster in terms of execution time than Rexx and Tcl.

Spinellis, et al. [8] found a similar result. Despite some particular inappropriate circumstances, script languages (Python, Perl, and Javascript) require only one third the number of lines-of-code to implement the same functions as non-script languages. These results suggest that a script language is more suitable for our test environment because expressivity is a valuable merit to our test script-writers. Shorter source code means fewer chances to make mistakes.

Another important factor is the built-in support for data structures and string processing because we need to deal with different data flows and gather test results. This functionality is supported quite well by script languages, such as Python and Perl.

2.1.2 Current Testing Environment

Currently, most of the testing code in RESI's code base is written in Python. To give a more precise impression, we calculated the LoC (line of code) for these testing scripts and related code. Two main folders are taking into account, TC_NCG and main.R2014 while most scripts are in TC_NCG folder and main.R2014 is a test automation framework (TaFw) providing support functionalities such as hardware abstraction, hardware (signal) modeling, function interfaces, tools, communication protocol implementation to servers and so on.

In general, the current project includes 3545 files and 3028 of them are Python files, accounting for 1191784 line of code (in Python). The TaFw project was started four years ago (2010) and delivered in 2014 after two years' preparation. As we can see from this similar example, moving to a new language means a huge amount of work to do and will take years of preparation. Additionally, the testers will need a period of time to study the new features of another language, forcing them to focus on the details of this new language, rather than focusing on the company's products.

2.1.3 Conclusion

Due to the nature of weak or dynamic type systems of scripting languages, many errors cannot be found during compile time[8]. However, we think with the help of offline debugging and other tools or mechanisms, such as unit testing, can solve this problem indirectly. We will discuss this later in Chapter 4. As a result of this chapter, the conclusion is that Python remains the best choice of language for the testing environment.

2.2 Test Environment

Figure 2-1 shows the workflow in our department, RESI (Vehicle Electrical Integration and Chassis System Software). The model is a combination of a general static model and a dynamic model. The dynamic model models all of the dynamic behavior, such as a combustion engine. The static I/O model describes how the I/O boards of the HIL are allocated – i.e., connected to specific hardware, and how the signals are transferred into other units, such as ECUs. The combination of a general and dynamic model is needed for executing tests against the many possible vehicle variants produced by Scania, avoiding the need for a per product model.

Each input and output, also known as a signal, has a unique layered name (such as `root/a/b/c/d`) over the Scania naming scope, which constructs a tree structure from a larger picture complying with their physical subordinate relationship, and the 'root' element identifies a specific server. From the testing code's perspective, each signal is represented as a subclass of "ModelVariable" including the mapped ***set*** and ***get*** paths, block type, I/O type, possible values, and so on.

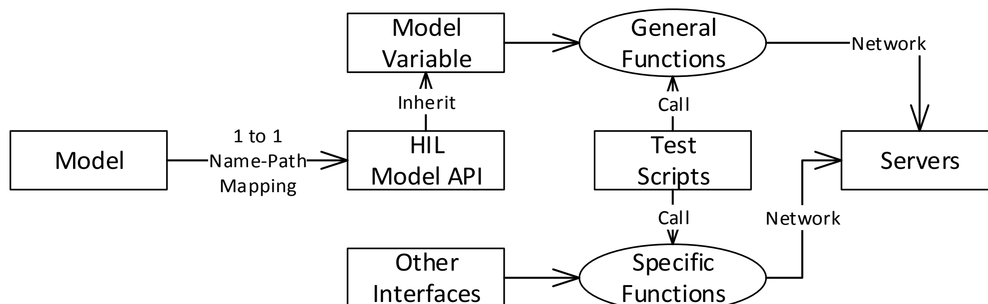

Figure 2-1: Work flow in our Client-Server architecture

Figure 2-2 illustrates the most basic level of communication between client and server, while ignoring the details of the architecture and workflow.

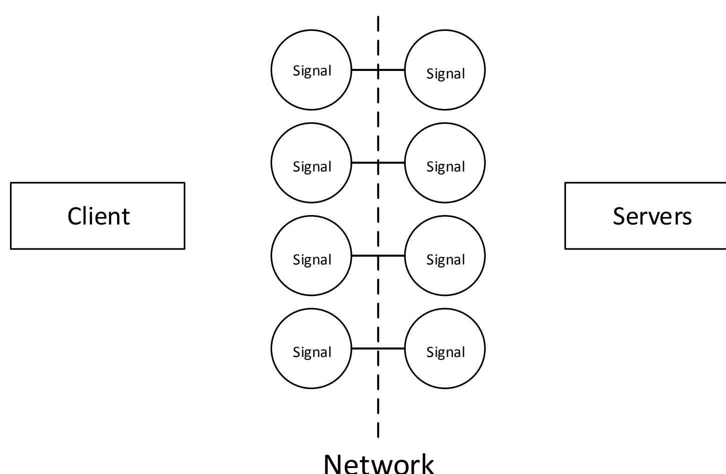

Figure 2-2: Communication between Client and Servers

A client-server architecture is a networking architecture where the client requests a service from the server, and the server processes the request and acts based upon the request [9]. An advantage of using a client-server architecture is that the client and the server can communicate with each other and are independent of their specific implementations – as long as they use a common protocol to communicate. As noted in Section 1.5, this enables us to focus on the client, while avoiding all of the implementation details of the server.

The "ModelVariable" class, shown in Figure 2-1, provides general-purpose functions to implement the underlying mapping relationship to ensure that when a function is called, its corresponding server peer will return a result. This mechanism greatly facilitates the process of manipulating signals in a friendly and human-readable way.

Apart from general-purpose functions, the test environment provides another set of interfaces to facilitate communication between the clients and the servers. These interface modules provide specific functions to the test scripts. In the test environment, some functions, such as setting neutral, starting the engine, or parking the vehicle, are used quite often in many scripts. Furthermore, these functions commonly consist of the same operations. For example, "parking the vehicle" consists of the operations "stop the vehicle based on the gear type", "trigger the parking brake", "set neutral", "release pedal and steering wheel", and "resume the key position". To stop a truck, the test script simply calls the "parking the vehicle" function, instead of calling all of the underlying operations. This enhances readability and write-ability, while decreasing cost since the test author has to write fewer lines of code.

When a function, either a common function or specific function, is called from a script, the request is sent through the network to its destination server. There are many servers in the test environment and they each have different responsibilities. However, we will not go into the details of how these servers process these requests, because these details are irrelevant to our work. Logically our request is simply dispatched to a target server by a name mapping function based on the first name of the requested path (which is 'root' server if a path '`root/a/b/c/d`' is given).

Although the test environment architecture is a standard client-server model, it still has some ***bottlenecks***, and all the problems result from one underlying cause: a tester has only limited access to these servers. In our department, we only have access to these servers 2 weeks out of every 4 weeks. Furthermore, during this time all of our team's members share these servers. It is a waste of time when a script is executed and then a simple run-time error occurs, as the tester now has to either waste resources correcting this run-time problem or yield the server to another tester. In the current client-server architecture not all of these run-time errors can be identified offline (for example, by using PyLint) as opposed to online (when the servers are available and online).

To deal with these problems, composing and testing a new test script is split into three consecutive phases: Dummy mode, Mat mode, and Normal mode. Chapter 4 will introduce each of these phases.

2.3 Symbolic execution

Symbolic execution is mainly used to automatically analyze and generate test cases for ***statically typed languages*** [16]. Instead of actual inputs, the interpreter of the symbolic execution tool uses symbolic values to carry out the execution of programs, ending up with constraints on symbols of each conditional branch, and a formula containing symbols in each branch. By analyzing the constraints and formulas, symbolic execution tools are able to achieve high test coverage [13].

More specifically, consider the following program:

```
1    x = readNumber()
2    y = x / 5
3    if (10 – y == 0)
4        return(failure)
5    return(success)
```

When the program is executed with symbolic execution, the variable x will be given a symbol as the return value of function readNumber(), for example, 'k'. The next line of code will assign variable y with value 'k/5'. Because of the following 'if' statement, the program will terminate with two branches: failure (`10 - k/5 == 0`) and success (`10 - k/5 != 0`), and the failure branch is also marked as a constraint path. After the previous steps, if the targeted result of the program is failure, then the analyzer of the symbolic execution will use a constraint solver to determine that `k == 50` will ensure the failure of the program, while other values of k will result in success.

However, there are two common concerns with symbolic execution:

1. As the size of the program increases, the paths generated by symbolic execution will also experience an exponential growth, even with a dead loop [14].

2. Multiple environmental factors, such as the operating system, user data and the network taking the same (input) path to the program will also pose a challenge to the symbolic execution [15].

Furthermore, symbolic execution will have more challenges when dealing with dynamic languages, for example Python or Perl, in terms of complicated semantics, difficult type inference, and so on [16].

Besides the issues mentioned above, there are also other reasons that symbolic execution is not used in the implementation of Dummy mode. These reasons will be given in Section 4.1.4.

2.4 Brief introduction to test scripts

Each script in RESI (Research-Engine-System-Integration) represents a specific user function. All of these specific user functions are stored in Scania's internal database and can be access through the Scania Electrical System Architecture for Modularization and Maintenance (SESAMM) management system. The scripts follow the same structure – shown in Figure 2-3.

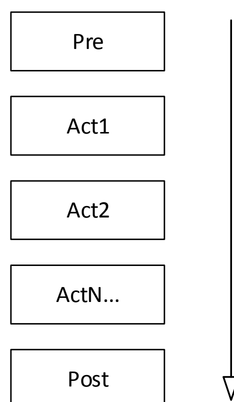

Figure 2-3: Structure of a test script in RESI

The 'pre' function normally includes detection of the System under Test (SUT). For example, if a script is going to test the steering light function, it has to ensure that the key is inserted in the vehicle and that the vehicle is in the correct state. If these preconditions are met, then each of the following actions (Action1 to ActionN – abbreviated Act1 to ActN) will be executed with a stimuli and an associated assertion. Any violation of an assertion will be recorded and will trigger a specific reaction of the execution, such as aborting the script. The 'post' procedure is responsible for collecting the data, generating a final report, and restoring the SUT to a known default state.

There are two issues when executing such test scripts that should not be ignored. The first one is the relationships between these steps, i.e., pre, actions, and post. Although the activities undertaken by each step are encapsulated within the step, these activities still have a strong correlation between each other. This means that the result of one step is strongly related to the activities of the previous step. Another problem is that the 'pre' step contains not only state checks, but may also include some unnecessary activities. These activities are undertaken even if the vehicle is already in the desired state.

Although the two problems highlighted above look quite minor at this point, they greatly reduce the applicability of a script in the new system – unless they are handled properly. A detailed interpretation of these two problems and a proposed solution will be given in Section 4.2.2.

2.5 Independent Guarded Assertions

From the earlier discussion of Figure 2-2 we can see that all the inputs and outputs between client and servers are done through the same super class: ModelVariable. More specifically all of the operations are done by two functions in this class: setValue() and getValue(). Therefore, any script can be translated into another (equivalent) version of the script containing only calls to setValue() and getValue().

It is very common that a script is structured according to the following pattern: “Do A”, “Check A done”, “Do B”, “Check B done”, ... and verify “Assertions” in the last step. For example:

```
Pre(State.idling(), setValue(), State.setGear(), State.setNeutral())
act1(self.toggle_worklight_function(), self.expected_response(assertions…))
Post(setValue(), Event.wait(), State.parked())
```

The above code was taken from an existing RESI script. Obviously, this code can also be transformed into an equivalent using only setValues() and getValues(). Based on the ‘Independent Guarded Assertions’ approach proposed by Gustafsson, et al. [10], setValue()s are classified into a stimuli group, while the remaining functions form another group (i.e., assertions guarded by conditions) as shown Table 2-1.

Table 2-1: List of functions

Do (stimuli)	**Check Done & assertions (guards & assertions)**
State.idling()	guards for idling()
setValue()	guards
State.setGear()	guards for setGear()
State.setNeutral()	guards for setNeutral()
self.toggle_worklight_function()	guards
	self.expected_response(assertions...)
setValue()	guards
Event.wait()	

After this first transformation, the origin script is subsequently transformed into another “independent guarded assertion” script *without* any setValues(). This new script focuses on describing the goals of a script, rather than the steps that need to be taken [10]. At the same time, a set of stimuli (which form the course) is generated and used along with the new declarative script.

Theoretically, the new script can be applied with any course because it will never change the state of SUT (as all of the setValue() operations have been removed). The new declarative script iteratively evaluates the condition of the SUT and decides whether to accept it (as meeting the desired state), or deny it and then repeat the current evaluation in the next iteration as a guard. Figure 2-4 gives a more direct description of this procedure. When an action is taken in a course, an action guard in the script will be used to identify if the action suits the action guard. In the course on the left hand, action A is first tested by the script but fails to satisfy its first guard, so the course moves to next action and the script remains its initial step (1). The next execution of the course is action B (2), which satisfies the first action B guard of test script, so the script will also move on to the second step (2) which corresponding to step (3) of the course. Therefore, after the first three executions (step 1-3) of the course, all guards in the script are satisfied which will trigger the assertion of the current system state. After any consecutive sequential execution of B and C, the assertion will be made. In this case, the assertion is used twice, hence the script is tested twice as well. It should be highlighted that at any time after B and C are matched, the assertion (3) *must* hold or the script will fail because B and C are sufficient and necessary conditions for the assertion in step 3.

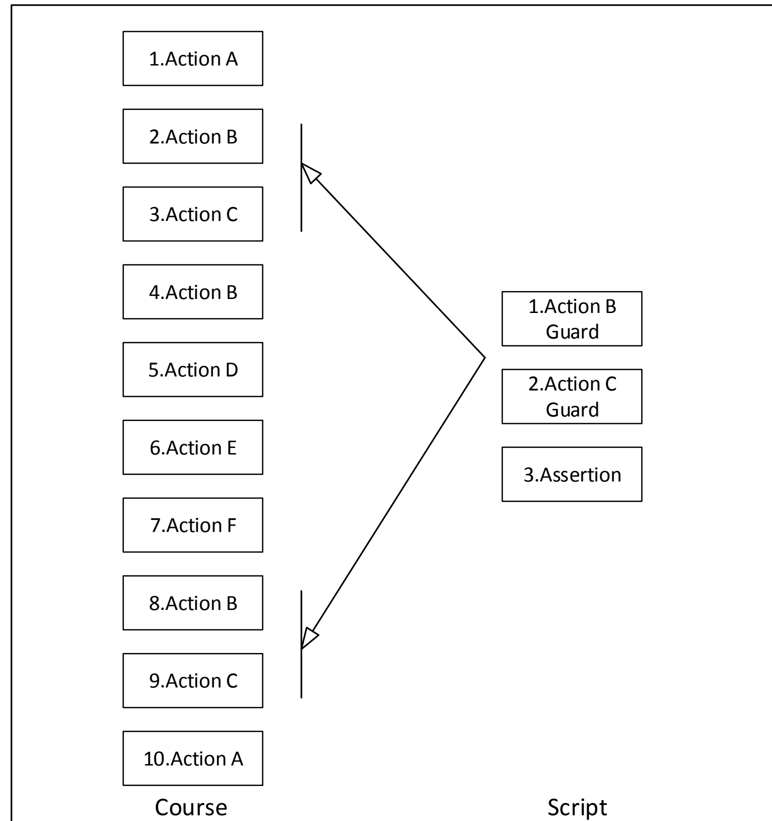

Figure 2-4: Matching of a course and a script

The 'Independent Guarded Assertions' design has the following merits:

1. Declarative scripts can be executed in parallel; thus saving a lot of time.
2. Declarative scripts are applied iteratively as many times as possible. As long as a scenario matches the script, the assertions will be tested. This increases the applicability of a script.
3. By performing statistical analysis of the (current) scripts, it is possible to derive an optimized and more meaningful course. This optimized course can be executed concurrently with multiple declarative scripts. For example, we can predefine a course containing a series of actions: starting the vehicle, ignition, speeding up, slow down, steering left or right, reversing, parking the vehicle, enable and disable the hazard warning lights, and leaving the vehicle. During this course, many scripts can be tested multiple times during one execution. For example, the following scripts could be evaluated: 'hazard warning activation on and off', 'reverse light activation on and off', and so on.

As a result, the declarative scripts can evaluate the correct functioning of a subsystem (in the case above, the hazard warning and reverse lights) both multiple times and in many different test scenarios (see Figure 4 in [10], the assertions of the scripts can be triggered simultaneously in a long course).

2.6 MAT-files

MAT-files are binary files used to store data generated by MATLAB. By using MATLAB's save() function the arrays of a running MATLAB function will be stored into a MAT-file as a continuous byte stream [6]. In general, there are two levels of MAT-files: level 4 (compatible up to MATLAB version 4) and level 5 (compatible with MATLAB 5 and up). MATLAB 8.2 is used in RESI, so the

level 5 MAT-file format is used throughout the project. A level 5 MAT-file consists of a Header and multiple Data Elements. Figure 2-5 shows the standard structure of a MATLAB level 5 MAT-file.

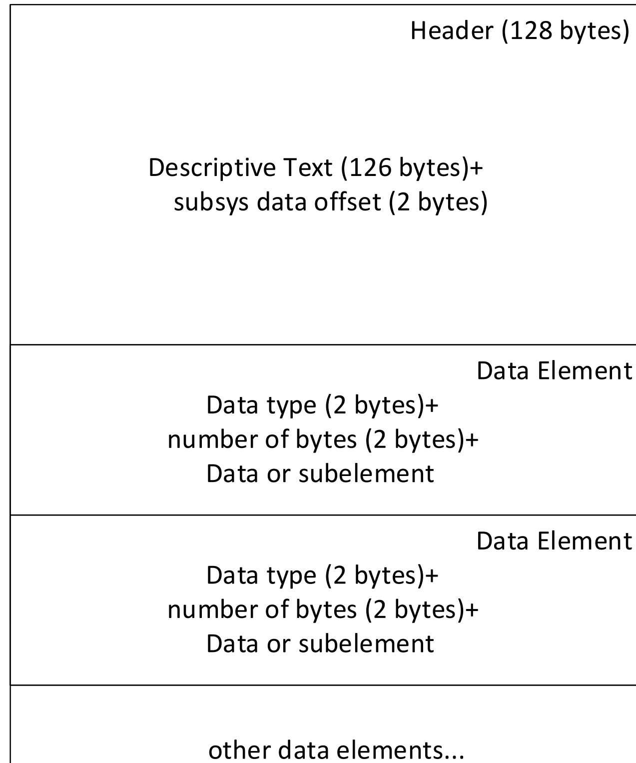

Figure 2-5: MATLAB level 5 MAT-file format

The Python Open Source Library of Scientific Tools (Scipy)* set of packages provides a set of interfaces to interact with MAT-files [16] without requiring that the programmer know the details of a MAT-file. After installing the Scipy package (version 0.16.0), the function loadmat(), found in the scipy.io package, can be called to return a standard Python dictionary consisting of Data Elements from a MAT-file as key-value pairs.

2.7 HIL in Scania

HIL has become the current de facto tool within the vehicle industries for testing ECUs [8]. Within Scania, ECUs and the buses connecting to these ECUs are the objects to be tested using the HIL environment. Automotive Simulation Models created with MATLAB are applied to simulate operations against the related hardware [9]. As a result, the ECU and one or more busses physically exist, while all of the rest of the system are realized by HIL.

In RESI, the HIL is provided by dSPACE corporation. Today dSPACE is highly involved in the vehicular, specifically automotive and aircraft, industries and provides both software and hardware to accelerate the development and testing procedures for vehicles.

In RESI, HIL is used in a more elaborate way than is typical in industry. HIL is deployed in a client/server fashion, where scripts are executed in the client machine and the HIL is connected to the servers. This client-server architecture isolates the technical specification of clients and servers, enabling them to be implemented with any suitable tools [10]. Moreover, this means that the client and server environments can use completely different choices of programming languages.

* http://www.scipy.org/

3 Method, methodology, and tools

This chapter introduces the tools and methods used in this project.

Unlike a problem-solving project with a list of functional requirements or performance indicators, this project was designed to be open to a variety of ideas (including use of a new programming language, a new Integrated Development Environment (IDE), or a new set of tools), as long as collectively they achieve the desired goals (as stated in Section 1.4). A good strategy when facing such an open-ended problem specification is to sort all of the requirements by priority, then eliminate alternatives that do not satisfy an *essential* requirement. Furthermore, it is also critical to restrict the research area due to limited duration of this project. That means that it is not practical to use a long time to solve any single problem. For this reason, weekly feedback was used to provide nearly continuous feedback keeping me focused and saving a lot of time.

To understand the current testing environment's advantages and disadvantages, a full case study and literature review of Scania's internal resources was necessary. This helped me to understand the workflow from how a script is composed from scratch to how it is applied during testing. A literature review of research papers and articles was used to investigate what other solutions have been proposed by other researchers and industrial companies.

3.1 Feedback meetings

A weekly discussion was held with Thomas Gustafsson (my supervisor and the department leader at Scania RESI) to develop my understanding and guide my implementation of the new test environment. This discussion focused on the following topics:

1. The summary of the previous week's work;
2. Feedback on the current design and implementation;
3. Planning the coming week;
4. Examining the anticipated result(s) and the gap remaining between this and the current work;
5. Focusing on specific results from the above;
6. Identifying problems and solutions.

3.2 Priority checklist

A checklist with priorities (see Table 3-1) was proposed during the initial phase of the project based upon the series of interviews (described above).

Table 3-1: List of priorities (ordered by priority)

‘Must have’	‘Better to have’
Offline debugging	GUI support with offline debugging
Efficient static analysis tool	Able to be integrated into Scania's Continuous Integration (CI) environment
Able to run automatically	Management of scripts (layered by functionality)

On the left hand side of Table 3-1, we can see that ‘offline debugging’ is the highest priority in the ‘Must have’ group because if this capability is provided, then providing ‘efficient static analysis tool’ and ‘typing check’ can also be tackled. Currently, PyLint* is used to do static check with restricted usage (details can be found in Section 4.1). If a run-time debugger could be applied together with PyLint, this would provide the top three ‘Must have’ priorities. The reason why ‘run automatically’ is of the lowest priority is that since the new environment is implemented in Python, this is easy to achieve - as long as the interfaces are designed to be triggered by external tools together with the appropriate parameters.

On the right side of Table 3-1, we see that having a GUI that supports offline debugging would be of great help when a new script is composed. This assumes that this debugger would give a direct hint as to the location where the script is likely to fail. Integration into CI is optional since only limited work is left after the ‘run automatically’ functionality has completed. ‘Management of the scripts’ is quite close to the functionality and purpose of the script itself - which is to some extent beyond the scope of this project. As a result, all of the priorities on the right side of Table 3-1 are optional. Whether they will be included in this thesis project will be determined based upon the progress made on the other priorities of the project and whether sufficient time is available to realize them.

3.3 Case study and literature review

Hardware-in-the-loop simulation is widely used in many areas such as automotive [17], power trains [18], heating/cooling industry [19] even unmanned aerial vehicle [20]. In automotive industry, many manufactures use the so-called V-model to design, implement, and validate their production with HIL and other related modeling fashions such as SIL (Software-in-the-Loop) [21]. The following figure depicts the V-model used in Scania.

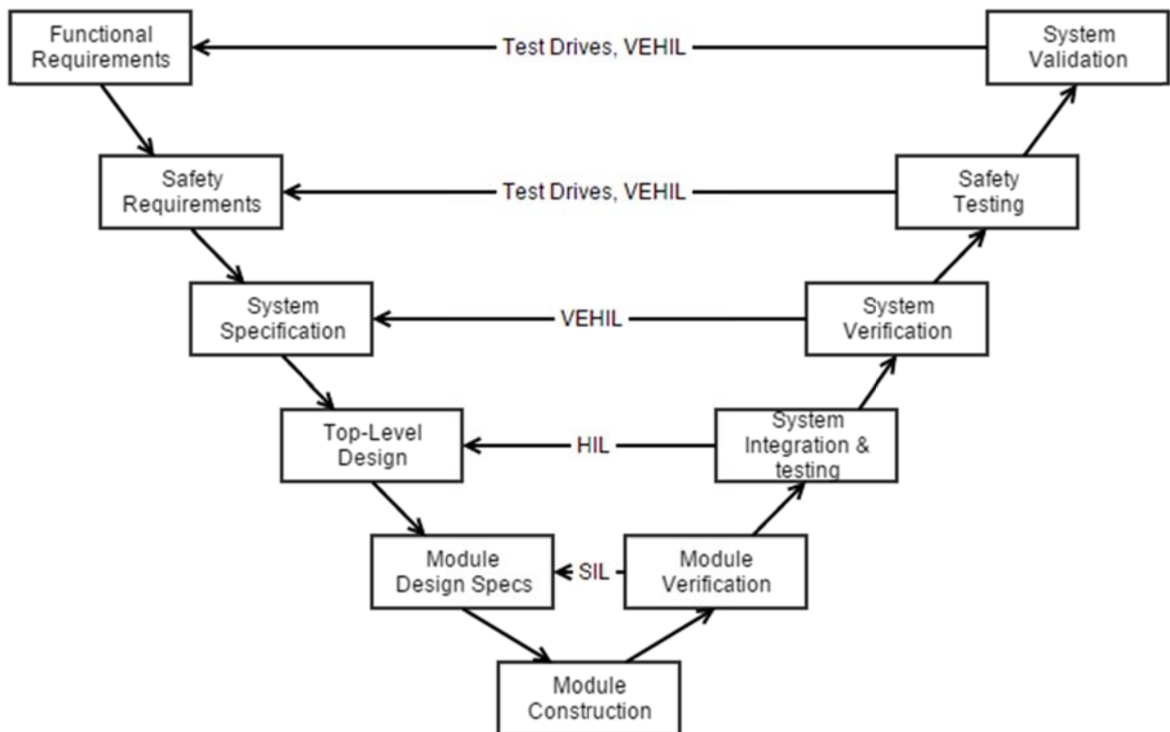

Figure 3-1: V-model in Scania

* http://www.pylint.org/

On the left side, the V-model goes from the top down to design the whole system in steps and goes up on the right side to validate the design and implementation. As we can see from the figure, “system integration and testing” utilizes Hardware in the loop (in our group, RESI). On the bottom of this process, “Module Verification” uses Software-in-the-loop (SIL) to verify module design specifications.

Ideally, SIL can help a lot with our test script debugging. Take the development of an ECU as an example. Before an ECU is physically implemented, a software prototype (or a simulation of part of it) can be carried out with the help of SIL to simulate its hardware [22]. This helps the ECU developers validate their design, while the simulation is a basic software version of the ECU. To validate our scripts, we can simply drive the software ECUs and execute the scripts to get the validated result.

However, this is not easy as it looks like. For example, because the ECUs are developed by different suppliers, their “software version” is not accessible due to information security and patent protection rules. Additionally, the job of the RESI group is integration and testing. That means many ECUs from different suppliers will be involved in the testing. It is quite common that these ECUs come from various suppliers, which makes the problem even more complicated. For now, the RESI group tryies to solve this problem by modeling the ECUs, but this topic is outside the scope of this thesis.

In contrast to the Client/Server architecture used in RESI, based upon those papers we read most of the Hardware-in-the-loop environments are built locally. For example, Cătălin Vasiliu and Nicolae Vasile [18] used AMESim and LabVIEW to model and simulate powertrains, with the HIL test bed directly connected to a PC. In [23], the simulation is also finished locally. In [20], the authors describe their system architecture and setup in detail. The HIL is used to connect the control board and control system to simulate the dynamics of a real vehicle. All of the hardware is connected to a CAN bus and then to a PC through a serial link. The simulated (fake) process executes on a standard Linux system locally.

3.4 Architecture and algorithm design

As mentioned before, the bottleneck of the existing test environment is the limited access to validation resources (i.e., limited access to the HIL hardware). As a result, the test script writers cannot get immediate feedback (by running their tests and getting results) on their latest scripts until every piece of the whole test chain is available ‘online’.

From a general view of the whole process (referred to in the following discussion as a “cycle” or “module”), there are two ways to handle this limited access to resources and the resulting inefficiency when writing tests:

1. **Early error detection**: Try to find more bugs before a new script goes online. This will greatly increase the productivity of the HIL when it is available for use.
2. **Reproduce and reuse the HIL results**: Normally, when composing a new script, the stimuli (of a given script) will not be changed *even if the script contains errors* - because each script has a fixed corresponding use case. Instead, a script will be executed several times while debugging the script, but the stimuli frequently remains (almost) the same. This enables the test script writers to test a script many times - while only needing to utilize the HIL hardware once.

On the other hand, there are other solutions, such as add more HIL servers and create a full emulation of the HIL servers. Honestly, buying more HIL servers can definitely solve the problem but our hands are tied in the department budget. Creating a software emulation of HIL, which is known as software in the loop (SIL) as we stated before, is technically available, but that requires

the modeling and implementation of the ECUs and all related IO and behaviors. This is considered a huge effort to put in.

It is possible to build smaller autonomous test modules which do not rely on external inputs and are able to generate internal outputs. This enables the construction of drivers and stubs for a specific module, enabling this test module to execute and operate independently. Drivers feed inputs to the test module, while stubs collect the output data from the module [11].

Depending upon the resources required, the current environment can be divided into three modules which can be driven independently in three different modes: Dummy mode, Mat mode, and Normal mode. Figure 3-2 shows these modes and their associated purposes and context.

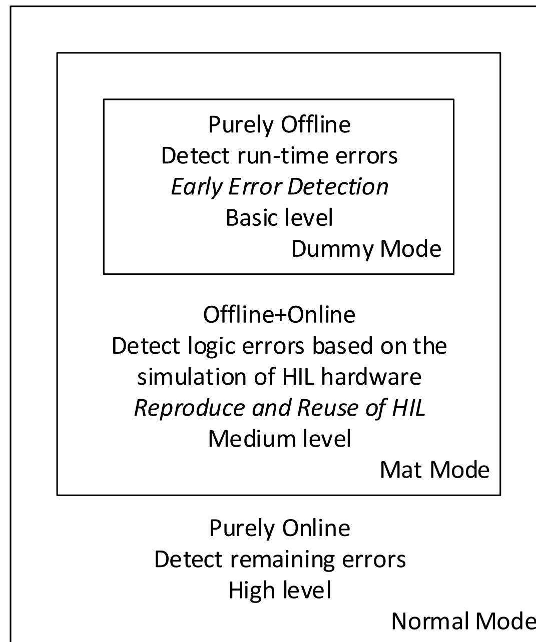

Figure 3-2: Three modes

All inputs and outputs of a script are done with setValue() and getValue() through a subclass of 'ModelVariable' which contains all possible values of a signal. An example is shown as follow, which a signal (also as a subclass of ModelVariable) and all its possible values (-1, 0 and 1) are listed.

```
1    class DS_TurnSignal(ModelVariable):
2        __api_get_path__     = "yellow3/Model Root/Yellow3/ControlPanel/
3        __api_set_paths__    = ("yellow3/Model Root/Yellow3/ControlPanel/Dr…
4        __api_base_paths__   = ("yellow3/Model Root/Yellow3/ControlPanel/Dr…
5        __api_text__         = "DriverSwitches.Visibility.DS_TurnSignal"
6        __api_block_type__   = "dSPACESetTASignal"
7        __api_io_type__      = "IO"
8        __api_default_values__= None
9        TURN_LEFT = -1
10       OFF = 0
11       TURN_RIGHT = 1
```

So how should one activate the "Turning Left" and read the current state of the Turning signal? The function setValue() and getValue() is defined in ModelVariable, so the tester can simply call these two functions with DS_TurnSignal object to achieve this goal.

Dummy mode is designed to identify run-time errors, such as 'too many values to unpack' or 'list index out of range'. For testing purposes when the system is running and it tries to get a value of

a signal, the first of the possible values of the signal will be returned. This happens until the end of the execution of a script and the chose values will be recorded as a "path". The path will be remembered and removed from Dummy mode for the next execution, thus ensuring that there are no missing or duplicated paths. The reason we choose to test our script this way is because we are fully aware of the input of the program (script), and with the help of optimizations we introduced in Section 4.1.2, we can further reduce the size of the input set. For the sake of execution speed and complexity, a dynamic tree structure and an exception list are used to represent the execution paths. Section 4.1 presents the details of this implementation.

In order to test a script with mat files, a transformation from the (original) imperative script to a declarative script is required. This transformation can be done in a few steps. After matching a script and a mat file, the GUI displays a report indicating conflict points (if any) to assist the test script writer. A conflict occurs when the value of a signal is expected to be X in the script, but is found in the Mat files to be value Y. Many signals are involved in the execution of a script, therefore in the report only ten signals plotted (this choice is based on the resolution of the user's screen). In order to provide more precise information, these signals are sorted vertically based on their relevance to the conflict. A covariance matrix is used by the sorting algorithm, where the covariance value expresses the strength of the correlation of two or more sets of variables [8].

3.5 Software tools

A number of different software tools have been used in this project. Each of these is briefly described in the following paragraphs.

Python 2.7 was utilized because most of the current code, including the tool chain provided by dSpace is written in Python version 2.7. An appealing point (which is closely related to solving the problem to be addressed by this project) is that Python 3 introduces function annotations [12]. However, the pay back is expected to be low in comparison with the effort required to shifting from Python 2 to Python 3. For this reason, Python 2.7 will continue to be used.

PyLint 0.28.0 is a very good static analysis tool for Python programs, hence it has been used in this project to facilitate the offline debugging of test scripts.

Pycharm 4.5 was used to develop the project. Pycharm is a popular Python IDE with some helpful features such as intelligent coding assistance, smart code navigation, effective code refactoring, and so on.

Matplotlib 1.4.3 was used to implement the GUI assistance module. Matplotlib is a 2-dimensional plotting library implemented in Python. It can generate high quality figures and provides various means of implementing interactive operations. This version was the latest stable version (as of when the project is being conducted).

SciPy 0.15.0 was used to load data from Mat files.

NumPy 1.8.0 was used to calculate the covariance for the ordering of the signals.

Jenkins is a tool for monitoring repeatedly executed tasks, for further details see https://wiki.jenkins-ci.org/. Jenkins is used to automate some of the testing (Section 5.3, and Section 6.3).

4 Implementation

As we mentioned before, the 'Dummy' and 'Mat' mode operate on different levels. Dummy mode performs run-time checks on all paths of the script, while in Mat mode the scripts will be executed with Mat files. The following sections of this chapter will give details of the implementation of Dummy mode and Mat mode.

4.1 Dummy Mode

This chapter will introduce the design and implementation of Dummy mode. Dummy mode is the initial step of the test environment. Based on the variables a script used, Dummy will do exhaustion on the values of variables to investigate the errors of a script. The optimization of the algorithm will also be given in the following chapter.

4.1.1 Algorithm and Implementation

As stated before, the class ModelVariable is the superclass of classes used to communicate between client and server. Each subclass of ModelVariable contains all possible values of this signal, thus it is possible to take over the control of the client program locally by feeding it different values without any server.

In Dummy mode, the script will be executed several times to test all possible paths. To implement this, Dummy mode has two programs running alternatively: an execution program and a trim program. The execution program runs first. Initially a list of objects (an execution list) will be generated, then the script is fed with the first possible value of each signal object. For example, the values returned from the Dummy mode of the execution program is [1,1,1,2] of signals A, B, C, and D, which is shown in Figure 4-1 in the middle of the list of signals.

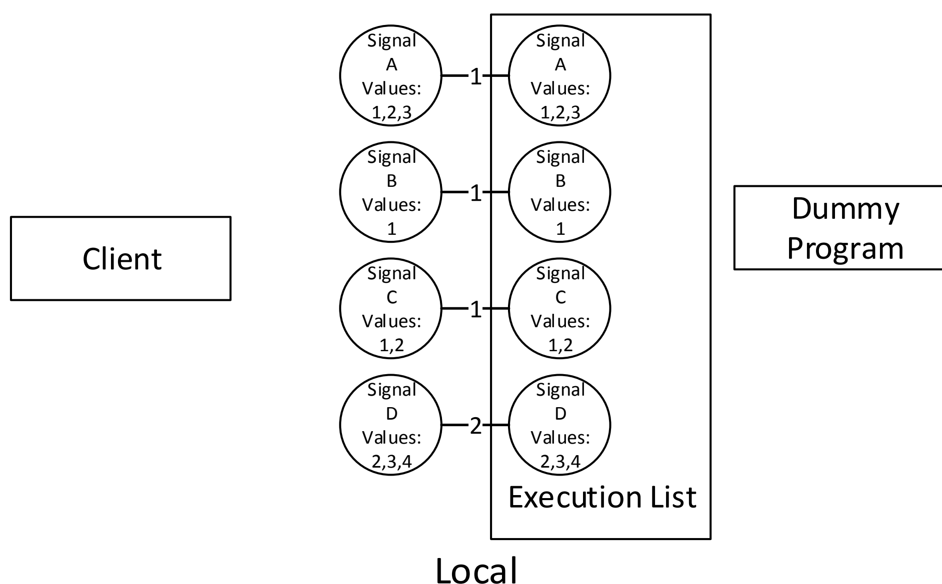

Figure 4-1: First Run of execution program

The execution list is a dynamic list where objects are removed or added in each execution. When this dynamic list is empty, then the execution program will be terminated. After the first run of the script, the history of the execution list is scanned and trimmed for next run. The trim algorithm works as follows:

(1) The first value of the last object in the list is removed (which is value 2 of signal D in Figure 4-2). Because the execution program always uses the first possible value as a result, removing this value of the last object will remove the latest tested path, as shown in Figure 4-2. Because the path [1,1,1,2] has been tested, the value "2" of signal D is removed after the first run of trim. Values [1,1,1,3] will be returned when the execution program is executed next time.

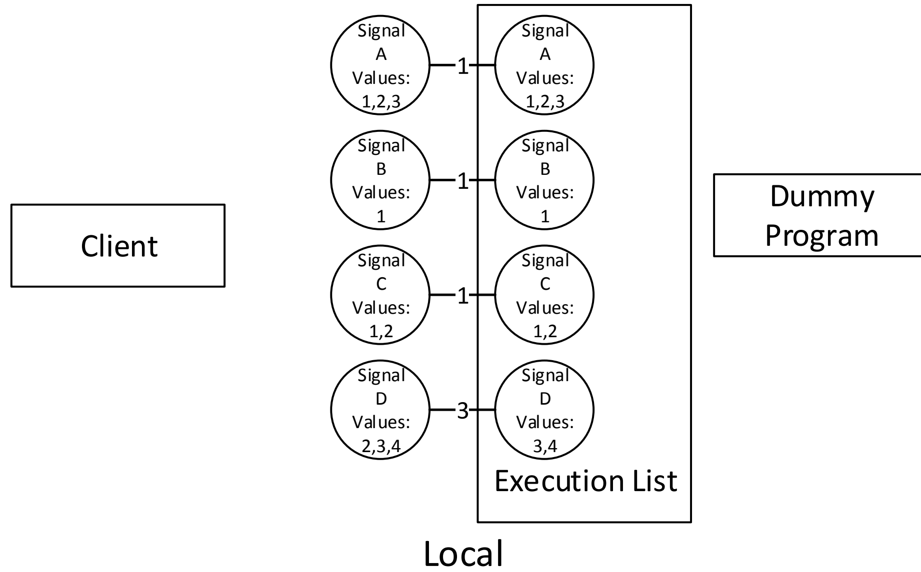

Figure 4-2: **After the first trim program (second run of execution program)**

(2) If there is no remaining value for the last object, then the trim function will remove this object, and repeat step (1) on the next to the last object (which is now the last object in the list, signal C). This loop will stop until there are more than one values of the last object in the execution list, or there is no object left in the list (and the program exits). From an execution path perspective, removing an object from the list means a branch has been fully tested. Figure 4-3 shows that signal D has already been removed and the branch of signal "C" with the value "1" was fully tested and hence and the value "1" of signal C is also removed.

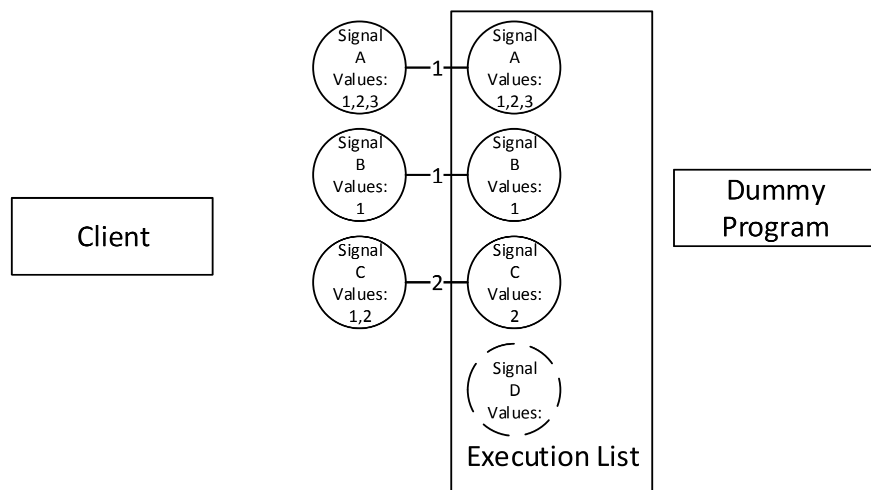

Figure 4-3: **Branch removed**

The Dummy program can also add new objects into the execution list, which represents the case where a new branch is executed and created. Figure 4-4 shows signal E has been created and added to the execution list after the path for signal C with value“2” is being tested.

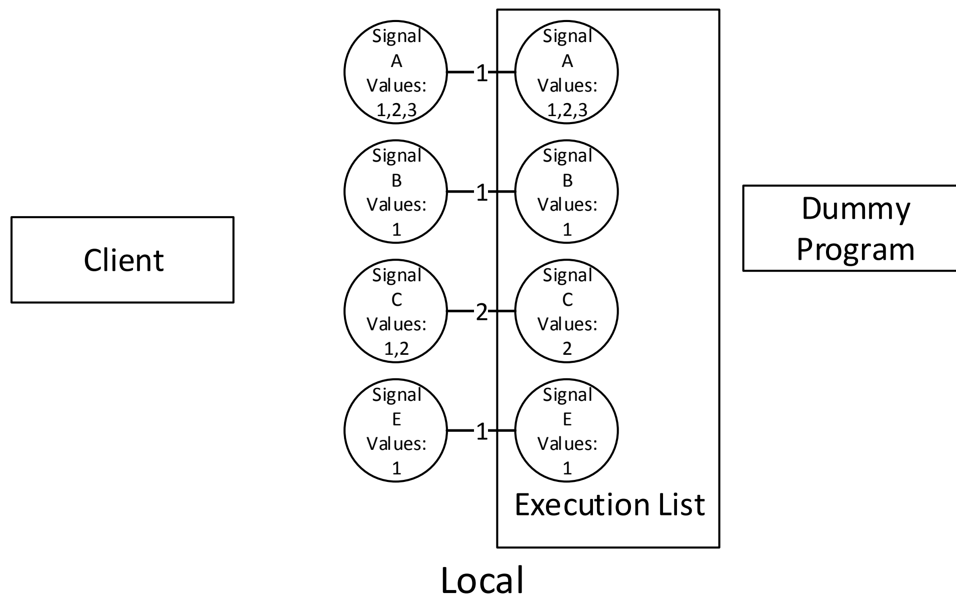

Figure 4-4: New Branch created

After the execution program stops, information regarding errors and exceptions are collected in a global file in dictionary format of Python and is showed in command line in the end. This information will also be used as results for Jenkins. Figure 4-5 shows how the whole procedure works.

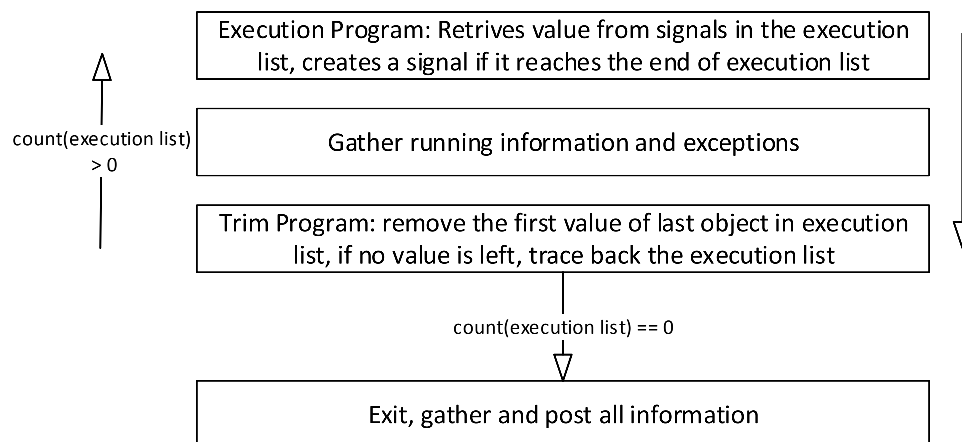

Figure 4-5: Gather information and exceptions from the running program

4.1.2 Optimizations

A common problem with this brute force exhaustive testing is performance. One of the solutions is to eliminate or restrict the exploration of some unnecessary paths. In our case, an exception list of signals is generated in advance of Dummy mode testing. Any signal that belongs to this exception list is given one or a set of default value(s) to avoid exploration for all its actual internal values. The signals are classified as follows:

1. Hardware environmental signal exception list

Since the Dummy program runs purely in software, it is necessary to avoid exhaustive testing on hardware-related signals when the script is running in Dummy mode. These hardware signals, such as “connect the battery” and “turn on battery switch”, are normally guaranteed for the hardware to work properly and have no influences on the logic or result of the execution. As a result, all of these types of hardware signals are pre-registered in the exception list and given a default value.

2. Script related signal exception list

In RESI, each action in a script ends with a set of assertions to test if the SUT is functioning as expected, but in Dummy mode, the result of these assertions is not our concern because we are only interested in testing all possible execution paths of a script, rather than the success or failure of an assertion. From another perspective, the execution path will ***not*** change due to different results of the assertions.
As Figure 2-3 showed, a “post” procedure is executed at the end of each script to clean up the environment for the following executions, but in Dummy mode, this is unnecessary because the Dummy program is stateless in terms of hardware. As a result, the post procedure is not executed in Dummy mode.

3. Functional signal exception list

Event.wait() is used in the script to trigger a synchronized suspension of the program (for a certain period of time) in order to wait for the occurrence of a given event; such as, waiting for a corresponding event (ClutchPedal <= 5) after releasing the clutch pedal (set ClutchPedal == 0). But since Dummy is running on purely software level, the pending operation should be removed to save time. Besides, there are only two consequences of Event.wait(): abortion or continuation. In either case, the consequence of this wait operation will not change the execution path.

4. Specific signal classes

There are some special signals used to communicate between client and server which only have ranges of values instead of possible values, such as EngineSpeed. Doing exhaustive testing for each possible values of a variable like this will be a waste of time. For example, if the range of EngineSpeed is from 0 to 10000 rpm, rather than doing exhaustive testing of all ten thousand cases (numbers), it is more productive to focus on some specific values, such as 0 (shut down), 3000 (stand by), and 10000 (running) representing different status of the engine. For such variable values, it is a good idea to put these variables into the exception list with a sufficient number of representative values.

As can be observed from the above, adding any of these types of signals to the execution list will increase the execution time, but will not change the execution path, hence these signals will be added to the exception list to save time and improve performance.

At the beginning, Dummy is designed to test all paths and return the aggregated error message to users, but we realize this is not a good strategy. Dummy as a debugger aims to find bugs, and the bugs should be found as early as possible to save the waiting time of users. Besides, it is quite possible that a bug triggers a series of error message. Investigating all error messages is also a waste of time for users. As a result, we setup a configuration to determine if Dummy will stop when a bug is found, this switch is ON if Dummy is executed by normal users and it is OFF when executed on Jenkins.

4.1.3 Analysis and Validation

In this section we will demonstrate a code snippet and its generated Dummy paths to analyze and validate the Dummy mode and its optimizations.

The following code is used to ignite a vehicle. It will basically be used in all scripts. The expected vehicle status should be “parked” and “ignition on” after the function is called.

```
1   def ignition_on():
2      _connect_battery_and_main_switch()
3      key_state = get_value(driver_variables.DriverSwitches.PowerSupply.KeyPosition)
4      engine_speed = get_value(asm_mdl_drv.MDL_DISP.EngineSpeed.n_Engine)
5      if key_state != driver_variables.DriverSwitches.PowerSupply.KeyPosition.IGNITION or engine_speed != 0:
6         if engine_speed > 0.5:
7            Event.wait(
                (asm_mdl_veh.MDL_DISP.Overview.v_x_Vehicle_CoG,
                        Event.plusminus(0, 2), Event.ACTION_FAIL_RETURN),
                 (asm_mdl_drv.MDL_DISP.ActiveTransmission.Gear,
                        0, Event.ACTION_FAIL_RETURN), timeout = 60)
8         set_value(driver_variables.DriverSwitches.PowerSupply.KeyPosition,
                 driver_variables.DriverSwitches.PowerSupply.KeyPosition.IGNITION)
9      set_value(driver_variables.DriverSwitches.Brake.DS_ParkingBrake, ON)
10     set_neutral()
11     utilities.sleep(2)
```

We only call this function in a script and execute the script in Dummy. The following is the output from Dummy.

```
1   Exceptions found:0
2   ------------------------------------------------------------------------------
3
4   total paths tested:105
```

As we stated in the chapter 4.1.2, the hardware environmental signals (BatteryConnect and BatteryMainSwitch) used in _connect_battery_and_main_switch() will be registered in the exception list and given a default successful value. The KeyPosition (line 3) signal has 5 possible values (KEY_REMOVED: -1, KEY_INSERTED: 0, RADIO_MODE: 1, IGNITION: 2 and START: 3). In line 4, the script tries to get the current engine speed. The possible values are (0, 3000 and 10000). Function set_neutral() (line 10) is shown below:

```
1   def set_neutral():
2      if gearbox_is_working:
3         gearbox_type =
get_value(asm_mdl_drv.MDL_DISP.CUSTOM_SWITCHES_DRIVETRAIN.Sw_GearShifter)
4         if 1 == gearbox_type:
5            set_gear(0)
6            res = Event.wait(
                (asm_mdl_drv.MDL_DISP.ActiveTransmission.Gear, 0, Event.ACTION_FAIL_RETURN),
                timeout = 30)
7            if res != 0:
8               # Could not set neutral gear
9               Print().debugPrint("Failed setting manual gearbox in netural")

10        elif 2 <= gearbox_type <= 7:
11           set_gear(Gears.N)
12           res = Event.wait(
```

```
                (asm_mdl_drv.MDL_DISP.ActiveTransmission.Gear, 0, Event.ACTION_FAIL_RETURN),
                timeout = 30)
13          if res != 0:
14              # Could not set neutral gear
15              Print().debugPrint("Failed setting gearbox in netural")
```

Variable gearbox_is_working is a global variable and is True by default. There are seven possible values of signal Sw_GearShifter, including the KeyPosition and engine speed which increases the combination number to 5*3*7 = 105. It is easy to see that these value combinations are able to cover all execution paths. On the contrary, if the optimization is not applied, the total paths will be 350000. The exception list helps a lot to reduce the validation time of Dummy mode.

4.1.4 Why not Symbolic execution

The algorithm and implementation of Dummy mode is similar to Symbolic execution at first glimpse, but when looking into the details, we found that our customized tool, Dummy, is more capable and suitable to be integrated into our current test environment based on the following reasons.

First of all, as a general-purpose code analysis tool, the Symbolic execution cannot be used to carry out the analysis of the signal classes of our code. For example, after analyzing the class DS_TurnSignal (listed in chapter 3.4) with Dummy, the result is that the possible values are -1, 0 and 1. This conclusion is drawn based on the truth that the possible values are the class members which the names are not surrounded with two underscores (which are class members TURN_LEFT, TURN_RIGHT and OFF of DS_TurnSignal class, for example). Similar analysis of signal classes is done in Dummy mode. Symbolic execution is not aware of this customized convention in the naming of the signals members, so it cannot provide any useful information from this point of view.

Secondly, as stated before, distributed systems, such as our test environment with a client-server architecture, poses a big challenge to Symbolic execution because of their complexity in networking. In our case, after analyzing related signal classes, Dummy mode registers all possible values and is able to use them locally, as if they are received from networking. This can greatly reduce the complexity of testing our scripts, but Symbolic execution is unable to do that.

The last and most important reason is, from the system design perspective, as the first step in our testing environment, it should be able to output customized analysis results and data structures (signal objects) for the next step (Mat mode). This is important and necessary for Mat mode because it will use these signal objects to form sequences, patterns and draw diagrams. But as a static code analysis tool, Symbolic execution can only generate testing results and related documentations.

4.2 Mat Mode

This chapter introduces the underlying mechanism of Mat mode and the detailed implementation, along with GUI to help locate logic errors in the script.

4.2.1 Algorithm and Implementation

Mat mode goes further than Dummy. Based on the theory of Thomas Gustafsson et al. [10], the test script is separated into two parts: stimuli that drive the SUT and independent guarded assertions. In Mat mode, operations in a script are firstly read as *sequences*, and then a *pattern* representing a test action is formed by these sequences (further details are given below). This pattern is used against a Mat file to evaluate the independent guarded assertions. The following gives a detailed description of the implementation:

1. Transform script operations into sequences

   In Dummy mode only "get" operations are considered because the returned values are the only factors determining the behavior of a script. In contrast, in Mat mode all types of operations, including "set", "get", "wait", and "assertions", are considered. Each of these four types of operations will be transformed into a unique class: sequence. In this transformation, the "wait" operation is more complicated because in a script it normally consists of many simultaneous getValue()s and subsequent reactions to the result of applying these value. A sequence object stores all the information relevant for each operation.

2. Create a pattern from the sequences

   In Mat mode, operations are captured by a Recorder module. The Recorder listens to the calls to ModelVariable, then based on the type (set, get, wait, or assertion) of the caller, it will create a sequence. Once the script has completely executed, a complete set of sequences, also known as a pattern, is generated. The following is a partial Recorder produced log of a pattern transformed from the test script TC0005. The full transferred log can be found in appendix.

```
///////////////ALL SEQs in Pattern///////////////////
SEQ: TYPE:31,
Name:set_neutral,
Value:None,Va:None,Action:0 with parallel of 0
False
SEQ: TYPE:2,
Name:yellow1/Model Root/Yellow1/ControlPanel/Driver_Switches/Brake/DS_ParkingBra
ke[0|1]/Control/Value,
Value:0,Va:0,Action:0 with parallel of 0
True
SEQ: TYPE:2,
Name:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_MainL
ightSwitch[0_3]/Control/Value,
Value:2,Va:0,Action:0 with parallel of 0
True
SEQ: TYPE:2,
Name:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_Workl
ightSwitch[0|1]/Control/Value,
Value:1,Va:0,Action:0 with parallel of 0
True
```

3. Run the test script against a MAT-file

   To evaluate the independent guarded assertion, we apply the mechanism described in [10]. All operations before an assertion in a script are guards of this assertion ({guard=>assertion}). The assertion should hold after all guards are applied.

4.2.2 Analysis

The algorithms introduced in Mat mode proposes a method of transferring imperative scripts into declarative scripts. The reason why we are able to achieve this is that we do not do the transformation from the programming language perspective, instead, we do this based on the nature of our scripts and testing environment:

1. The script communicates with the server only through signals, and for each signal, we know its entire possible values. This is very important because it greatly reduces the complexity of the communication of a script and a server. It helps us to abstract the

operations of a script and normalize them into sequences and patterns. For example, if a client operates a server through many different ways, such as command lines, function calls, remote procedure calls even through binary files, we would have never been able to do this transformation because there is no way to normalize them and express them in declarative way.

2. The transformed declarative scripts can be applied to Mat files. The nature of declarative scripts in our use case is to describe scenarios including the preconditions and assertions, but there is no information in the declarative scripts indicating what commands, or signal values should be accepted and executed by HIL servers. In short, the declarative scripts cannot drive HIL servers, but can be executed against Mat files.

Another benefit from the transformation procedure is that the operations of a script and expected results are separated. If a script is written in imperative way, the script is only able to be executed and validated once because the script itself has already strictly defines the operations required to be taken and the expected results. The test scenario will not be validated if another script contains exactly the same operations, shown in Figure 4-6.

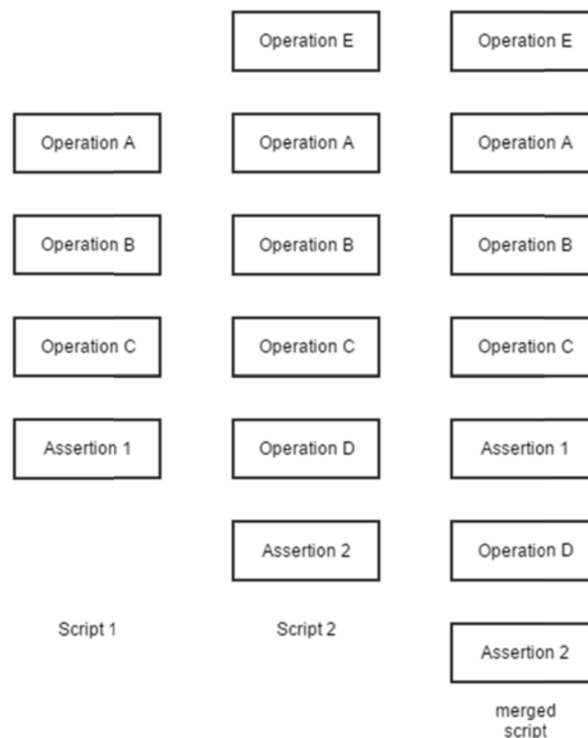

Figure 4-6: Same Operation in two Scripts

In the figure shown above, assertion 1 cannot be not validated in the execution of script 2 because they are not related at all. But if the scripts are written in declarative way, the assertion 1 in script 1 will be validated when the operation series (operation 1 - operation 2 - operation 3) is executed in script 2.

The above discussion raises another interesting topic: how to construct a good (high quality) course (a long operation series) to test more declarative scripts? A simpler answer is: just link all operations together. For example, a course for script 1 and script 2 can be operation A – operation B – operation C – operation E – operation A – operation B – operation C – operation D. This ensures that all scenarios described in the scripts will be validated, but the disadvantage is quite obvious: duplicated operations.

To solve the problem, we can compare and merge declarative scripts. For example, the script 1 and script 2 can be simply merged into “merged script” in Figure 4-6. Ideally, this can solve the problem, but we still have a few practical issues.

Generally speaking, when we transfer a script from imperative to declarative script, the “pre” and “post” will be reused several times with the actions. For example, a script containing “pre”, “act1”, “act2”, “post” will be converted into two scripts: “pre”, “act1”, “post” and “pre”, “act2”, “post” because act1 and act2 are considered as two smallest “validation unit”, and the “pre” and “post” are designed as the general precondition and post-condition of the actions. For one hand, it is possible to combine act1 and act2 together, but doing this will reduce the applicability of this script because it will require more conditions of a scenario. For another reason, the testing cases in each action of a script are logically designed to be independent to each other, so it is pointless to combine them together. In chapter 2.4, we introduced two current issues hampering the merge: the relationships between each actions (ACT1…ACTN in each script) of a script and too many operations taken in “pre” step.

Problem 1. The current script (Figure 2-3) does not guarantee the independence of each action. That means changing an action of a script (remove/add/simply execute) might cause the script fail. For example, consider a script with the following functions: “pre”, “act1”, “act2”, “act3”, “post”. Executing (“pre”, “act2”, “act3”, “post”) or (“pre”, “act2”, “post”) might not get successful result because some of the operations in “act1” is a precondition for “act2” or “act3”. The ideal goal is a script can run successfully with any action(s) between its “pre” and “post”.

Problem 2. Ideally, there should have as few setValue() calls as possible in “pre”. Currently, there are many unnecessary setValues() in “pre”. For example, in function ignition_on(), variable DS_ParkingBrake will be set to “ON” no matter what is the current state of this signal. This is fine in imperative script, but in declarative testing, too many setValue() in “pre” will greatly reduce the applicability of a script when we do the matching to a Mat file.

Although the above issues exist in the current environment and require some time to be solved, they do not block our project at all. If these problems are tackled, we are able to go one step further to build a more flexible, reusable and productive testing platform, and our new test environment gives full possibilities and supports for that.

The new environment is able to provide higher level abstraction and customization. The old environment is constructed with scripts, and the scripts can hardly be combined to build new scripts without modifications. In the new environment, each “pre”, “action” and “post” group makes a “meta validation” which is able to be combined with any other “meta validation”. With its own preconditions and post-conditions, each group forms a self-governing unit. As more imperative scripts are transformed into declarative scripts to create more “meta validations”, we are able to simply build new test scenarios by selecting and combining these existing “meta validations”.

4.2.3 Graphical User Interface (GUI)

The goal of the GUI is to provide information regarding the conflicts detected between a Mat file and a script in a user-friendly visual fashion. What the GUI shows is what a logic analyzer or digital oscilloscope would show when given the values provided as inputs and outputs to the SUT. The source data input to the GUI is a combination of the Mat file and the result of Mat program (the result of step 3 above). The Mat file serves as the base data to give the user a first impression of the general pattern of the signals of interest. If conflicts are found by Mat program, then they will be highlighted in the display of the signals.

The Mat file generally contains a huge number of time-value pairs, where each time-value pair is recorded every 10 milliseconds on the HIL servers. The normal recorded time span of a Mat file is

several minutes. This duration gives sufficient time to perform a satisfactory test course. Therefore, the GUI provides an interface to the analysis tool that analyzes the data collected when executing a script. The Matplotlib was used to implement the GUI. Although it took a lot of work to implement the GUI, its detailed implementation will not be discussed here because the implementation details of GUI are irrelevant to the main topic of this thesis. Chapter 5 gives a comparison of this new GUI with the previous GUI tool used in RESI.

The visual tool needs to be able to clearly show the value changes of a signal over a long period of time, typically a few minutes (as this is the duration of a complete test course). This tool should provide some common built-in functions, such as zoom in, zoom out, and pan, to enable users to focus on specific data. As we wish to investigate the relationship between multiple signals, it is convenient to show all signals of interest in one screen.

The tool should also be able to show the result of running a test script in terms of highlighted errors when conflicts are found by the Mat program. When composing a script, it is common that one signal is ignored when an event should be triggered by two signals as preconditions. However, this is not easy to know which is the ***relevant*** signal among many related signals. Therefore, the GUI module sorts the signals based on a measure of how close the other signals are to the signal responsible for the conflict. The NumPy covariance tool is used to calculate the strength of these relationships.

Figure 4-7 gives an overview of the output of the GUI tool in Mat mode. This figure shows ten signals plotted by the GUI for a time span of 500 seconds. Curves 1 to 5 are shown on the left half of the screen are and curves 6 to 10 on the right half of the screen (with the numbering in each case from top to bottom). The x-axis (horizontal) indicates time, while the y-axis shows the signal's value (scaled to the maximum possible value of this signal). This feature is very important because if a fixed range of y-axis is used in all curves, either some of the curves cannot be displayed completely (for example, if use 0 to 1 range, curve 10 cannot be displayed completely) or the changes of the curves cannot be easily observed (for example, if use 0 to 15 range, the 4 depressed pits within 100 and 300 seconds in curve 2 cannot be easily observed because their y-value is 0.2). Because we are showing the overall curves for a long period of time, it is possible that a signal is changed very frequently, exactly like the two blue rectangles in curve 1 (between 150 and 250 in x-axis).

Some general functions are listed on the left bottom of the screen. The first button is home button. When pressed, the GUI will resume to the initial state. All operations including zoom in, zoom out, drags and drops will be undone. The following two buttons, left arrow and right arrow are used to go back or forward of user operations. When the forth button is pressed, all diagrams are enabled to drag and drop with mouse click and release. The fifth button is used to zoom in or out on a single curve. The sixth button is used to configure the layout of the GUI, such as the space between two curves, the indent of each curve and so on. The last button is used to take a screen shot and save it as a JPG file. The x and y values indicated at the right down corner shows the actual values for which ever curve the mouse is in.

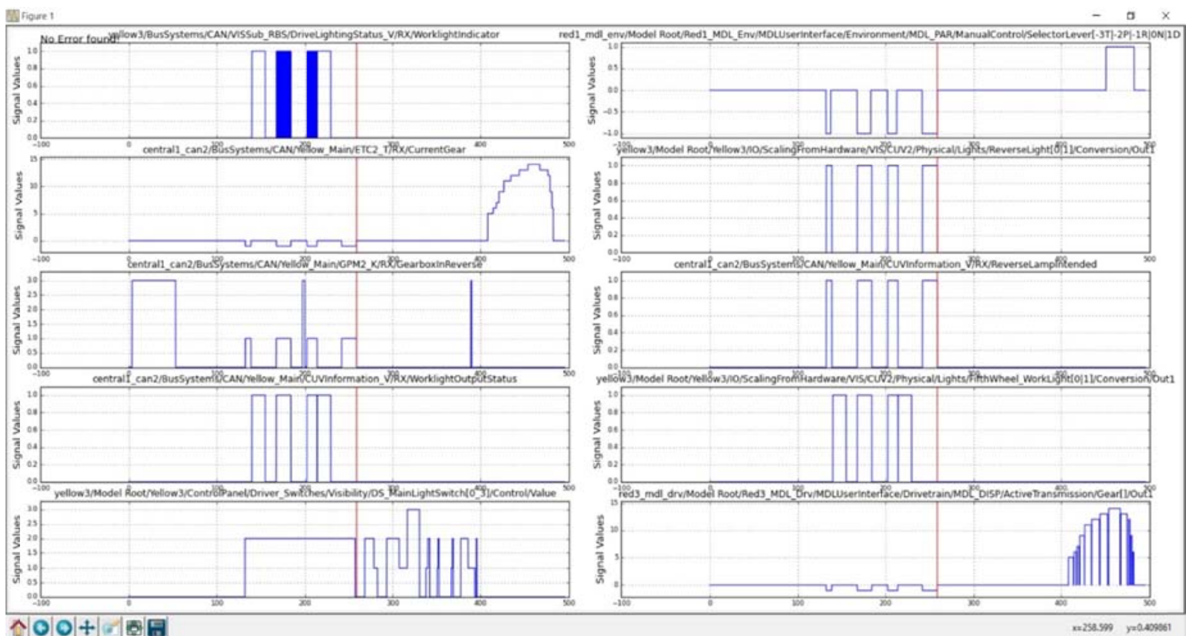

Figure 4-7: **General view of GUI, Mat mode**

The vertical red line helps to identify a specific value on each plotted signal at a fixed time point. It will follow the movements of a cursor dragged over it. All plotted areas can be moved horizontally by mouse using drag and drop. An important feature is that the x-axis (time value) of all plotted areas are linked, this ensures that all plotted areas are displaying the same time span and the signal value the red line indicates is always at the same time point for all curves. This feature is preserved when zooming in/out, as shown in Figure 4-8, thus all of the plots will be zoomed together.

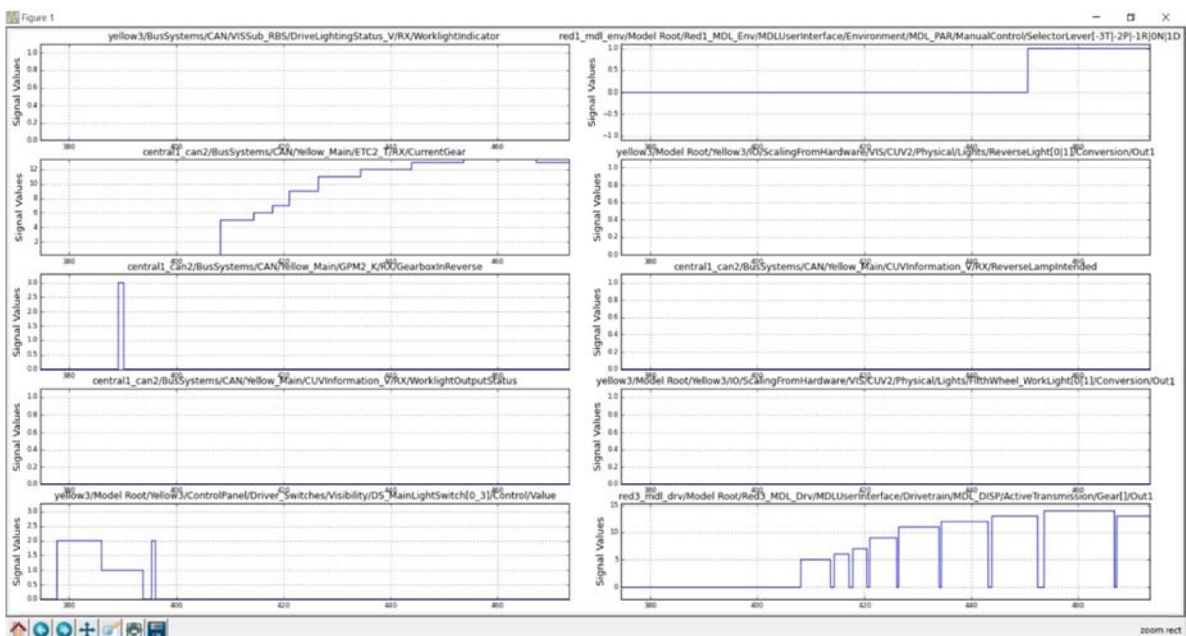

Figure 4-8: **Zoom in**

If a change in signal A results in a change of signal B, then signal A must change earlier than signal B, then we say that signal A is active and signal B passive.

4.2.4 A real case

Figure 4-9 demonstrates a real case when conflicts were found when a new script was being composed. The red highlighted portion of the curves indicates a conflict. When ctrl+left or ctrl+right is pressed, the vertical red line will move among the conflicts to give an accurate location of when in time this specific conflict began and will enable the user to see what the state of each of the curves is at that point in time. After browsing all of the signals we found that the signals 3 and 9 (the 4th signal from the top on the right side) are most likely responsibility for the conflict because their change covers the conflict's duration (i.e., time span), but the other curves seem to be behaving normally. It is not hard to see that the changes in signal 3 completely overlap the duration of the conflict, which makes it a *passive* signal. Our conclusion is that signal 9 (which changed prior to signal 3) is the trigger of this conflict. When re-examining this script, we found that we forgot to set the "DS_ParkingBrake" variable, which is exactly the name of signal 9, to "0" before making the assertions.

It should be highlighted that the above reasoning is purely based on the curve shown in the GUI, but in practical use, it is also very helpful to do the reasoning after understanding the meanings and the relationships of the indicators. For example, the signal 3 ("GearboxInReverse") has the following possible values and meanings: value 0 means "GearboxNotInReverse", value 1 means "GearboxInReverse", values 2 means "Error", and value 3 means "NotAvailable". Sometimes it is possible and easy to locate the time of an error solely based on one signal. For example, the value of signal 3 is "Error", which should not occur if the vehicle is in a correct or normal state, indicates that there must have been a problem during the specific period of time.

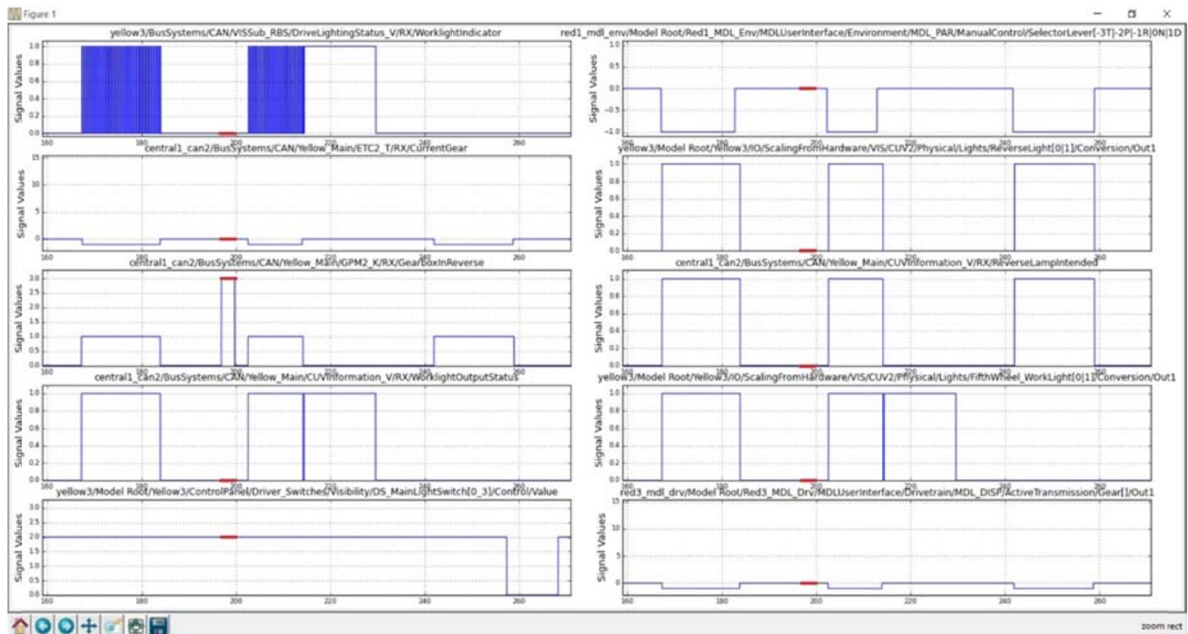

Figure 4-9: Plot with conflicts

5 Evaluation

This chapter introduces the criteria used to evaluate the differences between the new testing environment and the existing one. In Table 3-1 of Section 3.2 a priority list was introduced. The new environment is required to fulfill all priories on the left of the table and satisfy as many as possible of those on the right of the table. The evaluation in this chapter will be mainly based upon whether the new environment achieves the requirements shown in this list of priorities.

While some of the desirable properties were already provided in the old environment, for some reasons they are either unused or not easy to use. For this reason, the metrics used in the evaluation of this thesis project were selected considering two aspects: completeness (i.e., fulfilling the requirement or not) and ease of use. For some aspects such as the GUI, it is hard to determine which alternative is better because each user has his or her own preferences, therefore the comparison will be carried out based upon specific use cases in a number of different scenarios and the detailed operations will be listed and compared.

All experiments are run on a computer with Intel Core i7-3740 (Quad Core), 16 GB RAM, and Microsoft's Windows 7 64bit operating system.

5.1 Offline debugging

This function is very important in the new system because it forms the basis of many other functions. Because the new environment also uses the Pylint as a static code analysis tool in exactly the same way as in the old environment, we will not compare this part. The greatest improvement we made in the new environment is that we are able to find run-time exceptions. Python has a few types of exceptions, such as AssertionError, AttributeError, and so on. We selected 15 of the most common types of errors and deliberately put them into the script we are going to investigate. These errors are shown in Table 5-1.

On the other hand, due to the limited time for this project, we randomly selected 10 scripts (out of 110 scripts, accounting for 9% of the scripts) to validate the new environment. The selected scripts are: TC0001, TC0002, TC0004, TC0005, TC0014, TC0016, TC0023, TC0037, TC0066 and TC0076.

The result is shown in the following two sections in terms of two indicators: run time and error-detection rate.

Table 5-1: Exceptions

Exception Name	Exception Description
AssertionError	It is raised when a failed assert() is triggered.
AttributeError	It is raised when an attribute reference or assignment fails. For example, when trying to reference to a non-exist attribute of an object.
EOFError	It is raised when a built-in Python function encounter the end of the input stream before reading any data.
IOError	It is raised when an IO error happens, such as disk full, file not exist.
IndexError	This error is raised when referencing an item in the list with an index which is beyond the range of the list.
KeyError	This error is raised when trying to retrieve a value with a key which does not exist in the dictionary.
MemoryError	This error is raised when the program runs out of memory.
NameError	This error is raised when a non-exist name is referenced within the current naming scope.
NotImplementedError	If a method or behavior is required to be implemented but ignored by a subclass.
ReferenceError	When an object is referenced after it has been garbage collected, a ReferenceError will be raised.
StopIteration	StopIteration is raised on calling next() function of iterators when it has reached the end.
SyntaxError	This error is raised when the Python parser cannot understand the source code. For example, calling eval("two plus five") will raise this error.
TypeError	This error is raised when the type is mismatched or used in a wrong fashion. For example, print (3)+"six" will raise this problem.
ValueError	This error is raised when the value is matched but the value is incorrect.
ZeroDivitionError	This error is raised when 0 is used as a denominator of a divided operation

5.1.1 Run time

The reason why we have considered run time here is to investigate how much faster we can find a bug in a script compared to a normal test (run against HIL) of a script. Normally, if we run a script against real HIL environment, it will take some time, t. During this t time, the HIL environment will only test and validate one running path. It is time consuming and difficult to test all paths to get a total time, so in order to compare to the Dummy mode, we only calculate a minimal running time of each script. The minimal running time is measured in this way: we pre-assume the script will terminate successfully and for the path it executes, we will accumulate the time. For example, if a script has an operation time.sleep(10) and there is no way of avoiding the execution of this

operation unless the script fails, then 10 seconds will be added to the accumulated running time of this script.

For the Dummy mode, similar operations (sleep/wait operations) will be skipped because they are mainly used for the HIL hardware to respond, while Dummy mode is running purely in software. In Table 5-2 the running time for the selected 10 scripts is given and compared with the minimal running time on HIL.

Table 5-2: Running time comparison: HIL hardware and Dummy mode

Script	Minimal Running Time (seconds)	Dummy Running Time (seconds)	Saved Time
TC0001	2.0	1.85	7.5%
TC0002	4.5	1.96	56.5%
TC0004	26.0	7.88	70.0%
TC0005	12.0	5.96	50.4%
TC0014	3.0	1.80	40.0%
TC0016	3.6	1.56	57.0%
TC0023	10.0	4.30	57.0%
TC0037	10.0	2.10	21.0%
TC0066	10.0	6.50	65.0%
TC0076	124.5	33.20	73.3%

It can be seen from the table that for all 10 scripts Dummy mode takes less time and is able to test more test cases. For the least time saving case (TC0001) Dummy saves 7.5% time and for the most time saving case (TC0076) it saves 73.3%. The average savings in time for all 10 scripts is 52.6%.

5.1.2 Error Detection Rate

As stated before, we randomly inserted errors into different code segment of each script and test if these errors were found. The results (shown in Table 5-3) shows that almost all errors can be found in Dummy mode (99.3%), but can only partially be found when the script is executed in HIL (81.3%). In fact, the two groups are not comparable because group A is not designed to be used for error detection, but from another perspective, group A can, to some extent, represent a normal bug detection rate when the script is finished and tested against the HIL environment. It should be noted that the result in group A is collected in a simulated way which presumes the script will terminate successfully and all assertions within the script will hold.

After examined the scripts we found the test coverage is the most influential factor in the difference between the two groups. Because the goal of the test scripts is to validate the functionality of the production instead of making an error-tolerance program, most of the code serves the main branch (which is also the branch that will run successfully) and as a result, the error-detection rate is rather reasonable.

We also analyzed the fact that one error is not being detected in Dummy mode (script TC0076). Based on the context, the system will generate a log message after the program has waited for 15 seconds. As stated before, Dummy mode will not execute the wait() for any reason and that is why this error cannot be found. Admittedly, this might be a problem in logic of Dummy mode, but in

practical use this is not a problem, because the program will not behave differently based on the waiting time.

Table 5-3: Errors Detected

Script	Errors Encountered in HIL (group A)	Errors Detected in Dummy (group B)	Total Errors
TC0001	9	15	15
TC0002	15	15	15
TC0004	11	15	15
TC0005	13	15	15
TC0014	11	15	15
TC0016	12	15	15
TC0023	13	15	15
TC0037	15	15	15
TC0066	15	15	15
TC0076	8	14	15

We noticed that the bug detection rate in group A is not as high as group B, but as a default group with only one execution path, it is quite high. After investigating the scripts, we found that this occurs because the scripts are relative simple and do not contain complicated error-handling code. That is due to the nature of the script, as they are designed to find bugs of other systems, rather than being designed to be error-tolerant themselves. For example, the try/catch block is rarely used in our scripts, while it is quite common in Python applications.

5.1.3 Ease of use: when an error is detected

The following information is taken from an email from Jenkins when an error occurred.

```
1    Exception Traceback:
2      File "run_dummy.py", line 19, in <module>
3        tc.execute()
4      File "C:\Users\kevinyeoh\OneDrive\Code\CodesInScania\main.R2014\Interface\exec
5    ute.py", line 314, in execute
6        self.execute_act(actname)
7      File "C:\Users\kevinyeoh\OneDrive\Code\CodesInScania\main.R2014\Interface\exec
8    ute.py", line 171, in execute_act
9        actRunner(self, act_name)
10     File "C:\Users\kevinyeoh\OneDrive\Code\CodesInScania\main.R2014\Interface\exec
11   ute.py", line 160, in actRunner
12       getattr(obj, act)()
13     File "C:\Users\kevinyeoh\OneDrive\Code\CodesInScania\TC_NCG\TC0005_ReverseLigh
14   tActivationWithWorkingLights_Simple.py", line 110, in act1
15       print(spam[6])
16   IndexError: list index out of range
17   input serial:
18   [('OFF', 0), ('TEMP_VALUE', 250), ('TEMP_Manual', 1), ('TEMP_Manual', 1)]
19
```

```
20      Exceptions found:1
21      --------------------------------------------------------------------------------
22
23      total paths tested:1
```

Lines 1 to 16 show the error messages generated by Python. Lines 17 and 18 indicate what input values triggered this error. This information helps to analyze and reconstruct the problematic environment. Line 20 shows how many errors were found during the test and the total number of paths tested is displayed at the end.

To locate the errors, the old environment relies on log files which makes the debugging take quite a long time and is inconvenient. These error messages are located on HIL servers and the user maybe need to search several few log files to find the error message. In the new environment, error messages are immediately returned when exceptions are encountered along with the environment information. When a bug is found, everyone in the team will receive an email containing the error message.

5.1.4 Ease of use: debugging with GUI in Mat mode

In this chapter, the two test environments are compared based on different use cases.

The old environment provides a tool called captureVisualization. This tool is mainly used to read Mat files and display multiple signals, but it has nothing to do with the scripts. In other words, this tool is used for assistance or visualization when composing or debugging scripts. However, it cannot help to find any problem based on the running result of the scripts.

1. Plot multiple curves

Both environments can plot multiple curves, as shown in Figure 5-1 and Figure 5-2.

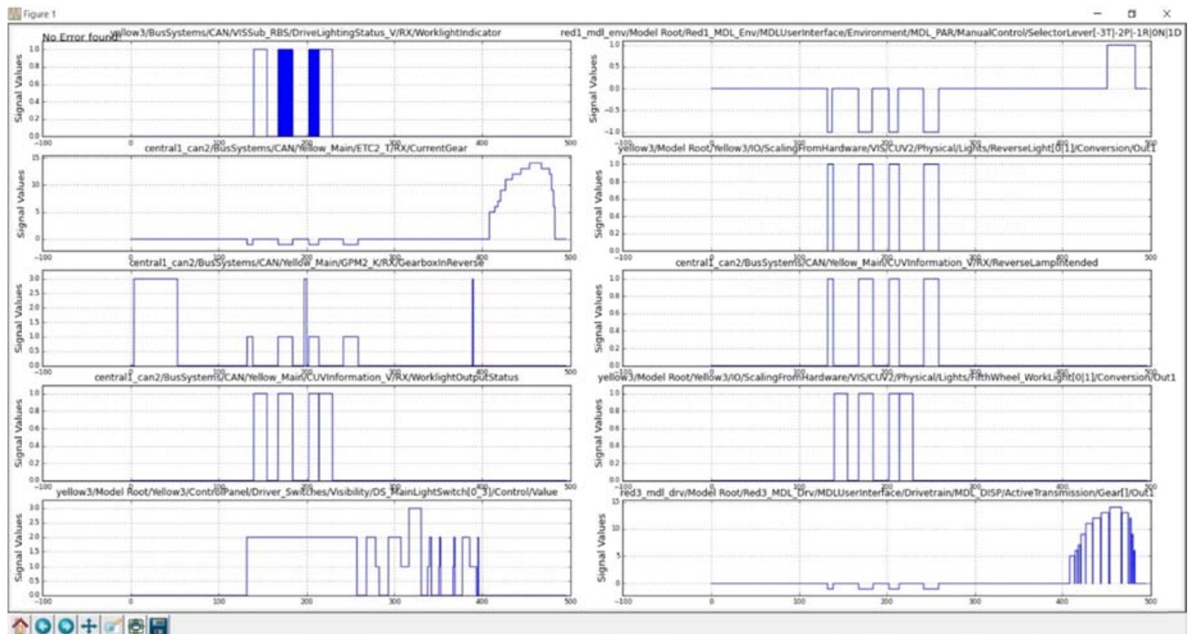

Figure 5-1: Multiple curves plotted in the new environment

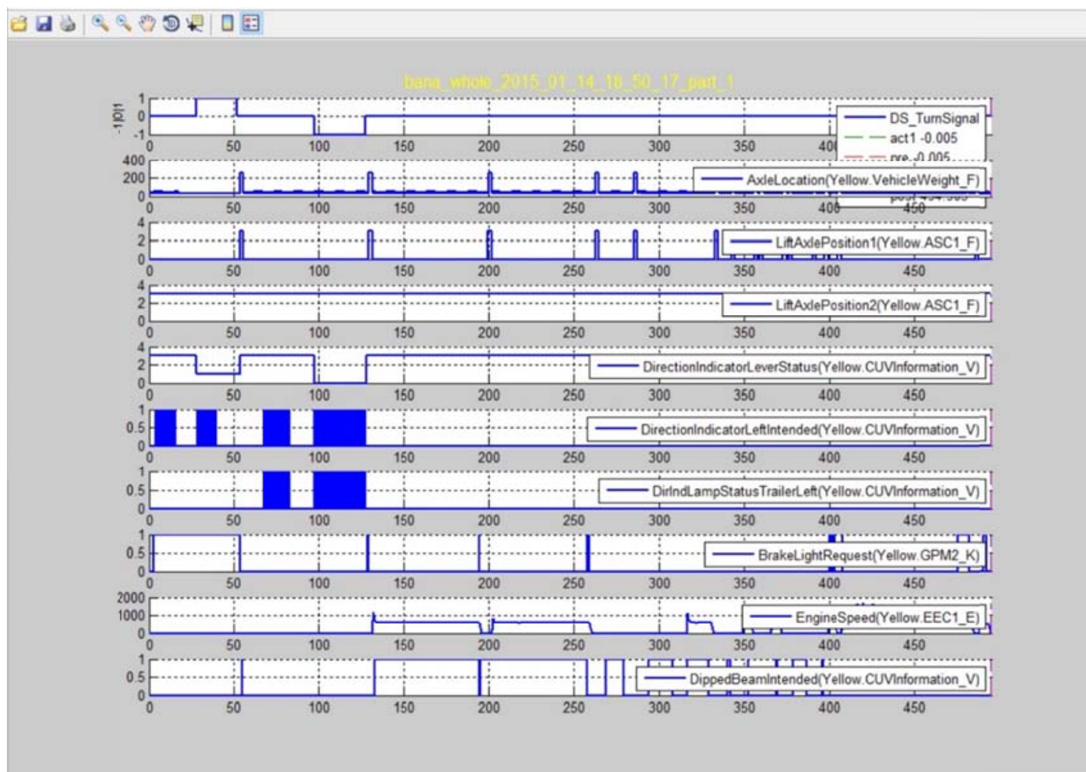

Figure 5-2: Multiple curves plotted in the old environment

The layout of the plotted curves in the old environment makes it hard to read the values, and there is no way to change the layout. For the new environment, it is easier to observe the general curve and read a single value.

2. Zoom in and out at a specific area

Both two environments provide convenient functions to zoom in and out, as shown in Figure 5-3 and Figure 5-4.

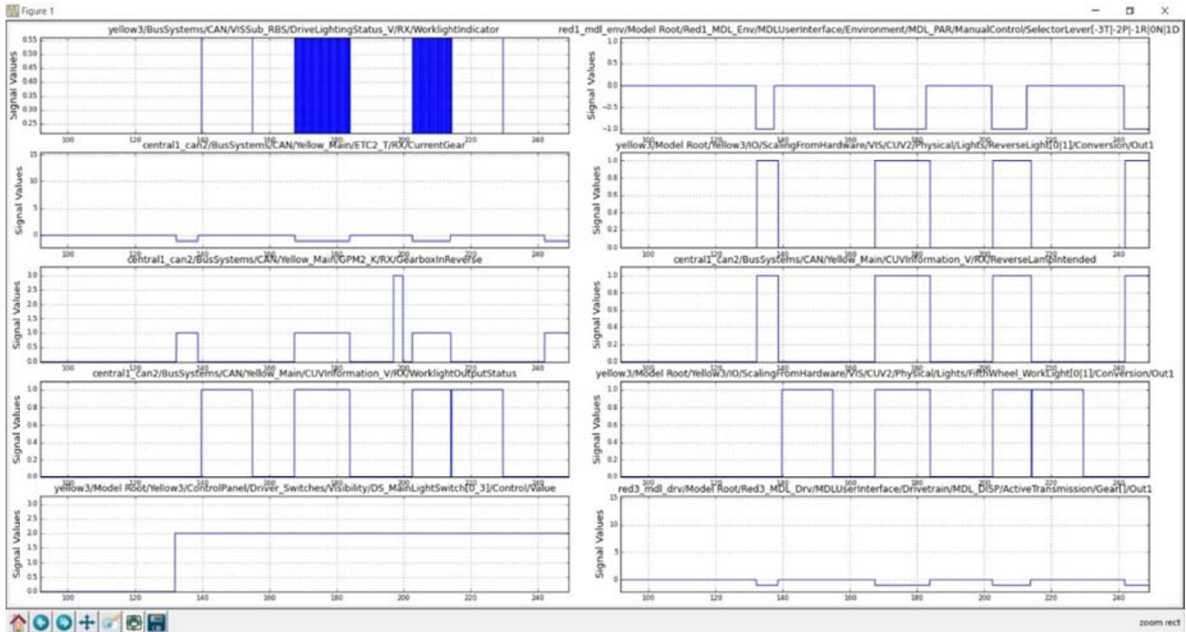

Figure 5-3: **Zoom in at a specific area-new environment**

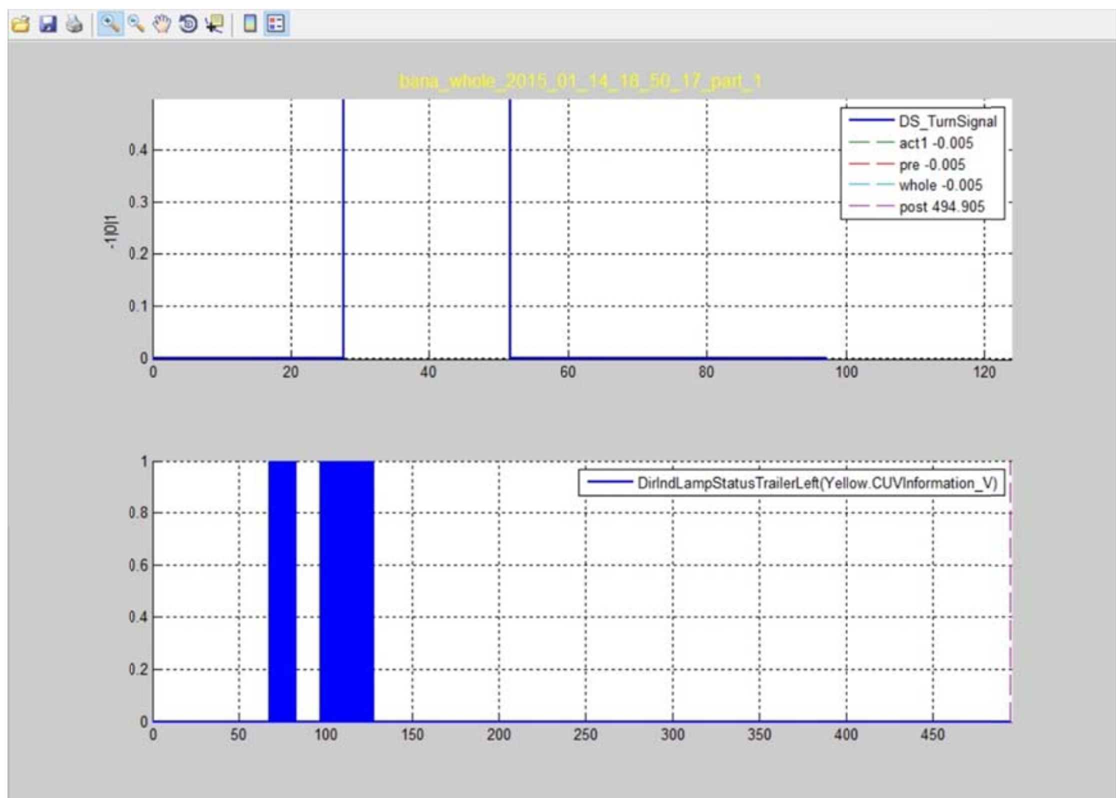

Figure 5-4: **Zoom in to a specific area-old environment**

We zoomed in on the first curve on both environments. It can be seen from the figure that for the old environment the x-axis of all curves is not zoomed in at a same scale (curve1 zoomed in 200% to 125 seconds, but curve2 remains at 500 seconds). As we stated before we aim to investigate the relationship between signals, and an important factor is to do this is to know the

values of a signal at a certain time point. After zooming in, the new environment facilitates this, but this is impossible for the old environment.

3. Zoom in and out on only x or y axis

   Zoom in solely on x-axis (time axis) is very useful when we are switching between a larger view (to investigate the relationship between signals) and a detailed view (to concentrate on the signal value at a specific time point). Because the signal value (y-axis) fluctuates within a fixed small range of numbers, zooming in on the y-axis will result in a non-continuous plot, and there is no way to see the value of a signal, as can be seen in Figure 5-6. In contrast, the new environment is able to zoom in solely on x-axis, as was shown in Figure 5-5.

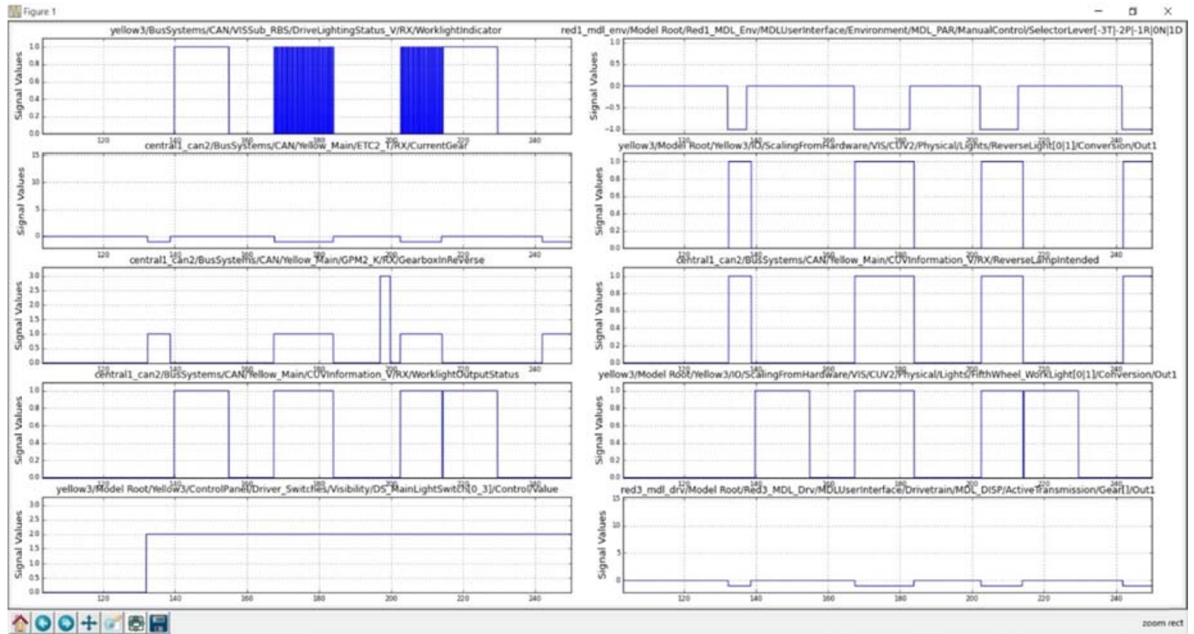

Figure 5-5: **Zoom in-new environment**

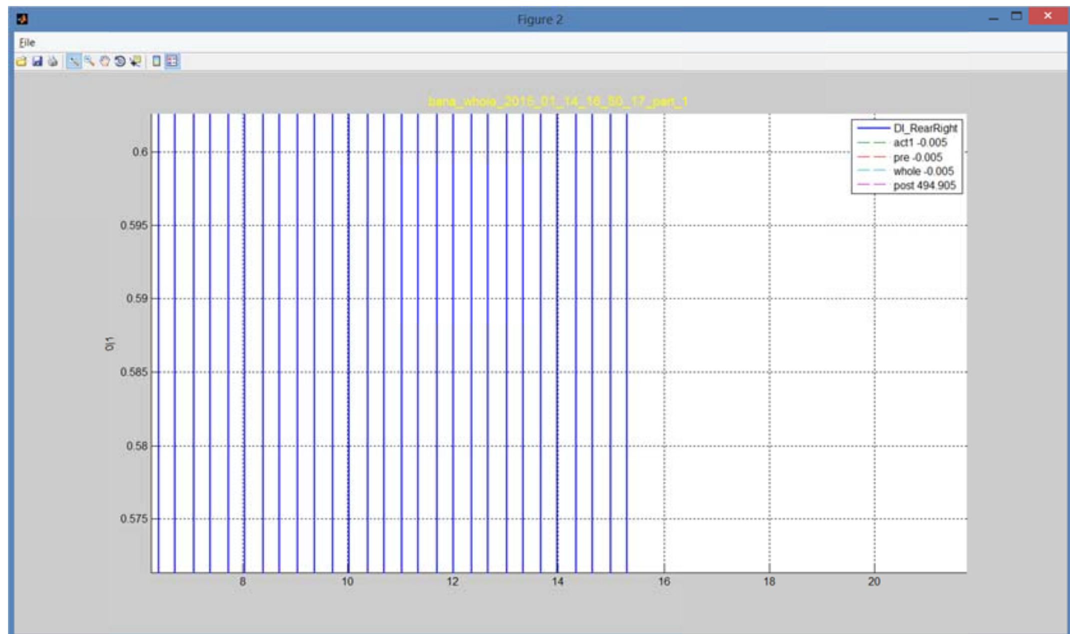

Figure 5-6: **Zoom in-old environment**

4. Order curves based on the degree of how close they are related (coefficient)

   Both environments have this feature. By ticking the "use coefficient" on the right down corner, the old environment can order the curves based on their coefficient (see Figure 5-7). This feature is enabled by default in the new environment, and if errors are detected, the curves will be ordered based on the error values within this time period (time range from the first error to the last error, as shown in Figure 4-9)

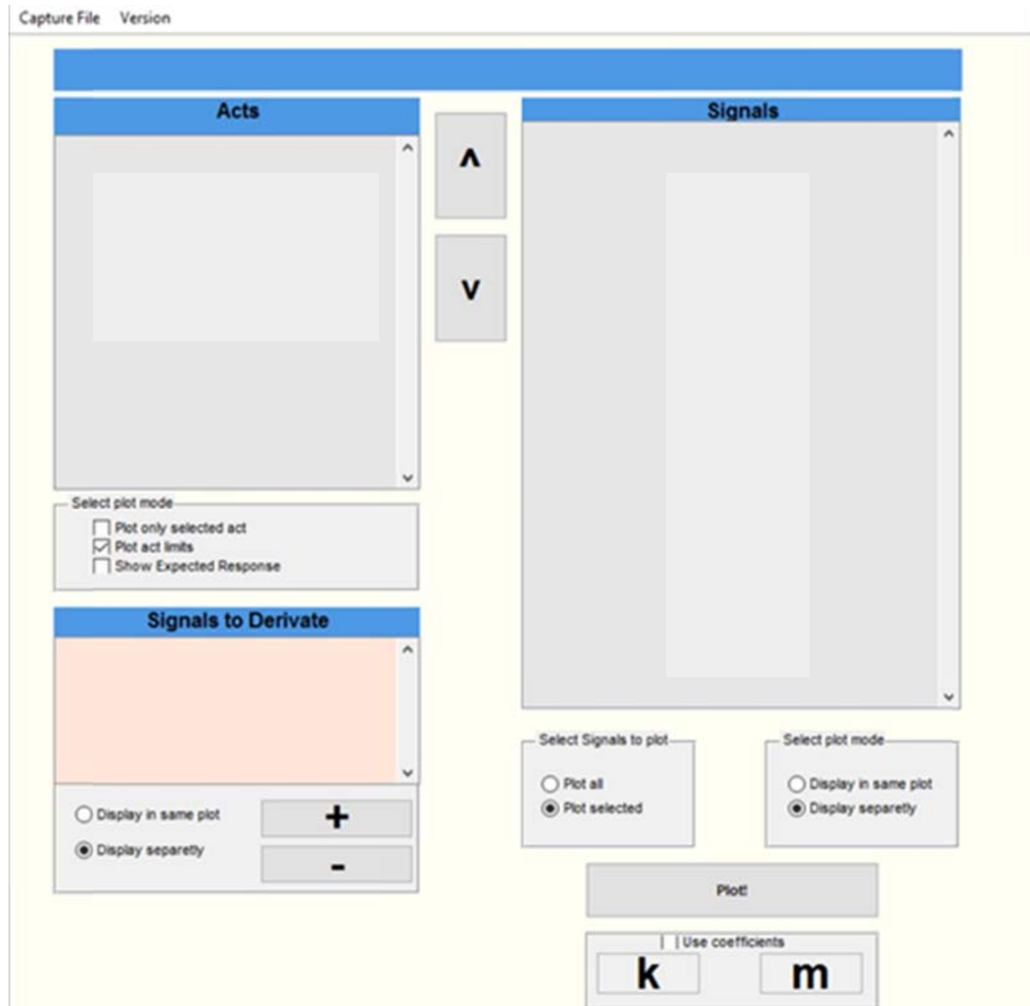

Figure 5-7: Coefficient setting-old environment

5. Show and locate conflicts of a script based upon a Mat-file

   The old environment can only read a Mat file and display it. It does not provide any functions related to the script that was being executed. The new environment displays conflicts in an integrated fashion (shown in Section 4.2.4), thus enabling the test script writer to better investigate them.

5.2 Efficient static analysis tool

PyLint is a very good tool for static analysis. In the new environment it is used in exactly the same way as in the old environment, hence there is no difference between them.

5.3 Be able to run automatically

Dummy mode can be easily triggered by Jenkins continuous integration system. We use unittest module in Python to trigger the Dummy mode. The unittest-xml-reporting package is used to output the result in XML and served as a bridge between Jenkins and our programs.

6 Conclusions and Future work

In this chapter, this thesis project is finalized by demonstrating the outcomes and proposing future work.

6.1 Conclusions

Based on the discussion in Chapter 5 we can draw some conclusions. The new environment provides a unified solution to improve the experience of writing a test script when developing a test script. Two tools, Dummy mode and Mat mode is designed and implemented by myself. In Dummy mode, it is possible to identify many different types of runtime exceptions within a short time. In Mat mode, with the help of Mat files we can detect logic errors of the test script ***without*** being connecting to the HIL servers. A total number of 2600 lines of code is used to implement the whole project (excluding ~800 lines of code used for evaluation).

In comparison to the old environment, the new environment is able to increase the efficiency of using HIL servers by means of reuse and reproduce the result of the signal. The new environment is better in terms of shorter running time and better support for investigating conflicts and errors offline.

The original goals of this master thesis project (from Section 1.4) have been met. The detailed evaluation of the new integration testing environment was given in Chapter 5. In general, most targets have been achieved as planned, except for an optional requirement (test script management). Overall, a new script testing environment is implemented. 10 out of 110 scripts are tested in Dummy mode and a new script is composed to evaluate the Mat mode. Finally, the new environment (Dummy mode part) is being integrated into the current integration testing environment of RESI in Scania and is expected to facilitate the development cycle.

I benefited a lot from conducting this project. By having a chance to work in Scania's RESI department, I experienced the most advanced HIL laboratory in Sweden. This cutting edge server can greatly benefit the development and test cycle in terms of saving time and increasing productivity. However, if we want to take this one step further, further integration and customization is also very important.

6.2 Limitations

Due to the limited curation of this thesis project, the trigger sequence was not considered. Figure 6-1 shows an example (highlighted by the red circle), where signals 1, 2, and 3 are triggered sequentially. Because the mechanism we are using in the Mat mode to detect if two signals are both triggered (equals to 1, for example) at a certain point of time is to validate their values at this time (in the picture, the time between each pair of green lines), the values before or after this point of time (the time outside each pair of green lines) are all ignored.

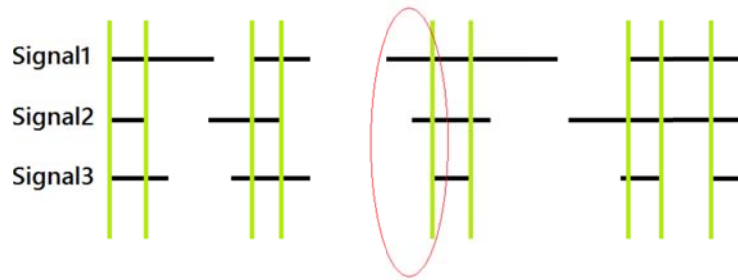

Figure 6-1: Trigger sequence

6.3 Future work

The new environment can easily be integrated into a Jenkins environment, as shown in the large gray rectangle in Figure 6-2. However, the elements in the upper right hand corner (the "GUI Script Design" and "GUI Course Design") were not implemented in this thesis project. Their implementation would offer great help to create longer and better-designed courses, and in return, would benefit the composing of new scripts.

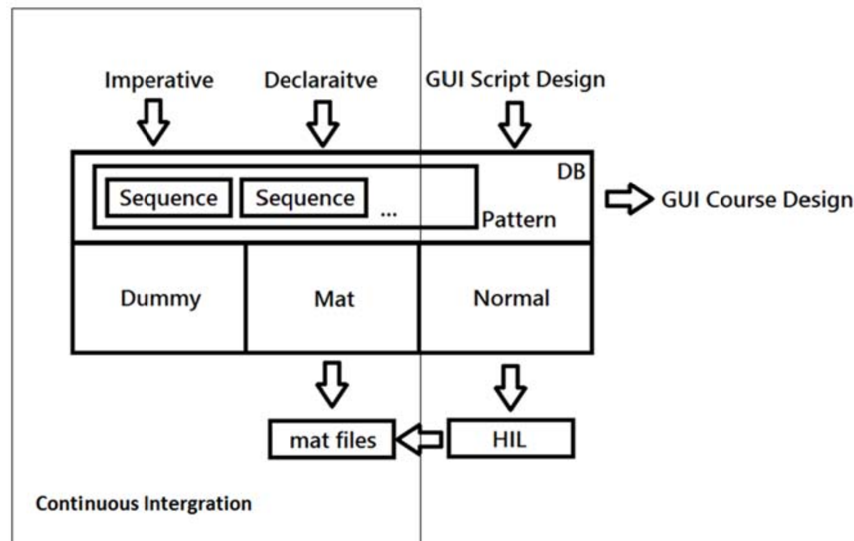

Figure 6-2: Future continuous integration testing

6.4 Reflections

The project focused on integrating the testing environment into a more stable and effective test environment. We are able to increase the coverage rate of each test script and to find problems as early as we can. From a more general view, the thesis aims at simplifying the debugging of the scripts and increasing the possibility of finding a bug. Visualization of the data also enables us to provide more useful information to our users. As a consequence, the users are able to debug their scripts offline and are given more chance to find a bug before their script go online to HIL environment.

From an economic perspective, this thesis provides guidance to test environment developers in terms of designing and implementing a new automatic test environment based on an existing one. The proposed solutions have been proven to be effective and productive in our project, which

provide very useful information for other similar projects. The new environment can also save some time comparing to the old one, which also saves money (i.e., is an economic benefit).

Maximizing the utilization of existing test scripts is also considered during the project. All existing scripts can benefit from the new environment with very limited changes, which is also a positive economic effect for Scania.

The positive social effect of this project is we are able to increase the satisfaction of the test script composers. In the old environment, they need to wait in queue to use the HIL, find useful information in piles of log files, correct the script, and the continue the development loop. In the new environment, they are able to finish part of the job locally in a more convenient way.

References

[1] The MathWorks Inc., *MATLAB Runtime Version 8.2 (R2013b) 64bit*. Natick, Massachusetts: The MathWorks Inc., 2010 [Online]. Available: http://se.mathworks.com/supportfiles/downloads/R2013b/deployment_files/R2013b/installers/win64/MCR_R2013b_win64_installer.exe

[2] Jim A. Ledin, 'Hardware-in-the-Loop Simulation', *Embedded Systems Programming*, vol. 12, no. 2, pp. 42–60, Feb. 1999.

[3] Susanne Köhl and Dirk Jegminat, 'How to Do Hardware-in-the-Loop Simulation Right', SAE International, Warrendale, PA, 2005-01-1657, Apr. 2005 [Online]. Available: http://www.sae.org/technical/papers/2005-01-1657. [Accessed: 15-Jun-2015]

[4] ASAM HIL workgroup, 'Application Programming Interface for ECU Testing via Hardware-in-the-Loop Simulation', Association for Standardisation of Automation and Measuring Systems, Programmers Guide: Part 1 of 4 Version 1.0.0 Base standard, Jul. 2009 [Online]. Available: http://wiki.simwb.com/swbwiki/swbdoc/docs/UserManuals/PYToolkit/ASAM_AE_HIL_BS-1-4_API-for-ECU-Testing-via-HIL-Simulation_V1-0-0.pdf

[5] Amirhossein Chinaei, 'Programming Languages: Chapter 1', Department of Electrical and Computer Engineering. University of Puerto Rico Mayagüez, Sping-2010 [Online]. Available: http://ece.uprm.edu/~ahchinaei/courses/2010jan/icom4036/slides/03icom4036Intro.pdf

[6] Lutz Prechelt, 'An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl', Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, Technical Report 2000-5, Mar. 2000 [Online]. Available: http://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt_computer2000.pdf

[7] L. Prechelt, 'An empirical comparison of seven programming languages', *Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000. DOI: 10.1109/2.876288

[8] Diomidis Spinellis, Vassilios Karakoidas, and Panos Louridas, 'Comparative language fuzz testing: programming languages vs. fat fingers', in *PLATEAU '12 Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, New York, NY, USA, 2012, pp. 25–34 [Online]. DOI: 10.1145/2414721.2414727

[9] National Instruments, 'Understanding Client-Server Applications -- Part 1', National Instruments, White paper NI-Tutorial-4431, Oct. 2011 [Online]. Available: http://www.ni.com/white-paper/4431/en/pdf

[10] Thomas Gustafsson, Mats Skoglund, Avenir Kobetski, and Daniel Sundmark, 'Automotive system testing by independent guarded assertions', presented at the IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Graz, Austria, 2015, pp. 1–7 [Online]. DOI: 10.1109/ICSTW.2015.7107474

[11] Paul Ammann and Jeff Offutt, 'Introduction to software testing'. New York: Cambridge University Press, 2008, ISBN: 978-0-521-88038-1.

[12] Collin Winter and Tony Lownds, 'PEP 3107 -- Function Annotations', 02-Dec-2006. [Online]. Available: https://www.python.org/dev/peps/pep-3107/. [Accessed: 15-Jun-2015]

[13] James C. King, 'Symbolic execution and program testing', Communications of the ACM, volume 19, number 7, 1976, 385--394

[14] Anand Saswat, Patrice Godefroid, Nikolai Tillmann (2008). "Demand-Driven Compositional Symbolic Execution". Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 4963: 367–381.

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.", Proceedings of the 8th USENIX conference on Operating systems design and implementation OSDI'08. USENIX Association Berkeley, CA, USA. 2008.

[16] Samir Sapra, et al. "Finding errors in python programs using dynamic symbolic execution." Testing Software and Systems. Volume 8254 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013. 283-289, doi: 10.1007/978-3-642-41707-8_20.

[17] Hosam K. Fathy, et al. "Review of Hardware-in-the-Loop Simulation and Its Prospects in the Automotive Area", Proceedings of SPIE - The International Society for Optical Engineering (Impact Factor: 0.2). 05/2006; 6228. DOI: 10.1117/12.667794. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.458.9760&rep=rep1&type=pdf

[18] Cătălin Vasiliu and Nicolae Vasile, " Hardware-in-the-loop Simulation for Electric Powertrains", Revue Romaine des Science Techniques, No. 2/ 2012, Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.2976&rep=rep1&type=pdf

[19] David Fischer, "Test Cases for Hardware in The Loop Testing of Air To Water Heat Pump Systems in A Smart Grid Context", Available: http://www.greenhp.eu/work-packages/wp10-smart-grid/?eID=dam_frontend_push&docID=2582

[20] Daniel Simon, "Hardware-in-the-loop test-bed of an Unmanned Aerial Vehicle using Orccad", 6th National Conference on Control Architectures of Robots, May 2011, Grenoble, France. 14p., 2011.<inria-00599685> Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.220.4932&rep=rep1&type=pdf or https://hal.inria.fr/inria-00599685/file/Simon-hardware.PDF

[21] F. Mosnier and J. Bortolazzi. Prototyping car-embedded applications. In Advances in Information Technologies: The Business Challenge, pages 744–751. IOS Press, Amsterdam, The Netherlands, 1997.

[22] Andreas Bayha, Franziska Grüneis, and Bernhard Schätz. "Model-Based Software In-the-Loop-Test of Autonomous Systems", Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium. SCS/ACM 2012. Available: https://www4.in.tum.de/~schaetz/papers/mod4sim.pdf

[23] Peter Waeltermann, Thomas Michalsky, and Johannes Held. "Hardware-in-the-loop Testing in Racing Application", SSAE Motor Sport Engineering Conference and Exhibition, 2004. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.474.4087&rep=rep1&type=pdf

Appendix A: Detailed results

The transferred script log:

SEQ: TYPE:31,

Name:set_neutral,

Value:None,Va:None,Action:0 with parallel of 0

False

SEQ: TYPE:2,

Name:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_MainL

ightSwitch[0_3]/Control/Value,

Value:2,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:2,

Name:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_Workl

ightSwitch[0|1]/Control/Value,

Value:1,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:3,

Name:/wait_event,

Value:0,Va:0,Action:0 with parallel of 1

subevent:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_W

orklightSwitch[0|1]/Control/Value,0,None,23

False

SEQ: TYPE:2,

Name:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_Workl

ightSwitch[0|1]/Control/Value,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:3,

Name:/wait_event,

Value:0,Va:0,Action:0 with parallel of 1

subevent:yellow3/Model Root/Yellow3/ControlPanel/Driver_Switches/Visibility/DS_W

orklightSwitch[0|1]/Control/Value,0,None,23

False

SEQ: TYPE:3,

Name:/wait_event,

Value:0,Va:0,Action:0 with parallel of 2

subevent:red1_mdl_env/Model Root/Red1_MDL_Env/MDLUserInterface/Environment/MDL_P

AR/ManualControl/Pos_ClutchPedal[%]/Value,0,None,23

subevent:central1_can1/BusSystems/CAN/Red_Main/TCProp_K/RX/ClutchPedalPosition,0

,None,23

False

SEQ: TYPE:31,

Name:set_neutral,

Value:None,Va:None,Action:0 with parallel of 0

False

SEQ: TYPE:31,

Name:set_neutral,

Value:None,Va:None,Action:0 with parallel of 0

False

SEQ: TYPE:3,

Name:/wait_event,

Value:0,Va:0,Action:0 with parallel of 4

subevent:red3_mdl_drv/Model Root/Red3_MDL_Drv/MDLUserInterface/Drivetrain/MDL_DI

SP/ActiveTransmission/Gear[]/Out1,0,None,21

subevent:central1_can2/BusSystems/CAN/Yellow_Main/GPM2_K/RX/GearboxInReverse,0,N

one,23

subevent:yellow3/Model Root/Yellow3/IO/ScalingFromHardware/VIS/CUV2/Physical/Lig

hts/ReverseLight[0|1]/Conversion/Out1,0,None,23

subevent:yellow3/BusSystems/CAN/VISSub_RBS/DriveLightingStatus_V/RX/WorklightInd

icator,0,None,23

False

SEQ: TYPE:1,

Name:red3_mdl_drv/Model Root/Red3_MDL_Drv/MDLUserInterface/Drivetrain/MDL_DISP/A

ctiveTransmission/Gear[]/Out1,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:red3_mdl_drv/Model Root/Red3_MDL_Drv/MDLUserInterface/Drivetrain/MDL_DISP/

ActiveTransmission/Gear[]/Out1,Va:0,Action:0 with parallel of 0

False

SEQ: TYPE:1,

Name:central1_can2/BusSystems/CAN/Yellow_Main/ETC2_T/RX/CurrentGear,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:central1_can2/BusSystems/CAN/Yellow_Main/ETC2_T/RX/CurrentGear,Va:0,Action

:0 with parallel of 0

False

SEQ: TYPE:1,

Name:yellow3/Model Root/Yellow3/IO/ScalingFromHardware/VIS/CUV2/Physical/Lights/

ReverseLight[0|1]/Conversion/Out1,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:yellow3/Model Root/Yellow3/IO/ScalingFromHardware/VIS/CUV2/Physical/Lights

/ReverseLight[0|1]/Conversion/Out1,Va:0,Action:0 with parallel of 0

False

SEQ: TYPE:1,

Name:central1_can2/BusSystems/CAN/Yellow_Main/GPM2_K/RX/GearboxInReverse,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:central1_can2/BusSystems/CAN/Yellow_Main/GPM2_K/RX/GearboxInReverse,Va:0,A

ction:0 with parallel of 0

False

SEQ: TYPE:1,

Name:central1_can2/BusSystems/CAN/Yellow_Main/CUVInformation_V/RX/ReverseLampInt

ended,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:central1_can2/BusSystems/CAN/Yellow_Main/CUVInformation_V/RX/ReverseLampIn

tended,Va:0,Action:0 with parallel of 0

False

SEQ: TYPE:1,

Name:yellow3/BusSystems/CAN/VISSub_RBS/DriveLightingStatus_V/RX/WorklightIndicat

or,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:yellow3/BusSystems/CAN/VISSub_RBS/DriveLightingStatus_V/RX/WorklightIndica

tor,Va:0,Action:0 with parallel of 0

False

SEQ: TYPE:1,

Name:central1_can2/BusSystems/CAN/Yellow_Main/CUVInformation_V/RX/WorklightOutpu

tStatus,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:central1_can2/BusSystems/CAN/Yellow_Main/CUVInformation_V/RX/WorklightOutp

utStatus,Va:0,Action:0 with parallel of 0

False

SEQ: TYPE:1,

Name:yellow3/Model Root/Yellow3/IO/ScalingFromHardware/VIS/CUV2/Physical/Lights/

FifthWheel_WorkLight[0|1]/Conversion/Out1,

Value:0,Va:0,Action:0 with parallel of 0

True

SEQ: TYPE:16,

Name:assertEqual,

Value:yellow3/Model Root/Yellow3/IO/ScalingFromHardware/VIS/CUV2/Physical/Lights

/FifthWheel_WorkLight[0|1]/Conversion/Out1,Va:0,Action:0 with parallel of 0

False

TRITA-ICT-EX-2015:236

www.kth.se