An Evaluation of Automatic Test Case Generation strategy from Requirements for Electric/Autonomous Vehicles

Athul Gangadharan

Subject: Information Systems
Corresponds to: 30 hp
Presented: VT 2020
Supervisor: Mudassir I. Mustafa
Examiner: Steve McKeever

Department of Informatics and Media
Abstract

Software testing is becoming more prominent within the automotive industry due to more complex systems, and functions are implemented in the vehicles. The vehicles in the future will have the functionality to manage different levels of automation, which also means that vehicles driven by humans will have more supportive functionality to increase safety and avoid accidents. These functionalities result in a massive growth in the number of test scenarios to indicate that the vehicles are safe, and this makes it impossible to continue performing the tests in the same way as it has been done until today. The new conditions require that the test scenarios and Test Cases both be generated and executed automatically. In this thesis, an investigation and evaluation are performed to analyze the Automatic Test Case Generation methods available for inputs from Natural Language Requirements in an automotive industrial context at NEVS AB. This study aims to evaluate the NAT2TEST strategy by replacing the manual method and obtain a similar or better result. A comparative analysis is performed between the manual and automated approaches for various levels of requirements. The results show that utilizing this strategy in an industrial scenario can improve efficiency if the requirements to be tested are for well-documented lower-level requirements.

Keywords

“software testing”, “nlp”, “test case generation”, “natural language requirements”, “requirements-based testing”, “electric vehicles.”
Acknowledgement

First of all, I would like to show my gratitude to the organization NEVS AB for providing me with a chance to collaboratively work on this thesis.

I would then like to thank my supervisor Mudassir I. Mustafa of Uppsala University for his useful guidelines since the very beginning of the thesis and supporting me throughout the whole period.

I wish to express my deepest gratitude to Veerappa Kaujageri, my supervisor at NEVS for his patient guidance, continuous encouragement and limitless help in each and every step of this thesis. I would also like to extend my gratitude and appreciation to Camilla Lindeblad, from NEVS, for her valuable comments and for providing me with all the necessary support and resources. I would also like to acknowledge the team members from the Software Tribe at NEVS for their help and feedback.

I am grateful to Gustavo Carvalho from Universidade Federal de Pernambuco in Brazil for his continuous support and by patiently providing me with all the useful information for this thesis.

Finally, I want to thank my parents and friends for their endless support and encouragement. It would have been impossible to finish this thesis without their hope in me.
INDEX

1. Introduction 9
   1.1. NEVS 9
   1.2. Research Problem 10
   1.3. Scope of the Thesis 11
   1.4. Delimitation 12
   1.5. Outline of the Thesis 12

2. Background 13
   2.1. Literature study 16

3. Method 22
   3.1. Case Study Design 22
   3.2. Data Collection Methods 23
   3.3. Data Analysis Methods 23

4. NAT2TEST 25
   4.1. SysReq-CNL 26
   4.2. Syntactic Analysis 30
   4.3. Semantic Analysis 31
   4.4. DFRS Generation 32
   4.5. SCR & CSPm 34
   4.6. FDR & Z3 35
   4.7. Test Cases 36

5. NAT2TEST COQ 38
   5.1. DFRS 39
   5.2. Coq Proof Assistant 40
   5.3. QuickChick 41
   5.4. SRCIROR 41

6. Manual Test Case Generation 43

7. Evaluation Analysis & Results 44
   7.1. System Under Test 1: Range Warn Level Estimator 44
       7.1.1. Requirements for SUT 1 44
       7.1.2. Manual approach of generating Test Cases for SUT 1 45
       7.1.3. NAT2TEST’s approach of generating Test Cases for SUT 1 46
       7.1.4. Analysis Results for SUT 1 47
   7.2. System Under Test 2: Hill Hold 48
       7.2.1. Requirements for SUT 2 49
       7.2.2. Manual approach of generating Test Cases for SUT 2 50
7.2.3. NAT2TEST’s approach of generating Test Cases for SUT 2 51
7.2.4. Analysis Results for SUT 2 52
8. **Discussion & Conclusions** 54
  8.1. Answering Research Questions 54
  8.2. Conclusion 55
9. **Limitations and Future Work** 57
References 59
10. **Appendix** 63
    A.1. Remaining Range Estimator NAT2TEST Test Cases: 63
    A.2. Hill Hold Manual Test Cases: 64
    A.2. Hill Hold NAT2TEST Test Cases: 65
**Disclaimer** 67
List of Figures

Figure 1 Overview of the scope of this thesis ................................................................. 11
Figure 2 V shape diagram ............................................................................................... 15
Figure 3 Overview of Litmus[7] ...................................................................................... 17
Figure 4 Using LG parser ............................................................................................... 17
Figure 5 Example of gherkin syntax .............................................................................. 18
Figure 6 Overview of KIRBY[26] .................................................................................. 19
Figure 7 Overview of the UMTG[17] technique ................................................................. 20
Figure 8 NAT2TEST overview ...................................................................................... 26
Figure 9 Structure of SysReq-CNL ................................................................................. 27
Figure 10 Detailed structure of SysReq-CNL ................................................................. 28
Figure 11 Screenshot of the Requirements window ...................................................... 29
Figure 12 Screenshot of the Dictionary window ............................................................. 29
Figure 13 Screenshot of the Syntax tree generation window ......................................... 31
Figure 14 Screenshot of the Requirement Frame generation window ......................... 32
Figure 15 Screenshot of the Variable & Types window ................................................. 32
Figure 16 Screenshot of the Function window ............................................................. 33
Figure 17 Screenshot of the Animation window ............................................................ 33
Figure 18 Screenshot of the CSPm window .................................................................. 35
Figure 19 Screenshot of the Test Case generation window ......................................... 36
Figure 20 Screenshot of the Test Cases in the project folder ....................................... 36
Figure 21 Overview of various NAT2TEST strategies ................................................... 38
Figure 22 Future scope of the working of NAT2TEST ................................................ 39
Figure 23 Test Cases in CSV for TIS(from NAT2TEST) example ............................... 41
Figure 24 Example of assigning the initial values at Variables & Types ...................... 46
Figure 25 Research instrument for answering the research questions ....................... 55
Figure 26 Mapping the signal values .......................................................................... 57
List of tables

Table 1 Exported Test Case ................................................................. 37
Table 2 Initial requirements for Range Warn Level ............................................... 45
Table 3 Range Warn Level Requirements in SysReq-CNL........................................ 45
Table 4 Manual Test Cases for Range Warn Level ............................................... 46
Table 5 Some of the Test Cases for REQ002 for Range Warn Level............................ 47
Table 6 NAT2TEST Test Cases for Range Warn Level ........................................... 47
Table 7 Time taken to develop Test Cases manually for SUT 1.................................. 48
Table 8 Analysis results for SUT 1 ..................................................................... 48
Table 9 Initial requirements for Hill Hold ............................................................... 49
Table 10 Hill Hold requirements in SysReq-CNL ..................................................... 50
Table 11 Time taken to develop Test Cases manually for SUT 2............................... 51
Table 12 Test Case data for both drive and reverse ............................................... 51
Table 13 Test Case data for Drive ........................................................................ 52
Table 14 Remaining Test Cases from Remaining Range Estimator......................... 64
Table 15 Remaining Test Cases from Hill Hold ...................................................... 66
List of Abbreviations

EV – Electric Vehicle
AD – Autonomous Drive
HIL – Hardware In Loop
MIL – Model In Loop
SUT – System Under Test
SIL – Software In Loop
NLP – Natural Language Processing
ECU – Electronic Control Unit
DFRS – Data Flow Reactive System
SysML – Systems Modelling Language
CNL – Controlled Natural Language
JDK & JVM – Java Development Kit & Java Virtual Machine
IMR – Internal Model Representation
SCR – Software Cost Reduction
CSP – Communicating Sequential Processes
CSPm - Communicating Sequential Processes model
POS – Part of Speech
LTS – Labelled Transition System

Glossary

MC/DC - Modified Condition/Decision Coverage is a code coverage criterion utilized in software testing.
1. Introduction

Autonomous driving and Electric vehicles soon are about to become increasingly common on our roads. The passengers of the automated vehicles of the future will sit safely and comfortably in a controlled room[1]. Autonomous vehicles will play an essential role in driving shared mobility. Machado et al., describe shared mobility as the short-term access to shared vehicles according to the user’s needs and convenience[2]. The benefits also include anticipated reductions in carbon dioxide emissions and more efficient transport. There are several methods involved to verify and secure the safety-critical goals of autonomous or electric vehicles. Testing is one of the ways of evaluating that the vehicles are on par with safety and practical parameters. Testing analyzes and provides certainty that all systems involved in the decision-making process are working in complete tandem with one another without any problems[3]. Since most of the automobile industry will be driven by software, the automobile industry is slowly but steadily moving towards becoming a software-centric industry[4].

Software Testing plays an essential role in the software development life cycle. It is performed to ensure that there are no discrepancies in developed products and meet the needs of the business[41]. Designing Test Cases and generation is a tedious manual process that requires around 40-70% of the software test life cycle[5]. The Test Cases written manually by inexperienced testers may not offer complete coverage of the requirements. Frequent changes in requirements reduce the reusability of the manually written Test Cases costing more time and effort[5]. Furthermore, the number of requirements when it comes to Electric Vehicles & Autonomous Driving vehicles will be vast, which only makes manual test case generation effort very huge. One of the vital approaches in Software Testing of vehicles consists of the Model-Based Design approach. It saves time during the designing phase of onboard software by testing each feature when it is developed. The whole process includes drafting the specifications, creating Test Cases, and executing Test Cases in different test environments. The Test Environments include testing on Model in Loop, Software in Loop, Processor in Loop (PIL), and Hardware in Loop[6]. In the past, there have been many methods developed to execute the Test Cases automatically, and the difference in this study is generating Test Cases automatically from requirements. Combining them both can possibly lead to having an approach for complete test automation.

1.1. NEVS

NEVS, or in full National Electric Vehicle Sweden AB¹, is a startup sprung from the assets of SAAB Automobile's bankruptcy (Wikipedia n.d.). NEVS focuses entirely on its vision: Shaping Mobility for a Sustainable Future. NEVS recently released information about their autonomous pod called Sango, part of NEVS’ mobility ecosystem named PONS². Since summer 2020, NEVS was wholly owned by the Chinese real estate giant Evergrande.

¹ https://www.nevs.com/en/
In this thesis, a case study was conducted at NEVS. This study was conducted at an automobile industry to discover and illustrate how Test Cases can be automatically generated from requirements. Furthermore, how this strategy can improve the current traditional method of manually creating Test Cases as well as guiding a path to increase the efficiency and effectiveness of the testing process. As the automatic creation of Test Cases can have a significant impact on cost reduction.

1.2. Research Problem

In this research, the primary focus is on investigating and evaluating a strategy that can generate Test Cases automatically from natural language requirements. Instead of creating Test Cases manually by analyzing the requirements, the tool should reduce the effort required to create Test Cases while improving the test coverage and product quality. In the software development life cycle, the high-level requirements are provided to the domain experts by the stakeholders; these requirements are usually in the form of texts or state diagrams. The domain experts then develop lower-level requirements that are written in natural language, and these requirements sometimes can also be supported by state-charts, diagrams, SysML, and other formats. Then the software test engineers will develop Test Cases manually according to the requirements, which are then used to test the software developed by the developers.

According to the surveys in the past, industries documented 79% of all requirements in free-flow natural language, and only 7% use formal specifications[7]. The software development projects usually are written in natural language as it is easily understandable by a majority of users who need not have domain knowledge. The natural language requirements usually do not follow a formal structure. In the test case generation point of view, if the natural language requirements do not have a defined structure, then they can cause multiple challenges[8][9].

This thesis also aims to explore a suitable formal structured rule to write the requirements as it can help in improving the testing process. Furthermore, an approach to find a solution that is suitable for lower-level requirements is also considered.

Figure 1 shows the traditional approach available and the approach that this thesis aims to provide.
1.3. Scope of the Thesis

The scope of the thesis is to investigate a strategy/tool that can improve the efficiency of the testing process by generating Test Cases that can be used in test execution. Along with test case generation, also include some features such as highlighting requirements for which Test Cases could not be generated, a strategy for reduction of Test Cases (e.g., equivalence classes), and a method for removing duplicates.

Given the overall scope & aim, the primary objectives of the research were to:

- Perform literature study to discover the appropriate approach available to identify Test Cases from the textual requirements at different test levels used in software testing the various distributed automotive embedded software functions
- Collect and analyze information on the current manual approach and try to implement the automated approach along with performing research to obtain a structured rule for requirements writing—moreover, the challenges faced in using the structure.
- Evaluate & implement a suitable strategy for a few sets of requirements and execute the Test Cases in test environments.
• Demonstrate the validity of the proposed strategy by comparing it with the manual approach and evaluating its feasibility in an industrial environment and providing limitations and conclusions.

The primary focus of the thesis revolves around the below research questions:

**RQ01:** What is the suitable strategy/tool that can be implemented in an automotive industrial practice for automatically generating Test Cases from Natural Language requirements?

**RQ02:** How do the Test Cases generated automatically by the tool compare to the traditional manual approach?

### 1.4. Delimitation

The study focusses particularly on exploring textual Requirements-Based Test Case Generation strategies and shortlisting a potential one and evaluating that in-depth. The level of requirements is focused only on lower-levels in the Software Development Life Cycle. The study aims to verify the usability of the shortlisted tool/strategy. The evaluation results are concluded from performing a comparison between the manual process and the automated approach of the tool/strategy. The data for the manual process is collected from a small group of testers in a specific industry.

### 1.5. Outline of the Thesis

The first chapter consists of the introduction, about the organization NEVS, research problem, scope, delimitation & outline of the thesis. The second chapter describes the background and literature study of various explored tools and strategies. The third chapter explains the methodology used in the study, NAT2TEST, its features, and benefits are explained in the fourth chapter. Chapter five talks about NAT2TEST COQ. The sixth chapter explains about the current manual approach. Furthermore, the results of the study and the test materials are explained in chapter seven. The eighth chapter explains the study and design used to answer research questions. The ninth and tenth chapters will have a conclusion, and limitations and future work, respectively.
2. Background

This part explains the theoretical framework of how the traditional approach is carried out from Requirements and what do the Requirements mean in this research. Additionally, some of the different techniques such as NLP approach and others that could support in the test case generation process is also explained.

Software Testing

Software Testing is conducted to analyze and evaluate the quality of System Under Test (SUT) and its risk of failure involved. Modern vehicles consist of multiple embedded systems/ECUs which provide various functions to run the vehicle. The ECUs perform communication with each other using different communication protocols such as Ethernet, LIN, a single area network called CAN, and a few others [4]. Many of the ECUs are safety-critical, and the software must go through rigorous testing processes, as even slight faults can lead to severe injuries. When it comes to functional and non-functional testing. The latter is a type of software testing where the system or component is tested against the functional requirements. It is done to make sure that software meets high-level and low-level requirements. The automotive systems follow the ISO26262 standard, and this standard is specific to safety-critical automotive systems.

The test case creation can follow one of the techniques, such as Requirement-Based Testing, Model-Based Testing, and Code-Based Testing of Test Cases. In Code-Based Testing, the Test Cases are executed to make sure that the various system test paths are covered. While in Requirements-Based Testing, the requirements specify the inputs and outputs of the system, and their functionality is evaluated[5]. Perceived advantages of Requirements-Based Testing technique are that the Test Cases are created from the requirements, and since it is performed at the beginning of development, the errors are discovered at an early stage, which in turn reduces the costs. Furthermore, this technique is performed at the stage where the most substantial portion of bugs has their root cause. Improving the quality of requirements will help in reducing the failure of projects[10]. In this thesis, the focus will only be on testing based on requirements, the test levels where the focus is only on requirements-based test coverage were considered. According to [11], test levels are groups of test activities that are organized and managed together. Each test level is related to other activities in the software development life cycle. The test levels can vary according to the need and general test levels followed are:

- Component testing: This type of testing focuses on separate components which are testable. In some cases, unit testing is also performed, which is a level below component testing.
- Integration testing: Integration testing is a type of testing for the interface as well as the architecture of the software, depending on the level of testing. It is aimed at interactions between components or systems. They are done to verify the behavior of interfaces and to reduce risks in interfaces or components itself.
- System testing: This is a type of testing performed at a higher level where the expected behavior and capability of the whole system is tested. The test objects involved in this level are applications, hardware or software systems, SUT, and others.
• Acceptance testing: Acceptance testing is similar to system testing; it can find the readiness by end-user of the whole product or system. Acceptance tests, in general, can be performed at every level. Here it is considered as System Acceptance testing. It is performed to validate the system is complete as expected.

Model-Based Testing

In the past few years in automotive development, the testing of automotive embedded systems such as ECUs has changed from just electrical & mechanical engineering discipline to a combination of software and electrical/mechanical engineering[12]. The automotive industry has been following new trends towards Model-based testing. MBT is a software testing technique where run time behavior of the system under test is checked against predictions made by a model[45]. The Test Cases are derived directly from the model of the system that is being developed. MBT is a conventional terminology which is used to indicate a testing process in which the most common testing tasks such as the generation of Test Cases and test case execution. The automated Test Cases are generated from the model of the system[12]. There are added benefits to this method, such as it is less prone to errors and less human interaction; the faults are caught in the initial stages, which helps in reducing testing time. MBT can also help in finding inconsistencies in the requirements.

Few limitations by using this method are that the models may not have been developed in the early stages so that it can be tested, and it can be challenging to perform MBT without having the knowledge of model syntaxes and semantics[14]. In a few circumstances, the working model will not be created by the domain architects since they will be focused on creating the requirements. The developers who create the model from requirements are also susceptible to create a model with fault. Thus, testing the model by creating Test Cases from requirements using NLP can be a way forward.

Requirements

Requirements are a commonly used formal phrase in automotive engineering design that are used to document functional and non-functional need that requires a particular design, system, or component should aim to satisfy. Requirements engineering is a technique used to ensure the validity of the requirements. Multiple studies performed by Standish Group3 suggest that inappropriate Requirements Engineering can play a vital role in project failures[15]. The requirements are usually developed by engineers who are responsible for specifying how the system/component should work. Requirements can be functional & non-functional requirements.

3 https://standishgroup.com/
The requirements can be on many levels starting from the high to low-level requirements, and they can vary accordingly. The ‘V’ shape diagram of automotive systems inspired by [16] explaining the design and verification stages of the life cycle is illustrated in Figure 2. The general level of software requirements in an automotive industry that can be considered in this study are:

- **Vehicle-Level Requirements:** These requirements are depicted as one of the higher-level requirements, which defines what the vehicle or product is supposed to perform. The final verification is performed on the vehicle at this level.
- **System-Level Requirements:** In the Domain/System Level Requirements, the integration between several sub-systems is defined. The sub-systems can be an ECU or any other system.
- **Component-Level Requirements:** These requirements come under lower-level software requirements where actions of each component in a system are defined.
- **Unit-Level Requirements:** The unit-level software requirements specify which software units are expected to perform.

The software requirements can be documented in multiple formats; they can be based on the specification language required. The requirements can be written in Natural Language, Unified Modelling Language (UML), flow diagrams, state charts, supporting graphs for clear understanding, and many other formats as required[17]. The formats for documentation can vary according to the needs and time available. In this thesis, the focus will be only on requirements written in Natural Language. The requirements are created, added, viewed, and managed by different stakeholders in a requirement management tool. Some of the examples of requirement management tools used in the automobile industry are IBM DOORS⁴, Siemens Polarion⁵, and SystemWeaver⁶.

---


⁵ [https://polarion.plm.automation.siemens.com/](https://polarion.plm.automation.siemens.com/)

⁶ [https://www.systemweaver.se/](https://www.systemweaver.se/)
NLP

Natural Language Processing (NLP) is broadly defined as the automatic manipulation of natural languages, like speech, text & pseudo code by software. It is a method to extract meaning from free texts. This technique makes use of linguistic concepts such as part-of-speech (which typically is performed by a POS-Tagger tool), grammatical structure, anaphor, and ambiguities. To solve some of the issues faced, NLP makes use of lexicons of words and their meanings, properties, and grammatical rules[18]. NLP has several popular applications such as speech recognition, language translation, spell-checking, finding plagiarism, and chatbots. In this thesis, since the requirements are in Natural Language, NLP is utilized. The Test Cases are derived from NL requirements, and NLP is used in syntactic and semantic analysis of the test case generation process.

2.1. Literature study

There have been numerous studies to investigate the automatic test case generation method from requirements. A number of authors have recognized requirements as inputs rather than models[7][5][17]. Several theories have been proposed, and some are implemented for several formats of requirements, some focusing on use case models, some on natural languages, and others on the model as input. And finally, the literature study of various papers that involves the shortlisted tool/strategy of this thesis is explained in the end of this chapter.

A paper written quite a long time ago by Bringmann et al. [12] explained how model-based testing was increasing popularity in the automotive industry. This paper explained the use of different test environments such as MIL, HIL, SIL, and others. They also explained how the characteristics of each test environment required the use of dedicated model-based testing approaches. The characteristics of requirements for different environments were different. They introduced an approach called Time Partition Testing (TPT), which has been specifically designed to bridge this gap for model-based automotive testing.

The authors in [7] employed a Test Case Generation from Functional Requirements methodology, which prescribes the use of Litmus, a tool developed by the authors, which generates Test Cases from NL requirements. The tool considers the requirement as a sentence as the input to generate Test Cases. An overview of the method of Litmus tool is shown in Figure 3.
Litmus uses the Link Grammar Parser, which was developed at Carnegie Mellon University as the first step of the tool to parse the requirements. An example of using LG Parser is illustrated in Figure 4. The parser will provide links of connected words if the syntax of the sentence is grammatically correct. The links are created by following an English grammar dictionary. In the next stage, the tool will first extract entities then obtain syntactic and lexical information. Secondly, it will check for the testability of the sentence and then breaks the complex sentence to a simple sentence. In the next stage, it will generate test intents of simple sentences. Then it will group and arrange Test Intents into positive Test Cases. Finally, the tool will generate negative Test Cases. A Test Case generated by Litmus consists of Test Condition, Test Sequence, and Expected Result.

Figure 3 Overview of Litmus[7]

Figure 4 Using LG parser.
The tool obtained an overall average accuracy of 77 percent. The tool had few drawbacks, which were mentioned in the limitations, and the tool is mainly dependent on the syntax of the sentence. However, there was no openly available information related to the tool explored for requirements at a lower level with input/output signals and dependent on time. Litmus tool had the potential to be investigated, but according to the authors, the tool was not available for research purposes, and it was also not available for commercial use outside the proprietary organization.

There exists a considerable amount of literature on Test-Driven Development and Behavior-Driven Development using Cucumber’s Gherkin parser. Cucumber is a tool that ensures that an acceptance test can be written and understood by anyone in the team. It is a tool that is focused on business stakeholders to be involved in creating acceptance tests[24]. Cucumber uses Behavior-Driven Development. The formal definition of BDD, according to its originator, was “BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple scales, high-automation, agile methodology.” An example of the high-level requirement of the system expressed as feature is shown in Figure 5. The domain language follows a ubiquitous property.

```plaintext
# Comment
@tag

Feature: Eating too many cucumbers may not be good for you

Eating too much of anything may not be good for you.

Scenario: Eating a few is no problem
    Given Alice is hungry
    When she eats 3 cucumbers
    Then she will be full
```

Figure 5 Example of gherkin syntax

Gherkin is the language used by Cucumber to define Test Cases. It is mostly for non-technical users to define the system specification. Kamalakar et al.,[26], wrote a paper about BDD and Kirby, a tool written in Java for auto-generating glue code from BDD scenarios. Figure 6 gives a comprehensive overview of the structure of Kirby. Kirby is mainly capable of automatically generating code for projects in Java. This paper was considered in this thesis because of its state of art NLP technique used to process the user stories. Few of the NLP techniques used in this generation process are POS Tagging, Treebank, WordNet8, Measuring similarity, Lemmatization & finally, automatic code generation.

---

8https://wordnet.princeton.edu/
Over time, an extensive literature has developed on Use Case Modelling for System Tests Generation (UMTG)[27], it is an approach that generates executable system Test Cases by exploiting behavioral information in use case specifications. The behavioral modeling is avoided by applying NLP to a more structured and analyzable form of Use-Case specification. UMTG tool requires multiple dependency tools such as IBM DOORS and Rhapsody⁹ to investigate, and they are not open source. The Figure 7 gives an overall working of UMTG.

---

⁹ https://www.ibm.com/products/systems-design-rhapsody
NAT2TEST\textsuperscript{10} is a strategy that generates Test Cases from requirements written in Controlled Natural Language (CNL). Seminal contributions have been made by several authors around the NAT2TEST strategy. It is a strategy for the automatic generation of Test Cases from requirements written in natural language. In this strategy, the requirements are first written in a Controlled Natural Language, and then they are parsed through the CNL parser, which consists of Syntactic Analysis, Semantic Analysis, and SCR code generation. The SCR code is given as input to a commercial tool called T-VEC. It then generates Test Cases, which in turn can be implemented in different environments such as HIL, Simulink/MATLAB, Rig testing, and others [22]. SysReq-CNL is the structure followed to generate Test Cases. A study by [31] analyzes the SysReq-CNL and evaluates it in a detailed manner, and this study helps in shortening the research gap.

Another paper [14] published in 2019, proposed DFRScoq, it models and tests timed Data Flow Reactive Systems by generating Test Cases from controlled natural language requirements. According to the definition from Coq, it is a formal proof management system, and it is a single framework to specify and verify DFRS models. Moreover, the specification of DFRSs in Coq is automatically derived from controlled natural language requirements. The Test Cases are then generated using the QuickChick\textsuperscript{11} tool.

\textsuperscript{10} https://www.cin.ufpe.br/~ghpc/
\textsuperscript{11} https://softwarefoundations.cis.upenn.edu/qc-current/index.html
[30] discuss about several Test Case generation approaches similar to NAT2TEST, but these approaches rely heavily on well-documented requirements, and these approaches are most suitable for low-level requirements. The approach proposed in this paper is to develop a test automation technique to combine Capture and Replay tools and coding approaches. The approach also includes involving a bottom-up approach to reuse Test Cases.
3. Method

The purpose of this thesis was to conduct a case study to investigate various strategies and tools that can generate Test Cases from requirements and evaluate a potential strategy that can be implemented in an industrial context. Furthermore, this study will also include proposing a better structure to document the requirements. The study will try to attempt to answer the specified research questions that have a knowledge gap[23]. The study will be carried out by utilizing the mix of variables obtained from the literature study and the data obtained from the stakeholders, and this will be performed by dividing the study into phases. Combining the literature study and conducting a case study is the primary methodology used in this thesis.

This study will mainly consist of two methodological phases. The first phase comprises of the literature study of relevant articles, journals, conference proceedings, theses, presentation slides, posters, and papers. The information was gathered from popular proceedings such as ScienceDirect, IEEE, ACM, Springer, Semantic Scholar, Diva, and others. As this literature study was conducted to explore various existing strategies, tools, and methods available. It primarily focused on investigating techniques such as Requirements-Based Testing, Test-Driven Development, Natural Language Processing techniques & Model-Based Testing. The investigation used the keywords Autonomous Driving, Test-Case Generation, Model-Based Testing, Natural Language Parsing, GHERKIN, Test-Driven Development, Agile-Based Testing, were amongst many others. The literature study was a vital stage in this study, as it helped in identifying the essential strategy/tool that is primarily evaluated in this thesis. From the literature study findings as explained in 2.1, the strategy NAT2TEST was shortlisted because of the least amount of human intervention needed to generate executable Test Cases, and this strategy used the approach of documenting requirements in SysReq-CNL, a CNL which was suitable for the automotive domain. The shortlisting of the instrument NAT2TEST was based on the previous instruments used in published research papers.

The second phase consists of conducting an evaluation case study at NEVS. The methodology followed in this phase is derived from [13] as it performs a similar case study on analyzing the application of MBT, whereas this study focusses on Requirements-Based Test Case generation. As stated by [32]“A case study is an in-depth exploration from multiple perspectives of the complexity and uniqueness of a particular project, policy, institution, program or system in a ‘real-life’”. The case study was performed to compare the analysis results between the automated test case generation approach and the manual test case generation approach at NEVS. Moreover, the obtained results from analysis paved the way for answering the research questions.

3.1. Case Study Design

To achieve the objectives in relation to the research questions, a case study was conducted at the organization. The need for such a case study was to gain an in-depth understanding of the requirements-based test case generation process in an industrial scenario. Since the study was performed at the organization, it was vital to discuss and carefully follow the steps involved in the manual test case generation process. The data from the manual process will eventually help in
performing a comparison of the results with the automated approach. The design for this case study was according to [40], and as followed from [13], the case that is specified here are the SUTs of the vehicle dynamics system. The theory studied in this case study is the requirements-based test case generation method. The generic steps involved in this method is analyzing the textual requirements, designing the Test Cases, reviewing the Test Cases, and develop executable Test Cases for various Test Environments. Furthermore, to analyze the acceptance of the NAT2TEST strategy based on its efficiency and effectiveness, the research questions are aimed to be fulfilled.

3.2. Data Collection Methods

For this study, there are multiple methods of collecting data that have been implemented. According to [43], data collection methods such as interviewing, observation, and document analysis are considered as qualitative methods of data collection. One of the methods of collecting qualitative data was by closely observing and analyzing each step involved in the manual approach at the organization as the proposed automated approach had to obtain similar or better outputs as of the manual approach. Further examples of data collection methods involved collecting data through utilized technologies such as emails and many others [44]. For this study, more detailed data was collected from the authors whose tools were found beneficial from the literature study; this data was useful while performing the evaluation. With regard to quantitative data, the collection and analysis were conducted at the organization. Initially, the two sets of requirements data were collected from the organization to perform the evaluation and analysis for both manual & automated approaches. Furthermore, the data from the manual approach’s analysis was collected from test engineers. The collected data from engineers included manually created Test Cases from the requirements and the time required to design the Test Cases and total number of Test Cases from each engineers. More information about the collected data and the selected engineers are mentioned in the chapter 7. Further details of the several test engineers were anonymized for privacy concerns.

3.3. Data Analysis Methods

The Data Analysis method will be an approach of both qualitative and quantitative analysis methods. Qualitative analysis was performed from the literature study by analyzing the existing strategies and tools to obtain the suitable approach that was shortlisted for the evaluation.

The two sets of requirements data mentioned in chapter 7 will be used to generate Test Cases from both the NAT2TEST approach and manual process. The same input requirements were provided to the engineers with different levels of experience, this was necessary because the experience of a tester can have an impact on the efficiency of the attributes considered for analysis. Whereas for the second set of requirements (SUT 2), only a single level of experience was necessary because the difference in the collected data was not significant and also because the test execution time was considered. For the automated approach, initially, the same requirements given to the manual testers had to be transformed into the format of SysReq-CNL to be able to generate Test Cases through NAT2TEST. The Test Cases generated from both the approaches were analyzed by
comparing the valid number of Test Cases obtained and the time required to generate them. Time is considered vital in this study because obtaining an efficient method was vital. The Test Cases are considered valid if they can be used for test implementation and execution processes in various Test Environments such as MIL and HIL. The Test Implementation and Execution processes were only considered to estimate the time required. Thus, detailed results and the process involved in it were considered outside the scope of this thesis.
4. NAT2TEST

This chapter describes an overall explanation and working of the shortlisted tool NAT2TEST CSP, which is primarily used in this study’s evaluation.

Natural Language Requirements to Test Cases - NAT2TEST, is a strategy that generates Test Cases from natural language requirements, which might consider discrete or continuous temporal properties. Automation is vital for this process because the practitioners cannot always have formal modeling knowledge[33].

NAT2TEST was designed mainly for reactive systems whose behavior is described as triggers and actions that should be taken when conditions are met. The NAT2TEST strategy operates in four main phases. Briefly, the third phase consists of Internal Codification, which can be SCR [4.5] specification, CSP [4.5] specification, or IMR specification. The Test Cases, which are generated in the fourth phase, can be performed by the commercial tool T-VEC\(^\text{12}\) for SCR specification. And RT-TESTER\(^\text{13}\) is used to generate Test Cases for IMR specification. The remaining phases of NAT2TEST are explained in detail in this chapter. The overview of the NAT2TEST strategy is illustrated in the figure, as shown in Figure 8. The method used in this study to evaluate is by using CSP specification, it uses formal conformance relation using tools like FDR\(^\text{14}\) and Z3\(^\text{15}\), in this case, the test generation is proven sound[33].

The tool NAT2TEST was developed in Java, and its Graphical User Interface was built using the Eclipse Rich Client Platform (RCP) framework.

For this thesis, the NAT2TEST tool was investigated with the dependent tools and packages which are mentioned below:

- Ubuntu 18.0.4 LTS OS
- JDK & JVM 8
- Python 2.7
- C++
- FDR2
- Z3
- Eclipse Luna

---


\(^\text{13}\) [https://www.verified.de/products/rt-tester/](https://www.verified.de/products/rt-tester/)

\(^\text{14}\) [https://cocotec.io/fdr/index.html](https://cocotec.io/fdr/index.html)

\(^\text{15}\) [https://github.com/Z3Prover/z3](https://github.com/Z3Prover/z3)
4.1. SysReq-CNL

SysReq-CNL is a Controlled Natural Language for System requirements. Controlled Natural Language is basically a subset of specific natural language whose grammars, structures, and dictionaries are restricted to reduce textual ambiguity and complexity. CNLs are used majorly to increase understandability or have equal understandings for all team members and stakeholders. Another widespread use of CNLs in industries is to use NLP to perform analysis of the data[34] provides a comprehensive, detailed discussion of different types of CNLs, its uses, and future scope. Some organizations adopt simple CNLs to provide standardization to their technical documentation for requirement specification. These CNLs consist of guideline rules, such as: “write short sentences,” and “avoid using passive voice” [33][34].

According to [22], SysReq-CNL was specially tailored for editing unambiguous requirements of data-flow reactive systems. It is defined by a phrase structure context-free grammar theory, and a lexicon containing the application domain vocabulary. The grammar used here was defined by a Context-Free Grammar in Extended Backus-Naur Form notation. A sentence is generally formed by also combining the Noun Phrase to the Verb Phrase. Furthermore, a CFG is a structure that was first discovered by Backus & Naur in the late 1950s[42].

This approach is mainly helpful in generating Test Cases that can be helpful in test execution. SysReq-CNL also has the capability to take input data directly from the requirements specifications. The input data are then translated into formal specifications[30].

The simple high-level structure of writing requirements in SysReq-CNL is as shown in Figure 9. Firstly, the input conditions are specified, then the agent or a system that is responsible for performing the action is mentioned. Finally, the output actions which should be verified are mentioned.
The input scenarios of the requirement are specified as the Conditions. The Conditions begin with a conjunction which in this thesis is generally “When”, and then its structure is similar to “conjunction of disjunctions.” The conjunctions are delimited by a COMMA (,) and the AND keyword, and the disjunctions are delimited by the OR keyword[22]. The Conditions are comprised of the noun phrase and the verb phrase; the noun phrase can include multiple nouns since the domain-specific signal names have unique names, which can also include underscores (_). The nouns are eventually then preceded by a determiner and the adjectives. The verb phrase Conditions begin with a verb Condition, which can be the “to be a verb or in the present or past tense”. A verb Condition is followed by an optional NOT condition, which is mentioned explicitly as ( ! ) in equations, then an optional comparative term and a verb complement[22].

The output scenarios are specified as the Actions. It consists of a noun phrase and then the verb phrase, the Actions begin with a SHALL followed by at least one verb Action and one verb complement. SHALL is followed by a colon if there are more than one verb action and verb complement. These elements also need to be delimited by commas. The verb complement consists of a noun phrase, an adjective, an adverb, or a number and then followed by 0 or more prepositional phrases, which persists of a preposition and a variable state[22]. It is also essential to know that NOT condition cannot be considered in output Actions.

A detailed structure of the SysReq-CNL used in this thesis is diagrammatically represented in Figure 10. The First half consists of the input conditions; the "system" is the entity or the agent, which is responsible for performing the actions, and the output actions are represented in the second half.
Figure 10 Detailed structure of SysReq-CNL
Initially, a project is created inside the NAT2TEST tool, and then the requirements can be added along with the requirement ID for traceability in later stages. While the requirements are being typed, the tool has an in-built word predictor, which checks if the requirement follows the SysReq-CNL structure. If the structure is invalid or an undefined word in the Dictionary’s lexicon is detected, then the errors are displayed in the console, as shown below.

Processing the requirement REQ003...
ERROR: after “when remaining...” one of the following Part-Of-Speech was expected [COMMA, OR, $, NPLUR, VPRE3RD, PREP, VTOBE_PRE3, VTOBE_PRE, NSING, VTOBE_PAST3, SHALL], but we found [VBASE].
done.
FINISHED the requirements processing in: 1ms.

The requirements window inside the tool is, as shown in Figure 11.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Part of Speech</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ003</td>
<td>When remaining range becomes greater than 20, the remaining range warn level shall assign 0 to range warn level.</td>
<td>Comma</td>
</tr>
<tr>
<td>REQ004</td>
<td>When remaining range becomes lesser than or equal to 20, and remaining range becomes greater than 15, the remaining range warn level shall assign 1 to range warn level.</td>
<td>Or</td>
</tr>
<tr>
<td>REQ005</td>
<td>When remaining range is lesser than or equal to 15 or remaining range becomes greater than 100, the remaining range warn level shall assign 2 to range warn level.</td>
<td>NPLUR</td>
</tr>
<tr>
<td>REQ006</td>
<td>When remaining range becomes lesser than or equal to 0 or remaining range becomes greater than 100, the soc shall assign 2 to range warn level.</td>
<td>VPREE3RD</td>
</tr>
<tr>
<td>REQ007</td>
<td>When range warn level is 1, the soc shall assign amber to range warn level indication.</td>
<td>PREP</td>
</tr>
<tr>
<td>REQ008</td>
<td>When range warn level is 2, the soc shall assign red to range warn level indication.</td>
<td>VTOBE_PRE3</td>
</tr>
<tr>
<td>REQ009</td>
<td>When range warn level is 3, the soc shall assign yellow to range warn level indication.</td>
<td>VTOBE_PRE</td>
</tr>
</tbody>
</table>

Figure 11 Screenshot of the Requirements window

The Dictionary is created prior to writing requirements. The domain-specific lexeme is classified according to its Part of Speech. The dictionary, as shown in Figure 12, is saved as a ”.dic” file inside the project folder, which can also be reused for other projects as well.

Figure 12 Screenshot of the Dictionary window
An example is considered to illustrate the difference and conversion to SysReq-CNL from initial requirements is explained below.

Initial requirements: If A1 == 0 AND A2 == 0 AND A3 == 0 AND A4 == 0 for more than 3 seconds, the system shall set X=False.

Requirements re-written in SysReq-CNL:

REQ001A: When A1 is 0, and A2 is 0, and A3 is 0, and A4 is 0, and A1 changes to 0 or A2 changes to 0 or A3 changes to 0 or A4 changes to 0, the system shall reset the timer.

REQ001B: When A1 is 0, and A2 is 0, and A3 is 0, and A4 is 0, and A1 was 0, and A2 was 0, and A3 was 0, and A4 was 0, and the timer is more significant than 3.0, the system shall: assign false to X, reset the timer.

According to further communication from one of the authors who developed the tool explained that the reason behind this approach of writing is that the Requirement REQ001A captures the precise moment when all variables become 0, which is when the timer is reset. Requirement REQ001B describes the moment when false is assigned to X. The "was" is necessary to prevent "firing" REQ001B at the same time of REQ001A. In addition, the "reset timer" in REQ001B can be removed if there is no need to assign false to X again 3.0 later (with no changes of the input signals).

By using a CNL in this approach, there can be several limitations, such as being able to use only a few verbs, and it can reduce the usability and can restrict the domain architect's thinking to a specific boundary drawn by the SysReq-CNL. Nevertheless, the brighter side of using this CNL can help in standardizing the format to document requirements so that everyone involved in the project will have the same understanding.

4.2. Syntactic Analysis

In this stage, the CNL-Parser will analyze if the requirements follow SysReq-CNL grammar; once that is passed, each word in the sentence is classified to its corresponding lexical class by the in-built POS-tagger[22]. The POS-Tagger is a software function that can read the text in a specified language and assign the part of speech to each word and tokens[36]. In this tool, the tagger will search for all possible classifications of lexemes. The Generalized LR Parsing (GLR) algorithm was implemented in this tool to handle ambiguity. The syntax tree for REQ003, as shown in Figure 13, is then finally generated. If an ambiguous scenario is raised, then multiple trees are generated, and then a warning is shown stating that each requirement should have only one syntax tree.
4.3. Semantic Analysis

In this stage of the initial analysis, the requirement's syntax tree is taken as input, and each word is assigned to its thematic roles. The roles are then accumulated into Case Frames, as represented in Figure 14 for REQ003.

According to the definition by authors[22] the thematic roles are:

“Action (ACT): the action performed if the conditions are satisfied
Agent (AGT): entity who performs the action
Patient (PAT): entity who is affected by the action
To Value (TOV): the Patient value after action completion.
Condition Action (CAC): the action that concerns each condition
Condition Patient (CPT): the entity related to each condition
Condition From Value (CFV): the CPT previous value
Condition To Value (CTV): the value satisfying the condition
Condition Modifier (CMD): a modifier related to the condition.”
4.4. DFRS Generation

The data from the semantic analysis is represented formally with a symbolically refined format. The DFRS model, as represented in Figure 15, specifies the input and output signal names along with its values acquired from its respective requirements. If the requirements consist of Timer related signals, then it is also shown in this stage. The Global clock is a variable which considers continuous time. This variable helps generate Test Cases when non-timer related requirements are used.

Figure 15 Screenshot of the Variable & Types window

Moreover, Figure 16 shows the functions of the requirements with the statements and traceability back to requirements.
The animation window in Figure 17 initially shows the initial state of the system. Then, it can be guided according to the needs for simulation. Briefly, a DFRS model has two types of transitions: delay and function transition. The former transition is available when the system state is stable (it is not a situation where the conditions described by a requirement hold and, thus, some system reaction is expected), whereas the latter transition is available when a system reaction is expected to happen. To use the animator, the user should click on "Delay" (Enabled transitions). Then, in the pop-up, the user should provide values for the input signals, besides the time elapsed (> 0). From that, a new state will be reached. Anytime, the user can click on a particular state to explore the possible transitions from it.
4.5. SCR & CSPm

The tool generates SCR & CSP codes from the DFRS model’s initial state. However, this study is focusing only on CSP based internal specifications.

An example of the SCR function is shown below. SCR implements monitored and controlled variables to define the system’s requirements.

```
-- Variable "m__range_warn_level" expected values = [1, 2, 0]
-- Variable "m__remaining_range" expected values = [20, 15, 1000, 100, 0]
-- Variable "c__range_warn_level" expected values = [1, 2, 0]
-- Variable "c__range_warn_level_indication" expected values = [amber, red, none]

spec the_remaining_range_warn_level__the_soc__the_ipc

type definitions
  constants definitions
  monitored variables
    m__range_warn_level : INTEGER;
    m__remaining_range : INTEGER;
  controlled variables
    c__range_warn_level : INTEGER;
    c__range_warn_level_indication : INTEGER;

term variables
mode classes
assumptions
assertions
function definitions

var c__range_warn_level :=
  ev
  [1] (@T(m__remaining_range > 20)) -> 0
  [1] (@T(m__remaining_range = 20)) AND (@T(m__remaining_range > 15)) -> 1
  [1] (@T(m__remaining_range > 1000) OR m__remaining_range' = 15) -> 2
  [1] (@T(m__remaining_range > 100) OR @T(m__remaining_range = 0)) -> 2
  ve

var c__range_warn_level_indication :=
  if
  [1] (m__range_warn_level = 1) -> 0
  [1] (m__range_warn_level = 2) -> 1
  [1] (m__range_warn_level = 0) -> 2
  fi
```

The CSPm code is used as an input to generate Test Cases using FDR & Z3. In this generation stage, the DFRS models are formed into CSPs. The CSP model is responsible for providing the completeness, consistency, and reachability of the system requirements. The CSPm code is as shown in Figure 18.
FDR is a tool developed to perform refinement checks for the process algebra CSP. The tool allows the processes defined in the machine-readable format of CSP & CSPm generated by NAT2TEST. The version of the NAT2TEST tool used in this study uses an older version of FDR as the syntaxes used in the newer version differ from the ones compatible with NAT2TEST.

Z3 is another tool used in this version of NAT2TEST. It is a high-performance theorem prover developed at Microsoft research.

The use of these two above tools is used in the test case generation stage. FDR is used in the enumeration of symbolic Test Cases, and Z3 is used in the instantiation of time-related scenarios, and the accumulation of Test Cases is performed with the help of a TCL script[33].
4.7. Test Cases

In the final stage of this strategy, the Test Cases are generated from CSPm. As shown in Figure 19, the requirements to which Test Cases are to be generated can be chosen accordingly.

Since time is an important variable, and it is considered at all times, and as it should evolve, an infinite number of possible Test Cases can be generated. Therefore, to limit the number of Test Cases to be generated, the user can manually enter the number. The refinement checks will be performed until the entered number of different counterexamples is achieved.

The Test Cases generated were aimed to be in executable formats. The Test Cases were in the form of test steps, which can be fed as test implementation steps to various test environments. The Test Cases exported from NAT2TEST were in .csv format in the project folder. The folder with all Test Cases is shown in Figure 20.

The exported Test Cases of REQ003 are as shown in Table 1, they have .csv as the extension for the files, but the data inside have semi-colon separated values instead of a comma.
<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.25</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>100</td>
<td>none</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1 Exported Test Case
5. NAT2TEST COQ

This chapter presents a review of recent literature on implementing the Coq proof assistant to improve the previous NAT2TEST strategy. In Figure 21, retrieved from [14] shows the introduction of the Coq strategy to generate Test Cases. This chapter of this thesis performs a study on implementing the available open-source NAT2TEST COQ strategy which is not yet integrated with the NAT2TEST(CSP) tool. In this section, the Hill Hold requirements’ DFRS models are implemented to generate Test Cases.

![Figure 21 Overview of various NAT2TEST strategies](image)

The initial steps, including Syntactic Analysis, Semantic Analysis, and DFRS generation involved in the previous NAT2TEST tool is followed, and then the steps involved from Coq specification are performed in the NAT2TEST COQ strategy.

---

17 [https://github.com/igormeira/DFRScocq](https://github.com/igormeira/DFRScocq)
The steps mentioned in Figure 22 is the pipeline of the proposed method in the paper[14] to create the next version of NAT2TEST.

The open-source strategy provides a Python script that helps perform all the tasks involved in the last three steps of Figure 22. The input values for the ‘name’ of the model, the number of sample calls, and replications for each call to generate for each example need to be specified before running the script. The script will perform the following tasks in the order mentioned below:

- Generate Test Cases from the DFRS model.
- Process the QuickChick output and save test data into CSV & Output folder.
- Generate the mutants from the created C reference implementation to perform mutation-based testing.
- Run all the generated Test Cases against all mutants.
- Collect and summarize empirical data with the variables Example, No. Of Samples, Replication number, Valid traces, Mutation Time & Score.

5.1. DFRS

DFRS, as explained earlier, are models that are suitable for modeling embedded systems whose inputs and outputs are always available as signals[35]. These models are obtained after Syntactic and Semantic analysis of SysReq-CNL requirements. The symbolic DFRS and expanded DFRS are the two models of DFRS, where s_dfrs consists of the system behavior and initial state, and the e_dfrs represents the initial state and other possible states[35]. These DFRS models are then verified using the proposed Coq framework for test case generation.

The DFRS model generated by the tool for the requirement set of Hill_Hold is shown below. The Hill hold requirements are illustrated in chapter 7.2. The below model is stored as a [project_name]_tests.v file. The .v file consists of the characterization of the DFRS model in Coq, and the outline of the same consists of the variables, initial state, functions, and s_dfrs. The complete content of the .v file is not shown in this thesis.

```coq
(*-------------- VARIABLES ---------------*)

Definition vehicle_speed : VNAME.
```
Proof.
apply \( \text{mkVNAME "vehicle_speed"} \).
apply theo\_rules\_vname. reflexivity.
Defined.

In this section, the input and output signals are analyzed from the requirements and specified here. The name of the signal is specified as VNAME.

(*--------------- INITIAL STATE ---------------*)

Definition Hill\_Hold\_state : STATE.
Proof.
apply \( \text{mkSTATE [}
  \{\"vehicle\_speed\", (i 0, i 0)}
  ; \{\"vehicle\_movement\_state\", (i 0, i 0)}
  ; \{\"brake\_control\_request\_rear\_left\", (b false, b false)}
  ; \{\"brake\_control\_request\_front\_left\", (b false, b false)}
  ; \{\"brake\_control\_request\_front\_right\", (b false, b false)}
  ; \{\"brake\_control\_request\_rear\_right\", (b false, b false)}
  ; \{\"gc\", (n 0, n 0)}
].
apply theo\_rules\_state. reflexivity.
Defined.

While generating Test Cases, it is vital to assign the initial states for generating Test Cases. In this section, the input and output variables are analyzed from the requirements and specified here along with their respective signal values.

(*--------------- FUNCTIONS ---------------*)

Definition REQ001\_sg\_disj1 : DISJ.
Proof.
apply \( \text{mkDISJ [}
  \{\text{mkBEXP (previous \{vehicle\_speed\}) lt (i 1)}
].
apply theo\_rules\_disj. reflexivity.
Defined.

Here the function definition of each requirements is defined

(*--------------- Hill\_Hold: S\_DFRS ---------------*)

Definition Hill\_Hold\_s\_dfrs : s\_DFRS.
Proof.
apply \( \text{mkS\_DFRS Hill\_Hold\_variables Hill\_Hold\_s0 Hill\_Hold\_functions} \).
apply theo\_rules\_s\_dfrs. reflexivity.
Defined.

Here the symbolic DFRS is defined, and if applicable, the expanded DFRS will also be represented.

5.2. Coq Proof Assistant

The formal definition provided to explain Coq is “It is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs”. Coq is mainly implemented in OCaml and C. The language provided in Coq is Gallina (similar to Haskell language), which is a dependently typed functional programming language. In this case,
Coq is suitable to test, validate, and verify the provided systems. The CoqIDE\textsuperscript{18} is a graphical tool used to navigate through a Coq .v file. The scope of NAT2TEST COQ is to perform Property-Based Testing. It is performed to check if a function or a software abides by its specified property. The properties usually have very few details about the output. It checks if the useful characteristics are satisfied[37]. This type of testing became popular after the introduction of QuickCheck in Haskell. QuickChick is inspired by QuickCheck for Coq [35].

### 5.3. QuickChick

QuickChick is a randomized property-based testing tool for Coq. As described in the chapter 5.2, Quickchick is cloned from QuickCheck\textsuperscript{19}. The Quickchick tool is used in generating Test Cases for DFRS models. The .v files of the DFRS models are processed to generate Test Cases. The Test Cases are then generated and exported in Outputs and CSV folders. A chunk of the sample output for REEQ001 in Hill Hold is:

```plaintext
[del (discrete 1, [(vehicle_speed, i 0); (vehicle_motion_state, i 1)])); del (discrete 1, [(vehicle_speed, i 1); (vehicle_motion_state, i 0)])); del (discrete 1, [(vehicle_speed, i 1); (vehicle_motion_state, i 0)])); del (discrete 1, [(vehicle_speed, i 0); (vehicle_motion_state, i 1)])); func ((brake_torque_request_front_right, b true); (brake_torque_request_front_left, b true); (brake_torque_request_rear_left, b true); (brake_torque_request_rear_right, b true)), REQ008];
```

The above sample is also exported as a CSV file for better usability of Test Cases.

An example of the Test Cases in CSV format is shown in Figure 23. This example is for Turn Indication System, a requirement sample provided along with the NAT2TEST tool.

![Figure 23 Test Cases in CSV for TIS(from NAT2TEST) example.](image)

### 5.4. SRCIROR

SRCIROR is a mutator which performs mutations for C/C++ based code. Mutation-based testing is a type of software testing where we change certain small statements in the software and check

\textsuperscript{18} https://coq.inria.fr/refman/practical-tools/coqide.html

\textsuperscript{19} https://hackage.haskell.org/package/QuickCheck
if the Test Cases can find the errors[38]. The change applied to the original software is very tiny, and this method is mainly suitable for the Unit test as the change of mutants do not much affect the overall objective of the software. This type of analysis is performed here to analyze the quality of Test Cases generated by NAT2TEST COQ strategy. A ‘C’ language reference implementation is used to mutate by using SRCIROR. The mutants are then obtained according to the number specified in the Python script by the user. The final score is then calculated by the percentage of killed mutants (changes to software) with the total number of mutants. According to the study, 75.8% of defects were detected by this process[35]. In this study, while performing analysis, the score was only generated for the requirements sample provided with NAT2TEST, which were for Vending Machine (VM) for representation, as shown in the below table. The mutation score was as generated for five replications of one sample.

<table>
<thead>
<tr>
<th>Example</th>
<th>Number of Samples</th>
<th>Replication Number</th>
<th>Time Samples</th>
<th>Valid Traces (%)</th>
<th>Mutation Time</th>
<th>Mutation Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm</td>
<td>1</td>
<td>1</td>
<td>4.2</td>
<td>100</td>
<td>28</td>
<td>74.82517482</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>vm</td>
<td>1</td>
<td>2</td>
<td>2.06</td>
<td>81.82</td>
<td>28</td>
<td>76.57342657</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>vm</td>
<td>1</td>
<td>3</td>
<td>1.89</td>
<td>81.82</td>
<td>30</td>
<td>74.47552447</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>vm</td>
<td>1</td>
<td>4</td>
<td>1.93</td>
<td>100</td>
<td>28</td>
<td>75.17482517</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>vm</td>
<td>1</td>
<td>5</td>
<td>1.94</td>
<td>81.82</td>
<td>28</td>
<td>76.57342657</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
</tbody>
</table>

Table 16 Mutation score analysis for VM
6. Manual Test Case Generation

The manual process involved in the industry is observed closely so that the automatic process can produce similar or even better outputs.

The requirements are maintained by various stakeholders and used multiple users at different levels. Furthermore, the requirements are organized according to the system and documented in various places for each function. To manage the requirements formally and efficiently, they are gathered in a requirements management tool. Moreover, all the users involved will have access to it. Whenever any changes must be made to the requirements, it will be updated, and everyone who has access to it will be able to see the changes along with the name of the user who edited it. With the help of some requirements management tool, the traceability of all requirements is also possible.

There are several strategies followed by the testers to perform the test case generation process such as MC/DC, ISTQB & ISO26262.

The conventional method used for manual test case generation is as shown below:

1. Receiving requirements and their supporting documentation from the domain architects.
2. Analyzing and understanding the requirement.
3. Identify all the relevant signals and their various states(values) from the requirement.
4. Write a positive scenario that satisfies the requirement.
5. List the signals and their expected values when the test case is executed.
6. Identify one or many pre-conditions to execute this test case.
7. Identify post-condition for the test case (Example: Vehicle should be turned off)
8. Write negative scenarios (since we already know the signals from step 3, vary the signals and write various scenarios) to verify the requirement; this is performed to check fault handling.

After the Test Cases are designed, they are peer-reviewed, and each test case is implemented in the required Test Environment and executed accordingly. The Test reports are then generated with Test coverage.
7. Evaluation Analysis & Results

In this chapter, the evaluation is performed for the NAT2TEST CSP tool. The latest NAT2TEST COQ was not considered to be evaluated because the final integrated tool was not yet released as it was still under development. For the evaluation, the two systems with different functions are being tested with both manual and automated test case generation approach for the same input. The requirements which were mostly generic were chosen to be analyzed in this study. While performing the analysis in both the approaches, the data is recorded for the amount of time taken and number of Test Cases generated. Time was considered very vital in the analysis of this study because the time needed to develop large number of Test Cases can be substantially longer. The initial requirements in both the below SUTs are altered for confidentiality, so the actual working of the system used in vehicles may differ.

7.1. System Under Test 1: Range Warn Level Estimator

The Range Warn Level Estimator is a feature in electric vehicles to calculate how far the vehicle can travel before the battery range runs out. This is a lower-level requirement set in which most of the input and output signals have values. These requirements were considered to be in component level.

7.1.1. Requirements for SUT 1

In Table 2, the Requirements used in the manual process are specified, which include requirements, supporting information, and pre-conditions. Considering all of the information provided in the table, the Test Cases are designed.

<table>
<thead>
<tr>
<th>Requirement Type</th>
<th>Initial Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
<td>Rng_Wrn_Lvl: Remaining Range Warn Level</td>
</tr>
<tr>
<td></td>
<td>Range: 0 – 2</td>
</tr>
<tr>
<td>Information</td>
<td>Battery_SOC: State of Charge of the High Voltage Battery</td>
</tr>
<tr>
<td></td>
<td>Range: 0 – 100</td>
</tr>
<tr>
<td></td>
<td>invalid value: &gt;100</td>
</tr>
<tr>
<td>Information</td>
<td>Rem_Rng: Remaining Range</td>
</tr>
<tr>
<td></td>
<td>Range: 0 – 1000</td>
</tr>
<tr>
<td></td>
<td>invalid value: &gt;1000</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>Check if range warn level is active when vehicle state becomes active</td>
</tr>
<tr>
<td>Req1</td>
<td>If Rem_Rng &gt; 20km, then Rng_Wrn_Lvl is set as, Rng_Wrn_Lvl = 0</td>
</tr>
<tr>
<td>Req2</td>
<td>If Rem_Rng &lt;= 20km, then Rng_Wrn_Lvl is set as, Rng_Wrn_Lvl = 1</td>
</tr>
<tr>
<td>Req3</td>
<td>If Rem_Rng &lt;= 15km, then Rng_Wrn_Lvl is set as, Rng_Wrn_Lvl = 2</td>
</tr>
<tr>
<td>Req4</td>
<td>If Rem_Rng is an invalid value, then Rng_Wrn_Lvl is set as, Rng_Wrn_Lvl = 2</td>
</tr>
<tr>
<td>Req5</td>
<td>If SOC is received as an invalid value, then Rng_Wrn_Lvl is set as, Rng_Wrn_Lvl = 2</td>
</tr>
<tr>
<td>Req6</td>
<td>If IPC receives Rng_Wrn_Lvl = 1, then Rng_Wrn_Ind is set as, Rng_Wrn_Ind = Amber</td>
</tr>
<tr>
<td>Req7</td>
<td>If IPC receives Rng_Wrn_Lvl = 2, then Rng_Wrn_Ind is set as, Rng_Wrn_Ind = Red</td>
</tr>
</tbody>
</table>
If IPC receives Rng_Wrn_Lvl = 0, then Rng_Wrn_Ind is set as, Rng_Wrn_Ind = None

Table 2 Initial requirements for Range Warn Level
In Table 3, the requirements are then written in the proposed structure, since there is no possibility to assign pre-conditions explicitly, information and post-conditions in NAT2TEST, the pre-condition, and the information are embedded separately as a requirement and included along with other requirements. The signal names IPC and SOC can be considered as nouns.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>The requirements were re-written in SysReq-CNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ001</td>
<td>When vehicle state becomes true, and remaining range becomes lesser than or equal to 15 or remaining range becomes greater than 1000, the remaining range warn level shall assign 2 to range warn level.</td>
</tr>
<tr>
<td>REQ002</td>
<td>When vehicle state becomes true, and remaining range becomes greater than 15, and remaining range becomes lesser than or equal to 20, the remaining range warn level shall assign 1 to range warn level.</td>
</tr>
<tr>
<td>REQ003</td>
<td>When vehicle state becomes true, and remaining range becomes greater than 20, the remaining range warn level shall assign 0 to range warn level.</td>
</tr>
<tr>
<td>REQ004</td>
<td>When vehicle state becomes true, and soc range is lesser than 0 or soc range is greater than 100, the remaining range warn level shall assign 2 to range warn level.</td>
</tr>
<tr>
<td>REQ005</td>
<td>When range warn level is 0, the ipc shall assign none to range warn indicator.</td>
</tr>
<tr>
<td>REQ006</td>
<td>When range warn level is 1, the ipc shall assign amber to range warn indicator.</td>
</tr>
<tr>
<td>REQ007</td>
<td>When range warn level is 2, the ipc shall assign red to range warn level indication.</td>
</tr>
</tbody>
</table>

Table 3 Range Warn Level Requirements in SysReq-CNL

7.1.2. Manual approach of generating Test Cases for SUT 1

In this set of requirements, with the range and values clearly specified for each input and output signals. A fraction of the total required Test Cases is mentioned in Table 4. This data for the manual process is collectively obtained from test engineers who analyzed the requirements and then designed the Test Cases; the Test Cases are then reviewed for its correctness. To perform the analysis, only the best Test Cases were shortlisted from the number of Test Cases obtained from multiple testers.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Case</th>
<th>Test Case Name</th>
<th>Steps/Description</th>
</tr>
</thead>
</table>
| TC_01   |      | Check if Range Estimation is active/enabled when vehicle becomes active         | 1. Start the vehicle  
       |      |                                                                                  | 2. Verify 0 <Rem Rng<1000                            |
| TC_02   |      | Check Remaining Range Warn Level when Remaining Range is 21                    | 1. Start the vehicle  
       |      |                                                                                  | 2. Set the Rem Rng to 21km                           |
|         |      |                                                                                  | 3. Verify Rng Wrn Lvl = 0                            |
| TC_03   |      | Check Remaining Range Warn Level when Remaining Range is 20                    | 1. Start the vehicle  
       |      |                                                                                  | 2. Set the Rem Rng to 20km                           |
|         |      |                                                                                  | 3. Verify Rng Wrn Lvl = 1                            |
| TC_04   |      | Check Remaining Range Warn Level when Remaining Range is 15                    | 1. Start the vehicle  
       |      |                                                                                  | 2. Set the Rem Rng to 15km                           |
|         |      |                                                                                  | 3. Verify Rng Wrn Lvl = 2                            |
| TC_05   |      | Check Remaining Range Warn Level when Remaining Range is 1002.38              | 1. Start the vehicle  
       |      |                                                                                  | 2. Set the Rem Rng to 1002.38km                      |
|         |      |                                                                                  | 3. Verify Rng Wrn Lvl = 2                            |
| TC_06   |      | Check Remaining Range Warn Level when SOC is 102.38                           | 1. Start the vehicle  
       |      |                                                                                  | 2. Set the SOC to 102.38km                           |

45
Table 4 Manual Test Cases for Range Warn Level

<table>
<thead>
<tr>
<th>TC_07</th>
<th>Check Range Warn Level Indication when Range Warn Level is 0</th>
<th>3. Verify Rng Wrn Lvl = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1. Start the vehicle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Set the Rng Wrn Lvl = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Verify Range Warn Level Indication = none</td>
</tr>
<tr>
<td>TC_08</td>
<td>Check Range Warn Level Indication when Range Warn Level is 1</td>
<td>1. Start the vehicle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Set the Rng Wrn Lvl = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Verify Range Warn Level Indication = Amber</td>
</tr>
<tr>
<td>TC_09</td>
<td>Check Range Warn Level Indication when Range Warn Level is 2</td>
<td>1. Start the vehicle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Set the Rng Wrn Lvl = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Verify Range Warn Level Indication = Red</td>
</tr>
</tbody>
</table>

As per ISO26262 standard safety guideline for Functional Safety, to generate Test Cases, methods like analysis of equivalence classes, boundary values, and error guessing are required. This results in generating Test Cases involving at least for a min, max, min+1, max-1, threshold+1, threshold-1 values for the specified condition[39]. So, each requirement is entitled to have six Test Cases. However, in Table 4, as mentioned earlier, only a few Test Cases are manually developed because of time constraints. Whereas, for the above requirements of Range Warn Level Estimator, at least 16 Test Cases need to be developed. This value of Test Cases was obtained after analyzing the missing Test Cases needed.

7.1.3. NAT2TEST’s approach of generating Test Cases for SUT 1

<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Type</th>
<th>Expected Values</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>remaining_range</td>
<td>INTEGER</td>
<td>[10, 15, 20, 21, 100, 1003]</td>
<td>0</td>
</tr>
<tr>
<td>INPUT</td>
<td>vehicle_state</td>
<td>BOOLEAN</td>
<td>false, true</td>
<td></td>
</tr>
<tr>
<td>INPUT</td>
<td>range_warn_level_indication</td>
<td>INTEGER</td>
<td>[amber, none, red]</td>
<td>red</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>range_warn_level</td>
<td>INTEGER</td>
<td>[0, 1]</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 24 Example of assigning the initial values at Variables & Types

The initial state for generating the Test Cases have to be set, as shown in Figure 24. The values 0,14,15,17,19,20,21,999,1000,1003 for remaining range consists of both manually entered values & values from requirements. The initial state can be defined at this stage according to the requirement for which Test Cases are generated. False(negative) values of the requirement are set here so that that it can verify the fault inputs and verify the following possible values for it in the Test Cases.

The Test Cases are obtained after initializing the data for REQ002, and Table 5 shows some of the Test Cases generated for the REQ002.

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_state</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
<td>19</td>
<td>amber</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 5 shows the Test Cases for REQ002 for Range Warn Level.

Table 6 shows the Test Cases for REQ004.

Table 6 shows the Test Cases for REQ004.

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_state</th>
<th>soc_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>true</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>true</td>
<td>21</td>
<td>none</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6 shows the Test Cases for REQ004.

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_state</th>
<th>soc_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
<td>0</td>
<td>none</td>
<td>0</td>
</tr>
<tr>
<td>0.5</td>
<td>true</td>
<td>10</td>
<td>none</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6 NAT2TEST Test Cases for Range Warn Level

Table 5 represents the Test Cases, which are only for the two requirements REQ002 and REQ004. The remaining Test Cases are mentioned in chapter 10.

The tool generated 24 (in this case, each line in a Test Case file was considered as a Test Case) Test Cases for all the requirements combined so that it can satisfy the initial safety guideline. From the generated Test Cases, there were several duplicates generated when combined. Nevertheless, this drawback was solved by deleting repetitive Test Cases using a Python script. After removing the duplicates, only 14 Test Cases were unique.

7.1.4. Analysis Results for SUT 1

The comparison analysis here for this requirement set between the automated and manual approach consisted of:

- Manual approach:
  - Time taken to generate Test Cases manually.
  - Time taken to review the Test Cases manually.

- Automated approach:
  - Time taken to analyze & initialize.
  - Time taken to generate Test Cases.

In the manual approach, the data observed shows that the nine Test Cases that were developed required an average total time of around 15-30 minutes, i.e., 1.6 – 3.3 minutes to develop each Test
Case depending upon the experience level. This average data was concluded from Table 7, which consists of an average time obtained from three testers with experience levels: where “experienced” tester had more than ten years of experience in testing, “intermediate” level tester had more than two years of experience in testing and “novel” level tester had lesser than two years of experience in testing respectively. The details of the testers were anonymized for privacy reasons. Therefore, from the provided data, the average of all the three experience levels is considered, and that is 2.45 minutes for each Test Case. However, the Test Cases developed by the testers were still not enough for the initial coverage of requirements. Thus, to conquer the maximum coverage, around 16 Test Cases need to be created, and it can take around 39.2 minutes to design them all manually.

<table>
<thead>
<tr>
<th>The experience level of the Tester</th>
<th>Experienced</th>
<th>Intermediate</th>
<th>Novel</th>
</tr>
</thead>
<tbody>
<tr>
<td>The time required to analyze, design TCs and review them.</td>
<td>15-20 minutes</td>
<td>20 minutes</td>
<td>30 minutes</td>
</tr>
</tbody>
</table>

Table 7 Time taken to develop Test Cases manually for SUT 1

Initially, to perform the automated approach, each requirement should be analyzed, and the initial values must be assigned in the tool. Furthermore, for all requirements, combinedly this process can take around 5 minutes if the user has basic familiarity with the tool.

<table>
<thead>
<tr>
<th>Signal values covered</th>
<th>Number of Test Cases generated</th>
<th>Time taken (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual approach</td>
<td>&lt;0, 15, 20, 21, 1002.38, amber, red, none, 102.38</td>
<td>9</td>
</tr>
<tr>
<td>Automated approach</td>
<td>Range [0,14,15,17,19,20, 21,999,1000,1003], indication [amber, red, none], soc range [0, 10, 100, 110].</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 8 Analysis results for SUT 1

After initializing, the Test Cases were generated in less than 1.5 seconds (Time consumed by FDR: 1002ms & Time consumed by Z3: 455ms) for each requirement initialized. The results of the analysis are represented in Table 8. These Test Cases can then be used while implementing Test Cases in the execution environment. Here the time saved in comparison with the manual process is drastic. According to the study in NAT2TEST related papers, the generated Test Cases are aimed to be proven sound.

### 7.2. System Under Test 2: Hill Hold

The following set of requirements is for the Hill Hold system, a function that is commonly available in modern vehicles as an assist feature for the driver. The requirement data is perceived to be at a higher level than the previous requirements set for Range Estimator. These requirements were considered to be in the sub-system or system level.
7.2.1. Requirements for SUT 2

The below table specifies the requirements initially obtained to perform the manual process.

<table>
<thead>
<tr>
<th>Requirement Type</th>
<th>Initial Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req10</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Drive, the hill hold system shall request Brake control request front left so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req11</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Drive, the hill hold system shall request Brake control request front right so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req12</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Drive, the hill hold system shall request Brake control request rear left so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req13</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Drive, the hill hold system shall request Brake control request rear right so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req14</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Reverse, the hill hold system shall request Brake control request front left so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req15</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Reverse, the hill hold system shall request Brake control request front right so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req16</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Reverse, the hill hold system shall request Brake control request rear left so that the vehicle speed remains 0.</td>
</tr>
<tr>
<td>Req17</td>
<td>When vehicle speed is reduced to 0 km/h and vehicle movement state == Reverse, the hill hold system shall request Brake control request rear right so that the vehicle speed remains 0.</td>
</tr>
</tbody>
</table>

Table 9 Initial requirements for Hill Hold

The requirements are now documented in SysReq-CNL’s format in Table 10. Since the first half of the requirements were assigned for “drive” and the rest for “reverse”, and since the requirements specified same intention, it was reasonable to reduce them to two requirements, one for “drive” and another for “reverse” which are REQ001 and REQ002 respectively.

The REQ000 was added to specify the fault condition for the initial requirements to generate counterexamples.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>The requirements re-written in SysReq-CNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ000</td>
<td>When vehicle speed becomes greater than or equal to 1, the hill hold system shall: assign false to brake control request front left, assign false to brake control request front right, assign false to brake control request rear left, assign false to brake control request rear right.</td>
</tr>
<tr>
<td>REQ001</td>
<td>When vehicle speed becomes 0, and vehicle movement state is drive, the hill hold system shall: assign true to brake control request front left, assign true to brake control request front right, assign true to brake control request rear left, assign true to brake control request rear right.</td>
</tr>
<tr>
<td>REQ002</td>
<td>When vehicle speed becomes 0, and vehicle movement state is reverse, the hill hold system shall: assign true to brake control request front left, assign true to brake control request front right, assign true to brake control request rear left, assign true to brake control request rear right.</td>
</tr>
</tbody>
</table>
7.2.2. Manual approach of generating Test Cases for SUT 2

The below steps are the data obtained from the manual process of developing Test Cases for Hill Hold. Four Test Cases were aimed to develop manually according to priority.

Test Case #01:

Main data to observe:

1. Vehicle movement state
2. VehicleSpeed
3. Brake control request front left
4. Brake control request front right
5. Brake control request rear left
6. Brake control request rear right

Test Implementation Instructions:
Set Vehicle movement state == “drive”

During speed reducing procedure, while vehiclespeed > 10km/h Verify:
Brake control request front left == inactive (tolerance considered)
Brake control request front right == inactive (tolerance considered)
Brake control request rear left == inactive (tolerance considered)
Brake control request rear right == inactive (tolerance considered)

During speed reducing procedure, while vehiclespeed <= 10km/h Verify:
Brake control request front left == inactive (tolerance considered)
Brake control request front right == inactive (tolerance considered)
Brake control request rear left == inactive (tolerance considered)
Brake control request rear right == inactive (tolerance considered)

When Vehiclespeed < 1km/h Verify:
Brake control request front left == active(tolerance considered)
Brake control request front right == active (tolerance considered)
Brake control request rear left == active (tolerance considered)
Brake control request rear right == active (tolerance considered)

Test Case #02:

Main data to observe:

1. Vehicle movement state
2. VehicleSpeed
3. Brake control request front left
4. Brake control request front right
5. Brake control request rear left
6. Brake control request rear right

Test Implementation Instructions:
Set Vehicle movement state == “drive”

During speed reducing procedure, while vehiclespeed > 1km/h Verify:
Brake control request front left == inactive (tolerance considered)
Brake control request front right == inactive (tolerance considered)
Brake control request rear left == inactive (tolerance considered)
Brake control request rear right == inactive (tolerance considered)

When Vehiclespeed < 1km/h Verify:
Brake control request front left == active (tolerance considered)
Brake control request front right == active (tolerance considered)
Brake control request rear left == active (tolerance considered)
Brake control request rear right == active (tolerance considered)

Set Vehicle movement state == “reverse”
During speed reducing procedure, while vehiclespeed > 1km/h Verify:
Brake control request front left == inactive (tolerance considered)
Brake control request front right == inactive (tolerance considered)
Brake control request rear left == inactive (tolerance considered)
Brake control request rear right == inactive (tolerance considered)

When Vehiclespeed < 1km/h Verify:
Brake control request front left == active (tolerance considered)
Brake control request front right == active (tolerance considered)
Brake control request rear left == active (tolerance considered)
Brake control request rear right == active (tolerance considered)

Remaining Test Case data is mentioned in the appendices. Four Test Cases were created manually by the tester for two requirements. Generally, a tester can take up to 20-30 minutes to analyze and create those four Test Cases. The data from Table 11, shows that a tester with intermediate level of experience in the industry utilized 21.6 minutes to design the Test Cases and 5 minutes to review them.

<table>
<thead>
<tr>
<th>The experience level of the Tester</th>
<th>Intermediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>The time required to analyze, design TCs and review them.</td>
<td>21.6 minutes</td>
</tr>
</tbody>
</table>

Table 11 Time taken to develop Test Cases manually for SUT 2

The manual approach combinedly consisted of performing analysis of requirements data and designing the Test Cases. Then the Test Cases are reviewed for its validity. The time taken for these processes separately are represented in the chapter 7.2.4

7.2.3. NAT2TEST’s approach of generating Test Cases for SUT 2

After initializing the initial conditions in the ‘variables and types’ window, the below Test Cases are generated. Tables 12 and 13 show the two out of six Test Cases generated by the tool, and remaining Test Cases are represented in chapter 10.

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>Drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.25</td>
<td>0</td>
<td>Reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
</tbody>
</table>

Table 12 Test Case data for both drive and reverse
Table 13 Test Case data for Drive

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.25</td>
<td>10</td>
<td>Drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.5</td>
<td>20</td>
<td>Drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.75</td>
<td>0</td>
<td>Drive</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
</tbody>
</table>

After initializing for each requirement specific conditions, the Test Cases were generated in less than one second (Time consumed by FDR: 402ms & Time consumed by Z3: 213ms) for each requirement. To obtain the results similar to the manual approach, the tool had to generate six Test Cases, whereas four through the manual approach. The Test Case TC00 generated by the tool was ignored because it was considered redundant since other Test Cases generated had the same meaning as of TC00. This shortlisting was performed by a script which creates a whitelist of prioritized Test Cases, by removing the redundant and repetitive Test Cases.

### 7.2.4. Analysis Results for SUT 2

The analysis for Hill Hold requirements was performed comparing the below variables. In automated approach, the tool required an average of one second to generate Test Cases for each requirement and lesser than five minutes was required for initializing the conditions for all requirements. Hence, the combined time required to obtain Test Cases from the requirements in the automated test case generation approach was five minutes. After obtaining the Test Cases a python script was implemented to remove the redundant Test Cases. Finally, the Test Cases were implemented and executed through Test Automation.

- Automated approach:
  - Time taken to analyze and generate all Test Cases: 5 minutes
  - Number of Test Cases: 5
  - Average Time taken for execution: 5 minutes per test case

- Manual approach:
  - Time taken to analyze requirements and design Test Cases manually: 21.6 minutes
  - Time taken to review the Test Cases manually: 5 minutes
  - Number of Test Cases: 4
  - Average Time taken for execution: 5 minutes per test case

The obtained Test Cases were executed from both approaches were implemented and executed in a MIL Test Environment. As the Test Execution was performed through Test Automation, the execution time were the same for both manual and automated test case generation approaches. From the above result of both automated and manual approaches, to obtain the result similar to the manual approach, the automated approach had to consider a higher number of Test Cases. However, the time taken to generate Test Cases in an automated approach was substantially lesser. In this scenario, the execution time required for both the approaches is considered. In total, the automated approach required around 30 minutes, whereas the manual approach needed 46.6
minutes. When comparing the total execution time required by both the approaches, the difference was very similar, and it would not make a more significant impact to use the automated approach. However, when the automated approach is implemented for more complex requirements with a higher number of Test Cases, the benefit of using NAT2TEST is higher. To validate the Test Cases generated by NAT2TEST and to generate Test Cases in a better way, NAT2TEST COQ can be considered, and a brief investigation of it was explained in chapter 5.
8. Discussion & Conclusions

This chapter includes the summarization of the findings, and the research questions posed at the start of the thesis are answered.

8.1. Answering Research Questions

Figure 25 below shows the research instrument, which illustrates the background behind answering the research questions.

**RQ01:** What is the suitable strategy/tool that can be implemented in an automotive industrial practice for automatically generating Test Cases from Natural Language requirements?

By the completion of the first phase, the literature study was performed alongside closely observing the manual process. The strategy NAT2TEST was shortlisted for further evaluation in an industrial context. The motivation behind choosing this strategy was that it was specifically designed for natural language requirements, and this strategy provided the use of Controlled Natural Language as the structure to document requirements. Another reason to shortlist this tool was that it was one of the only open-source tools available, which were supported by multiple research papers. To perform automatic Test Case generation, the SysReq-CNL is implemented. Moreover, in this thesis, the process involved to re-write the requirements to SysReq-CNL is explained.

The strategy was implemented in between the requirements management tool and the test case execution tool. The input requirements to the tool were fed in from the requirements management tool, and the generated Test Cases were implemented in a Test Execution environment.

**RQ02:** How do the Test Cases generated automatically by the tool compare to the manual approach?
The Test Cases generated by the strategy were executable Test Cases in CSV format. The output data is then used to implement the Test Cases for execution. The strategy was beneficial mostly for requirements at the lower level. For detailed requirements in the system level and component level with input and output signals specified, the tool has the possibility to generate Test Cases efficiently.

8.2. Conclusion

The findings of the study have concluded that as long as the requirements satisfy SysReq-CNL, the Test Cases can be generated with some restrictions. Implementing a CNL can improve the quality of requirement documentation, and it will help in having an equal understanding of requirement specification throughout the organization.

Since the NAT2TEST strategy considered Global Clock as the variable to generate Test Cases even for non-timer related requirements, there were several redundant Test Cases generated. Moreover, if needed, the duplicate Test Cases can be removed using a simple script. The generated Test Cases were also possible to trace back to its requirement number.

The results from the analysis performed showed that for SUT 1, it required around 5 minutes to generate Test Cases, where lesser than 10 seconds was needed to generate Test Cases in the automated test case generation approach and ~22.05 minutes for the manual process. Whereas in SUT 2, when the execution time was also considered, the whole manual process took 46.6 minutes for four Test Cases. However, replicating the manual process by using the automated approach took 30 minutes.

The Test Cases were generated when the requirements were well-documented in Sub-System Level & Component Level. However, for the generated Test Cases to be executable, they had to
be transformed down to software/unit level requirements. However, without much human intervention and validation of Test Cases, if the tool is used in testing at the Unit Level, then it can be distinctly compelling. Since in Unit Level, all possible combinations need to be tested. Even invalid combinations of input variables should go through testing. This can help in saving much time and reduces future risks at a higher level. It also improves code coverage and decision coverage.

In conclusion, the tool showed higher potential towards lower-level requirements with useful Test Cases generated, and if there is a possibility to assign pre-conditions and post-conditions explicitly, then the strategy is much more usable in an industrial context.
9. Limitations and Future Work

While testing higher-level requirements, the requirements can contain several pre-conditions that need to be satisfied to run the test case. Because of that, some of the test combinations generated can be invalid. Maintaining a whitelist that can possibly exclude invalid Test Cases can be beneficial. This can be considered as future work.

A major limitation while using this CNL is the restrictions in writing requirements; some of the issues were unable to use negative values and unable to use NOT condition in output signal values.

While generating Test Cases, the strategy maps non-Boolean and non-integer signal values to Integers. For example, if the requirements have signal values as active/inactive, then they are mapped as 0/1 respectively in the Test Cases. This can only be traced in the DFRS model window. A limitation caused because of this is that when a user enters multiple input values other than the existing ones in the variables and types window, the Figure 26 shows that when “16 (entered by the user)” is selected as the initial value to generate Test Cases, then the first step (Time: 0.0) in the Test Case “16” is mapped as “4” and in the second step (Time: 0.25) “16” is not changed. This issue can cause several misunderstandings if not addressed before, or it can also be fixed in the future.

In a requirement set, if some of the requirements contain few input/output signals which are not needed/present in other requirements, then while generating Test Cases, all signals are considered for each and every requirement’s respective Test Cases. For example, if Requirement 1 has A1, B1, C1, and Requirement 2 has A1, B2, and Requirement 3 has A1 as its signals. The Test Cases generated have A1, B1, C1, B2 as signals for all requirements, this is not necessary for each requirement if the requirements are not related, and this can cause issues for safety-critical systems. The current solution to avoid this is to generate Test Cases separately as for each requirement in this scenario.

As part of the future study, the validation of the structure and standard of SysReq-CNL in the critical safety point of view can be analyzed. The generation of Test Cases can follow certain standards for requirement coverage in automotive software testing such as ISO26262, ISTQB & MC/DC. According to the author of NAT2TEST, other criteria that can be considered while setting a standard to generate Test Cases are, such as (node/edge) coverage of the underlying LTS and also consider MC/DC with respect to the conditions of the requirements. Nevertheless, the earlier mentioned alternatives are considered as future implementations.
To validate the accuracy of Test Cases generated, the NAT2TEST COQ is introduced, where the mutation score of Test Cases are provided. Another future study of this thesis is to evaluate the completely integrated NAT2TEST COQ tool once it is completed.
References


10. Appendix

In this chapter, the test case data from both the manual and automated approach is illustrated here. These data are referenced back to chapter 7.

A.1. Remaining Range Estimator NAT2TEST Test Cases:

Here, the remaining Test Cases generated by the tool is shown here. These Test Cases are for requirements REQ001,003,005,006 & 007

<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.25</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>21</td>
<td>none</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.25</td>
<td>15</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1003</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.25</td>
<td>14</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
<tr>
<td>0.75</td>
<td>15</td>
<td>red</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>none</td>
<td>0</td>
</tr>
<tr>
<td>0.25</td>
<td>15</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>999</td>
<td>none</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>remaining_range</th>
<th>range_warn_level_indication</th>
<th>range_warn_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>0.25</td>
<td>20</td>
<td>amber</td>
<td>1</td>
</tr>
</tbody>
</table>
**A.2. Hill Hold Manual Test Cases:**

The remaining Test Cases from 7.2.2, that was designed by the tester is shown here.

**Test Case 03:**

Main data to observe:

1. Vehicle movement state
2. VehicleSpeed
3. Brake control request front left
4. Brake control request front right
5. Brake control request rear left
6. Brake control request rear right

Test Implementation Instructions:
Set Vehicle movement state == “reverse”

During speed reducing procedure, while vehiclespeed > 10km/h Verify:
- Brake control request front left == inactive (tolerance considered)
- Brake control request front right == inactive (tolerance considered)
- Brake control request rear left == inactive (tolerance considered)
- Brake control request rear right == inactive (tolerance considered)

During speed reducing procedure, while vehiclespeed <= 10km/h Verify:
- Brake control request front left == inactive (tolerance considered)
- Brake control request front right == inactive (tolerance considered)
- Brake control request rear left == inactive (tolerance considered)
- Brake control request rear right == inactive (tolerance considered)

When Vehiclespeed < 1km/h Verify:
- Brake control request front left == active (tolerance considered)
- Brake control request front right == active (tolerance considered)
- Brake control request rear left == active (tolerance considered)
- Brake control request rear right == active (tolerance considered)

**Test Case 0:**

Main data to observe:

1. Vehicle movement state
2. VehicleSpeed
3. Brake control request front left
4. Brake control request front right
5. Brake control request rear left
6. Brake control request rear right

Test Implementation Instructions:
Set Vehicle movement state == “reverse”

During speed reducing procedure, while vehiclespeed > 1km/h Verify:
- Brake control request front left == inactive (tolerance considered)
- Brake control request front right == inactive (tolerance considered)
- Brake control request rear left == inactive (tolerance considered)
Brake control request rear right == inactive (tolerance considered)  
When Vehiclespeed < 1km/h Verify:  
Brake control request front left == active(tolerance considered)  
Brake control request front right == active (tolerance considered)  
Brake control request rear left == active (tolerance considered)  
Brake control request rear right == active (tolerance considered)  

Set Vehicle movement state == “drive”  
During speed reducing procedure, while vehiclespeed > 1km/h Verify:  
Brake control request front left == inactive (tolerance considered)  
Brake control request front right == inactive (tolerance considered)  
Brake control request rear left == inactive (tolerance considered)  
Brake control request rear right == inactive (tolerance considered)  
When Vehiclespeed < 1km/h Verify:  
Brake control request front left == active(tolerance considered)  
Brake control request front right == active (tolerance considered)  
Brake control request rear left == active (tolerance considered)  
Brake control request rear right == active (tolerance considered)

A.2. Hill Hold NAT2TEST Test Cases:

The remaining Test Cases from 7.2.3, that was generated by the tool are displayed below.

TC02:

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>reverse</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.25</td>
<td>10</td>
<td>drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
</tbody>
</table>

TC03:

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.25</td>
<td>10</td>
<td>drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
</tbody>
</table>

TC04:

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
<tr>
<td>0.25</td>
<td>10</td>
<td>reverse</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.5</td>
<td>20</td>
<td>reverse</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
</tr>
<tr>
<td>0.75</td>
<td>0</td>
<td>reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
</tbody>
</table>

TC05:

<table>
<thead>
<tr>
<th>TIME</th>
<th>vehicle_speed</th>
<th>vehicle_movement_state</th>
<th>brake_control_request_rear_left</th>
<th>brake_control_request_front_left</th>
<th>brake_control_request_front_right</th>
<th>brake_control_request_rear_right</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>drive</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>10</td>
<td>reverse</td>
<td>inactive</td>
<td>inactive</td>
<td>inactive</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>reverse</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td></td>
</tr>
</tbody>
</table>

Table 15 Remaining Test Cases from Hill Hold
Disclaimer

The report is based on the study and investigations carried out at NEVS AB. NEVS is neither responsible nor liable for the accuracy or the claims made if any, in this report. Any use of NEVS specific information/data outside this report, without prior permission from NEVS, is forbidden.