FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Bakalářská práce

Automatic infotainment testing

Studijní program: B0714A270001 Mechatronika
Autor práce: **Michal Sojka**
Vedoucí práce: Ing. Ekaterina Nyrobtseva
Ústav mechatroniky a technické informatiky

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání bakalářské práce

Automatic infotainment testing

Jméno a příjmení:	**Michal Sojka**
Osobní číslo:	M21000054
Studijní program:	B0714A270001 Mechatronika
Zadávající katedra:	Ústav mechatroniky a technické informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Familiarize yourself with the options for testing display units, especially in terms of communication interfaces and used protocols.
2. Get acquainted with the possibilities of creating automated tests.
3. Create an automated test and implement a tool for evaluating test results.
4. Describe the workflow for implementing fixes.

Rozsah grafických prací:	dle potreby dokumentace
Rozsah pracovní zprávy:	30 až 40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	angličtina

Seznam odborné literatury:

[1] DOLEŽAL, Jan. *Agilní přístupy vývoje produktu a řízení projektu: komplexně, prakticky a dle světové praxe*. Praha: Grada, 2022. ISBN 978-80-271-3705-3.

[2] BUREŠ, Miroslav; RENDA, Miroslav; DOLEŽEL, Michal; SVOBODA, Peter; GRÖSSL, Zdeněk et al. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. Profes!onal. Praha: Grada, 2016. ISBN 978-80-247-5594-6.

Vedoucí práce:	Ing. Ekaterina Nyrobtseva Ústav mechatroniky a technické informatiky
Datum zadání práce:	12. října 2023
Předpokládaný termín odevzdání:	14. května 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. Ing. Josef Černohorský, Ph.D.
garant studijního programu

V Liberci dne 12. října 2023

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

30. dubna 2024 Michal Sojka

Automatické testování infotainmentu

Abstrakt

Tato bakalářská práce se zaměřuje na automatizované testování infotainment systémů ve vozidlech. Práce poskytuje teoretický popis základů automatizovaného testování. Základ tvoří architektura vozu, sběrnice CAN a skriptovacího jazyka Tcl. Seznámení s procesem vývoje grafického uživatelského rozhraní je doplněno motivací ke zlepšení těchto procesů. Na základě této motivace práce popisuje principy automatizovaného testování a jeho začlenění do procesu vývoje grafického rozhraní. Zahrnuje dokumentaci testovacího stavu (Test Bench) včetně hardwarových a softwarových komponent. Je použit jazyk Tcl k vytváření testovacích funkcí a scénářů (TestCases) k identifikaci chyb. Dále práce popisuje celý proces testování a uvádí jeho výsledky.

Klíčová slova: Tcl, CAN, Testování, Infotainment, GUI, Automatizace, HMI, CANoe

Automatic Infotainment Testing

Abstract

This bachelor's thesis focuses on the automated testing of in-vehicle infotainment systems. The thesis provides a theoretical description of the basis for automated testing. The basis consists of the vehicle architecture, the CAN bus and the Tcl scripting language. Familiarity with the GUI development process is accompanied by a motivation to improve these processes. Based on this motivation, the bachelor thesis describes the principles of automated testing and its integration into the graphical interface development process. The work includes documentation of the test bench (test rack), including hardware and software components. The Tcl language is used to create test functions and scenarios (TestCases) for identifying bugs in a graphical interface. The thesis describes the entire testing process and reports the results.

Keywords: Tcl, CAN, Testing, Infotainment, GUI, Automation, HMI, CANoe

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Ing. Ekaterina Nyrobtseva, for her exceptional guidance throughout the development of this thesis. Her meticulous review process and insightful feedback have significantly improved the clarity and quality of this work. I am particularly grateful for her perseverance in refining the thesis and her assistance in correcting my grammatical oversights.

I would also like to express my sincere thanks to Digiteq Automotive for providing the necessary environment to carry out this project, and to the members of the DTF-T Front-End Testing department for their invaluable support and willingness to share their expertise during my research.

Special thanks go to Ing. Tomas Zimmerhakl for his close cooperation and for providing essential information that contributed greatly to this thesis.

Contents

List of abbreviations . . . 12

1 Introduction **18**

1.1 Project Background . . . 19

1.2 Motivation . . . 19

1.3 Bachelor Thesis Goals . . . 20

2 Technical Background **22**

2.1 Electronic Control Units in Automotive Systems . . . 22

2.1.1 The Necessity of ECU Testing in Modern Vehicles . . . 22

2.2 Overview of In-Vehicle Infotainment Systems . . . 23

2.3 Evolution of In-Vehicle Infotainment Testing . . . 25

2.4 Testing Methods . . . 26

2.4.1 Manual Testing . . . 26

2.4.2 Automated Testing . . . 27

2.4.3 Regression and Performance Testing Methods for In-Vehicle Infotainment Systems . . . 28

2.5 Role of Automated Testing in Software Development . . . 29

2.6 Functional Safety in In-Vehicle Infotainment Systems: A Focus on ISO 26262 . . . 30

3 Integration of Automated Testing **32**

3.1 GUI Development Process in In-Vehicle Infotainment Systems . . . 32

3.2 Life Cycle of the Screen . . . 33

3.3 Motivation for Improving GUI Development Processes . . . 37

3.4 Benefits of Integrating Automated Testing in GUI Development . . . 38

4 Foundations of Automotive Testing **40**

4.1 Introduction to CAN Bus . . . 41

4.1.1 CAN Bus Architecture . . . 41

4.1.2 CAN Bus Protocols . . . 44

4.2 Basics of CAN Bus in Automotive Testing . . . 48
4.2.1 Comprehensive Testing Methods for CAN Bus Systems . . . 49
4.2.2 Integration of On-Board Diagnostics in CAN Bus Systems . . . 50
4.2.3 Types of CAN Buses in Vehicles . . . 50
4.3 Introduction to Tcl Programming . . . 51
4.3.1 Overview of Tcl Language . . . 51
4.3.2 Advantages of Using Tcl for Infotainment System Testing . . . 53
4.3.3 Programming in Tcl . . . 54

5 Test Bench **55**
5.1 Overview of the Test Bench . . . 56
5.2 Hardware Components . . . 57
5.2.1 Windows PC . . . 57
5.2.2 Grabber . . . 58
5.2.3 CAN Case . . . 59
5.2.4 Manson Power Supply with Remote Control . . . 59
5.2.5 12V Power Supply . . . 60
5.2.6 PCAN-PCI Express . . . 60
5.2.7 LED Bar Signalisation . . . 60
5.2.8 Two Phones (One Android and One with iOS) . . . 61
5.2.9 Front Panel with All CAN Buses . . . 61
5.2.10 Quido by Papouch . . . 62
5.2.11 UPS . . . 63
5.2.12 Vehicle Units . . . 64
5.3 Software Components . . . 65
5.3.1 CANoe . . . 65
5.3.2 Grimr . . . 66
5.3.3 TestAut2 . . . 66
5.3.4 Git Extensions . . . 66
5.4 System Architecture of Test Bench . . . 67

6 Implementation **69**
6.1 Understanding Code Structure . . . 69
6.1.1 Implementation Strategy . . . 72
6.2 Test Environment Setup . . . 73
6.3 Writing and Preparation of Test Scenarios . . . 75
6.3.1 Test Scenario Structure . . . 75
6.4 Setting Up the Screen . . . 77

6.4.1 Click Functions . . . 78
6.4.2 Checkbox Controls . . . 79
6.4.3 Dropdown Controls . . . 80
6.4.4 Slider Controls . . . 81
6.5 Screen Capture . . . 82
6.5.1 Screen Splitting . . . 83
6.5.2 Capturing Modes . . . 85
6.5.3 Understanding the 'endCondition' Parameter . . . 86
6.6 Comparison of New Images with Reference and Detection of Differences 87
6.6.1 Case Study: Detecting and Analyzing Interface Discrepancies in HMI Updates . . . 89
6.6.2 Case Study: Identifying and Resolving Interface Bugs in HMI Updates . . . 91
6.7 Final Report on Automated Infotainment System Testing . . . 93

7 Results and Analysis **96**
7.1 Test Automation Results . . . 96
7.2 Limitations . . . 97
7.3 Test Automation Cost . . . 98
7.3.1 Exclusions in Cost Estimation . . . 99
7.3.2 List of Components and Their Costs . . . 99

8 Future Directions **100**
8.1 Transition to Android-Based Infotainment Systems . . . 100
8.2 Future Research and Development . . . 100
8.3 Speculative Outlook . . . 101

9 Conclusion **102**

References **104**

A Appendices **109**
A.1 Tcl Syntax . . . 109
A.1.1 Loops in Tcl . . . 117
A.1.2 Mastering Functions in Tcl using 'proc' . . . 123
A.1.3 Harnessing the Power of Dictionaries for Efficient Data Management in Tcl . . . 124
A.1.4 Using Namespaces in Tcl for Modular Programming . . . 126
A.1.5 Leveraging Lists in Tcl . . . 128

A.1.6 Arrays 128
A.2 ImageMagick: A Powerful Tool for Image Processing in Automated Testing 129

B Appendices **131**
B.1 Attached Files 131

List of abbreviations

ABT Anzeigebedienteil
AR Augmented Reality
ASIL Automotive Safety Integrity Levels
CAN Controller Area Network
CRC Cyclic Redundancy Check
ECU Electronic Control Unit
GUI Graphical User Interface
HMI Human Machine Interface
ID Identifier
IoT Internet of Things
IVI In-Vehicle Infotainment
MEB Modularer Elektrifizierungsbaukasten
MGB Modular FrameGrabber
OBD On-Board Diagnostics
Tcl Tool Command Language
TCs Test Cases
UI User Interface
V2X Vehicle-to-Everything
VW Volkswagen

List of Figures

2.1 An example of Škoda Infotainment tested with automated methods . 24

3.1 Life cycle of a GUI screen in infotainment system development 36
3.2 Flowcharts showing the different screen life cycle processes of manual and automated testing in the GUI development 38

4.1 CAN bus twisted pair cable 42
4.2 Signal levels in a CAN Bus 43
4.3 CAN bus termination resistors 43
4.4 The figure shows the structure of a CAN 2.0A (Standard Format) message frame, detailing the fields involved in data transmission . . . 45
4.5 The figure shows the structure of a CAN 2.0B (Extended Format) message frame, detailing the fields involved in data transmission . . . 46

5.1 Test bench for automated testing 55
5.2 High-performance Windows PC 57
5.3 Modular FrameGrabber (MGB) 58
5.4 Vector CAN Case 59
5.5 Power supply of Manson company 59
5.6 12V power supply 60
5.7 Example of the front panel implemented in a test bench 61
5.8 Front panel connection design 62
5.9 Quido ETH 2/16: 2 inputs, 16 outputs and thermometer 63
5.10 Uninterruptible power supply (UPS) 63
5.11 Škoda Enyaq infotainment unit 64
5.12 Škoda Enyaq 13" ABT 64
5.13 Škoda Enyaq Gateway unit 65
5.14 Diagram of the power source of the test bench components 67
5.15 Hardware connection schematic 68
5.16 Software connection diagram 68

6.1 The code hierarchy within the HMI library 69
6.2 Example of code hierarchy for F327 project (Enyaq) 71
6.3 Interface of TestAut2 73
6.4 Illustration of the 'OFF' and 'ON' status of checkboxes 79
6.5 Displaying the transition states of checkboxes for system preferences . 79
6.6 View of the dropdown control in the Enyaq infotainment system . . . 80
6.7 Expanded view of the Speed Alert settings dropdown 80
6.8 Audio settings slider controls 81
6.9 Infotainment display segmentation 84
6.10 Focused testing of the driving data homescreen tile 84
6.11 Exact match verification: Green indicates identical screens 87
6.12 Minor discrepancy detected: Yellow for under 1% pixel difference . . 87
6.13 Significant variation: Red for over 1% pixel difference 88
6.14 Manual review required: Blue when reference is missing 88
6.15 Error indication: Highlighted when a test issue occurs 88
6.16 Reference image 90
6.17 Image of new HMI update 90
6.18 Differential image 90
6.19 Reference image 91
6.20 Image of new HMI update 91
6.21 Differential image 92
6.22 An example of the numerical result of one of the tests 93
6.23 Sample part of the final report 95

List of Tables

6.1 TestCase parameters description 75
6.2 Overview of capturing modes 85

A.1 Overview of fundamental operations in Tcl 113
A.2 Summary of Tcl 'dict' command options 126

Listings

6.1 Testing the GUI of the headlight controls on the IVI's touchscreen . . 76
6.2 Implementation example for endCondition 86
A.1 Comments example 109
A.2 Command example 109
A.3 Command to print "Hello Tcl World" in Tcl 110
A.4 Variable substitution 110
A.5 Example of command substitution 110
A.6 Example of removing variable 111
A.7 Example of expr command in Tcl 111
A.8 Implementing basic arithmetic operations 112
A.9 Simulating volume adjustment based on vehicle speed 112
A.10 Implementing conditional logic 113
A.11 Example of using bitwise operators 115
A.12 Example of implementing relational operations 116
A.13 Template of a Tcl 'for' loop 117
A.14 Example of 'for' loop in Tcl 117
A.15 Template of a Tcl 'foreach' loop 118
A.16 Example of 'foreach' loop in Tcl 119
A.17 Output of 'foreach' example 119
A.18 Template of a Tcl 'while' loop 119
A.19 Programming your way to graduation: A Bachelor's thesis progress simulator in Tcl 120
A.20 Template of a Tcl 'switch' command[42] 121
A.21 Practical example of a Tcl 'switch' command 122
A.22 Template of a Tcl 'proc' command 123
A.23 Template of calling procedures 123
A.24 Example of a simple procedure 123
A.25 Example of a simple procedure with 'return' command 124
A.26 Example of creating a dictionary 124

A.27 Example to access a value in a dictionaryl 124
A.28 Example of 'dict set' and 'dict unset' commands 125
A.29 Example of 'dict for' command 125
A.30 Example of creating a namespace in Tcl 127
A.31 Example of importing a namespace 127
A.32 Examples of creating a list 128
A.33 Example of accessing element in list 128
A.34 Example of lappend and linsert commands 128
A.35 Initializing test outcomes in an associative array 129
A.36 Example of accessing array elements 129

1 Introduction

It took humanity almost 200,000 years to invent the wheel. Six thousand years later, humankind tainted the Earth's surface with first cars and 83 years after that; humankind reached the Moon's surface. Today, technology is advancing at an incredible pace, affecting every industry, including the automotive. In the context of increasing competition and ever-changing consumer preferences, the automotive industry has begun to look for innovative ways to attract and retain the attention of its customers. One of the key elements that has become an integral part of modern vehicles is the infotainment.

Modern in-vehicle systems not only give vehicles a luxurious look but also provide an interactive experience that meets the needs and expectations of drivers and passengers. This segment of the car is changing the way people experience their journeys. Today, in-vehicle infotainment is more than just a radio that plays radio tunes. It provides the user with the complete experience of information and entertainment. From intuitive touchscreens and voice control to connectivity with smart devices. It gives the driver greater awareness of the car's behaviour, allowing them to control a range of systems including handling, comfort systems such as air conditioning and entertainment systems such as listening to music or podcasts[31].

Infotainment is becoming an increasingly complex and sophisticated part of the car. As a result, there is an increasing need to ensure the reliability and functionality of these systems. Individual systems are tested for functionality. But an essential part of this is transferring this information to the car's display, the direct interface between the user and the infotainment system. This graphical display requires proper testing. This process is known as Human Machine Interface (HMI) testing. Failure to display the correct information could prevent one of the systems from working properly or make it impossible to set up. It can also affect the overall driving experience.

IoT connectivity is also becoming increasingly common in the automotive sector. Modern in-vehicle infotainment systems connect with all the smart automotive technologies like Advanced Driver-Assistance Systems, V2X connectivity solutions,

telematics devices, smartphones, sensors, etc., and integrate them to provide a great driving experience. This integration leads to more screens showing on the car's display, subsequently increasing the demand to test. Today's cars can display up to 1,000 of these unique screens[31].

1.1 Project Background

Building on the theoretical foundations set out in the introduction, this section looks at the specific environment in which the research and development for this thesis unfolded - Digiteq Automotive.

Digiteq automotive is dedicated to developing automotive innovations such as autonomous driving, connectivity, electromobility, and digitalisation. The company was founded in 2001 and has become a strategic partner to members of the Volkswagen (VW) Group.

In response to the increasing demands on software and electronic systems in automobiles, Digiteq Automotive focuses on testing methodologies. It combines modern approaches with proven testing methods to ensure efficiency, sustainability and maximum value for their customers. Testing services include component testing, integration testing and full vehicle testing using virtual reality, virtualisation and simulation[9].

The automotive industry is undergoing a period of transformation driven by rapid technological advances. In-vehicle infotainment (IVI) systems are becoming more sophisticated and complex, and the process of testing these systems is becoming more demanding. However, the customers demand for the quality and test agility to remain the same, and traditional test methodologies cannot keep pace with the complexity of today's IVI systems.

Therefore, test methodologies must evolve to maintain their quality and financial viability. An automated testing system would reduce the need for human resources and lead times, while improving quality, efficiency and performance[28, 1].

1.2 Motivation

Part of the development of the infotainment's graphical user interface (GUI) is testing. This has mainly been done manually, with the tester going through the screens in several rounds and visually checking them. However, as mentioned above, IVI systems are advancing in complexity and in the richness of their content. As a result, the testing process is becoming more tedios and more expensive. This is

reflected in the final price of the car. The aforementioned system dynamics also increase the chances of human error and the tester missing some bugs. The manual testing process and its limitations are described in subsection 2.4.1.

This is why most companies today are trying to introduce an element of automation into their processes. A machine can easily perform less demanding or repetitive tasks. When automation is well integrated, processes can be streamlined, and large numbers of screens can be tested on a regular basis in shorter periods compared to manual testing. Last but not least, automated testing is generally cheaper than manual testing, making it also financially beneficial for companies[19].

1.3 Bachelor Thesis Goals

The overarching goal of this Bachelor's thesis is to harness analytical skills for developing an array of test functions and scenarios. These will establish a foundation for future progress in the field of automated testing of IVI systems. The core objective of these tests is to meticulously uncover and catalog bugs in the infotainment GUI. Beyond mere detection, this work aims to outline the workflows for rectifying identified issues, thereby enhancing system reliability.

In parallel, the thesis aspires to inject innovative ideas and systematic optimisations into the testing process. These improvements are envisioned to streamline procedures, bolster test efficiency, and yield cost-effective strategies in automated testing.

Specific Goals

1. **Theoretical Framework**
   Provide a theoretical description of key components, including vehicle architecture, Controller Area Network (CAN) bus and Tcl scripting language, to provide a basis for automated testing.

2. **Integration into GUI Development**
   Gain an understanding of the GUI development process and identify issues and opportunities for improvement.

3. **Test Bench Documentation**
   Create documentation of the test bench, including hardware and software components, to enable a full understanding of the test environment.

4. **Results Reporting**
   Describe detailed reporting of automated testing results, including identification and classification of errors, to ensure transparency and accountability.

Subgoals

1. **Efficiency and Cost-effectiveness**
   The aim is to evaluate the efficiency and cost-effectiveness of automated testing compared to manual testing, and to highlight the potential benefits to organisations in terms of time, accuracy and overall financial savings.

2 Technical Background

A detailed examination of automated testing for IVI systems reveals a dynamic interplay of disciplines. The precision of software engineering, the analytical rigor of system analysis, and the creative insights of user experience design all contribute to developing technologies that enhance the functional aspects of infotainment systems and their interaction with users. The convergence of knowledge and technique is crucial for advancing the capabilities of these systems, with the goal of achieving optimal performance and reliability in the rapidly evolving automotive industry.

2.1 Electronic Control Units in Automotive Systems

An Electronic Control Unit (ECU) is a critical component in modern vehicles, acting as the brain behind their many automated features and systems. An ECU is essentially a microcontroller-based device embedded in the vehicle's electronics that manages and controls a wide range of vehicle functions. These include engine management, transmission operation, airbag deployment and much more. As automotive technology has advanced, the complexity and number of ECUs in a vehicle has increased significantly, reflecting the greater integration of digital control technologies into automotive design[47].

2.1.1 The Necessity of ECU Testing in Modern Vehicles

Testing ECUs is paramount for several reasons:

- **Functionality Assurance:** Ensuring that each ECU performs its designated functions correctly, thereby guaranteeing the vehicle operates as intended.
- **Safety Verification:** Vehicles rely heavily on ECUs for the management of critical safety systems. Testing ensures these units operate reliably under all conditions, protecting passengers from system failures.

- **Compliance and Standards:** Automotive manufacturers must adhere to stringent regulatory standards, which include rigorous testing of ECUs to meet safety, emissions, and operational guidelines.
- **Reliability and Durability:** ECUs control systems essential for the long-term performance and durability of the vehicle. Testing helps identify potential failures that could degrade the vehicle's functionality over time.
- **Software Integration:** With the increasing role of software in automotive systems, ECUs must be tested to ensure new software updates integrate seamlessly without disrupting existing functionalities.

As vehicles become more integrated with electronic systems, the role of ECUs becomes increasingly important. One of the most important ECUs in modern vehicles is the infotainment unit. This system enhances the driving experience by integrating entertainment, information delivery and user interface technologies into a single system. Testing the infotainment ECU is critical not only for functionality and user experience, but also for its interactions with other vehicle systems to ensure that it operates seamlessly within the broader automotive ecosystem.

Given their centrality to both driver engagement and vehicle functionality, infotainment units present a unique mix of challenges and opportunities for automotive technology developers. The following sections delve into the specific aspects of in-vehicle infotainment systems, exploring their development, integration and the specialised testing methodologies developed to ensure these systems meet both user expectations and stringent automotive standards.

2.2 Overview of In-Vehicle Infotainment Systems

Automotive infotainment systems have evolved from simple radios to sophisticated multimedia centres, transforming vehicles into interactive environments. Infotainment, a portmanteau of 'information' and 'entertainment', refers to the vehicle system that provides essential driving information and guidance. The IVI system is an integrated system that supports automobile navigation, connection with digital multimedia broadcasting, instrument panel, radio, multimedia, various sensors, and external devices[6].

IVI systems have changed the way we perceive the road and driving. Travelling is no longer just about getting from A to B. By using and interacting with these systems, the car crew can also make time on the road more enjoyable. As illustrated in Figure 2.1, modern infotainment systems integrate a variety of functionalities, seamlessly combining entertainment and information delivery to enhance the driver's experience.

One of the most important elements of IVI is the user interface (UI), which should be clear and intuitive for the crew. The UI of an IVI system is the medium through which drivers and passengers interact with the myriad of features on offer. In the past, there were physical knobs and buttons in the car. Today, these are increasingly being replaced by sophisticated touch screens and voice recognition systems.

As technology advances, IVI systems are no longer an isolated unit but are integrated to communicate with external networks and technologies. These new trends include augmented reality (AR) displays, Artificial Intelligence-based personalisation or integration with smart home devices[33].

Figure 2.1: An example of Škoda Infotainment tested with automated methods[34]

Despite these advances, IVI systems encounter significant challenges. They face issues related to user distraction, cyber security and the rapid pace of technology obsolescence. Most IVI units are delivered from the factory, and correct operation must be ensured throughout the life of the vehicle, which is a very difficult challenge.

Third-party applications must be run in an environment that does not interfere with the vehicle and compromise passenger safety. These and many other obstacles then present challenges that developers and designers must look at and consider during development[22].

2.3 Evolution of In-Vehicle Infotainment Testing

Initially, IVI systems were relatively simple, comprising basic audio and radio functionalities. Testing these systems primarily involved manual methods where testers would interact with the infotainment system and manually check each function. This manual testing process included checking user interfaces, audio outputs, radio functionality and basic connectivity features. Testers had to physically manipulate the systems' controls, such as buttons and knobs, to ensure that each function worked as intended. While straightforward, this approach was time-consuming and prone to human error, especially as systems became more complex with the integration of features such as navigation and Bluetooth connectivity.

With the advent of advanced technologies, IVI systems began to incorporate sophisticated features such as touchscreen interfaces, voice recognition and integration with smartphones and other devices. The complexity of testing increased significantly as these systems now required validation of complex software algorithms, user interface responsiveness and seamless connectivity with external devices[32].

To address these challenges, the industry has begun to move towards automated testing methodologies. Automated testing involves the use of software tools and scripts to perform tests on the infotainment system without the need for constant human intervention. For example, a common automated test might involve a script that simulates user input on a touchscreen interface to test its responsiveness and accuracy. Other automated tests could include voice command recognition, Bluetooth connectivity checks and performance assessments under various simulated conditions[46, 48].

2.4 Testing Methods

In the field of IVI system development, testing methods play a pivotal role in ensuring the functionality, reliability, and user experience of these intricate systems. This section outlines the various approaches employed to meticulously evaluate each aspect of infotainment systems, encompassing both manual and automated testing strategies. By examining these methodologies, we gain a comprehensive understanding of how each testing type contributes to the final product's efficacy and safety.

2.4.1 Manual Testing

Manual testing is a traditional approach where testers interact directly with the infotainment system and perform tests manually. This method often requires a tester to interact with the system as a regular user would, checking for usability, functionality and any anomalies.

Key Aspects of Manual Testing:

1. **User Experience Evaluation**
   Testers assess the system's interface for user-friendliness, responsiveness, and intuitiveness.

2. **Physical Interaction**
   Testers engage with physical components like buttons, touchscreens, and knobs to ensure their proper function.

3. **Observation-Based**
   This method relies heavily on the tester's attention to detail and ability to notice issues.

Limitations:

- **Time-consuming Review:** The process can be lengthy and laborious, requiring a significant investment of human resources.

- **Vulnerability to Human Error:** Although skilled, testers are susceptible to the vagaries of human nature, including oversights, particularly when personal concerns interfere with professional focus.

- **Inconsistency in Repetition:** Humans, unlike machines, may struggle to maintain consistent performance across repetitive tasks or extensive test suites, with fatigue and loss of concentration affecting the consistency of results.

2.4.2 Automated Testing

Automated testing uses software tools and scripts to perform tests on the infotainment system without constant human intervention. It is particularly useful for repetitive tasks, regression testing and scenarios that are difficult to simulate manually.

Key Aspects of Automated Testing:

1. **Scripted Scenarios**
   Testing is performed using pre-defined scripts that simulate user input and system interactions.

2. **Regression Testing**
   Automated testing handles regression testing efficiently, ensuring that new changes do not break existing functionality.

3. **Performance Testing**
   Can simulate demanding conditions to test system performance and stability under load.

4. **Consistency**
   Provides consistent test execution, eliminating human error.

The benefits:

- **Efficiency and Speed:** Automated testing is more efficient and faster than manual testing, allowing for rapid validations.
- **Continuous Operation:** Can run continuously and handle large volumes of tests without the need for constant human oversight.
- **Suitability for Complexity:** Ideal for complex systems that have extensive testing requirements due to their intricate functionalities.

Challenges:

- **Initial Setup and Development:** The initial setup and script development can be time-consuming, requiring careful planning and resource allocation.
- **Ongoing Maintenance:** Requires consistent maintenance to keep scripts up to date with system changes, ensuring reliability.

- **Limitations in Experience Capture:** May not adequately capture user experience or the visual aspects of the interface, potentially missing subtle yet critical user interactions.

Figure 3.2 provides a comparative overview of manual versus automated testing processes, highlighting the efficiency gains achieved through automation.

2.4.3 Regression and Performance Testing Methods for In-Vehicle Infotainment Systems

When testing IVI systems, it is important to use a variety of test methods to ensure that the system meets the high standards expected by the automotive industry. In automotive infotainment, various testing methodologies are employed, including unit testing, integration testing, and system testing. These techniques complement our focus on regression and performance testing, ensuring a comprehensive evaluation of system robustness and functionality.

Regression Testing

Regression testing is a software testing method that verifies the correct functioning of previously developed and tested software after it has been modified or interfaced with other software. In the context of IVI systems, regression testing is crucial whenever updates are made to the system's software, including the implementation of new features, bug fixes, or performance improvements. The primary goal of regression testing is to identify any unintended side effects caused by the latest code changes and to ensure that the new software version does not regress in terms of functionality and stability.

1. **Purpose**
   The purpose of this task is to verify that updates or changes have not negatively impacted existing functionalities. This task applies to any software update or change.

2. **Methodology**
   The methodology typically involves rerunning a set of predefined tests on the updated software to compare the results with previous test runs. Automated testing tools are often used to facilitate this process, given the repetitive nature of the tests.

3. **Application**
   After software updates, patches, or enhancements are applied, system stability and functionality are validated.

Performance Testing

Performance testing evaluates the speed, responsiveness, and stability of a system under a particular workload. It is critical for IVI systems where users expect a seamless and responsive interface for navigation, entertainment, and communication features. Performance testing helps identify bottlenecks and areas for optimisation to ensure the system meets the performance standards required for a smooth user experience.

1. **Purpose**
   To assess the system's responsiveness, stability, and scalability, ensuring it can handle the expected load with acceptable performance levels.

2. **Methodology**
   Includes simulation of various scenarios that an infotainment system may encounter during its lifetime, such as system operation in extreme weather conditions, increased workload due to complex operations, frequent device reboots, or high user activity. System performance is then evaluated, including load time, processing speed, and memory usage.

3. **Application**
   Essential during the development phase to benchmark the system's performance and after any significant updates to ensure the new features do not degrade the system's overall performance.

2.5 Role of Automated Testing in Software Development

Automated testing improves the process of ensuring software quality by moving beyond traditional, manual testing approaches. It provides a systematic, repeatable, and scalable solution to meet the rigorous demands of modern software development. Through the use of specialised software tools, automated testing enables the execution of a series of tests on the software application to assess its functionality, performance, and compliance with specified requirements. Unlike manual testing, which relies on human effort to perform tests, automated testing uses scripts and tools to perform these tests efficiently and consistently over multiple iterations.

A key benefit of automated testing is its ability to be reused. Once a TestCase (TC) has been developed, it can be executed repeatedly over time, providing consistent results. This reusability not only saves significant time and resources but also ensures a higher level of test accuracy throughout the software life cycle.

The role of automated testing is essential in the development of IVI systems. It serves as a cornerstone for improving software quality, accelerating development schedules, and effectively managing the inherent complexity of the system. Automated testing enables developers to consistently achieve and maintain a superior standard of reliability and performance. This is critical to meeting the ever-increasing expectations of consumers in the automotive industry, where the sophistication and integration of infotainment systems are constantly evolving.

2.6 Functional Safety in In-Vehicle Infotainment Systems: A Focus on ISO 26262

Functional safety is a crucial aspect of automotive development. It ensures that electronic and electrical systems, including IVI systems, operate safely even in the event of system failures. **ISO 26262**, titled '**Road vehicles - Functional safety**', is an international standard for ensuring the functional safety of electrical and electronic systems in production automobiles. This section examines the significance of **ISO 26262** in relation to IVI systems, with a focus on its requirements, implementations, and impact on safety protocols[30].

ISO 26262 Overview

ISO 26262 provides a framework for ensuring system safety throughout the life cycle of automotive development, from design to decommissioning. It addresses possible hazards caused by malfunctioning electronic systems and prescribes requirements for safety management, development, production, operation, service, and decommissioning[37].

Applicability to In-Vehicle Infotainment Systems

ISO 26262 primarily focuses on critical automotive systems such as steering and braking. However, it also applies to infotainment systems due to their indirect impact on vehicle safety. The standard examines how unexpected behaviors in an infotainment system might distract the driver or malfunction in ways that could lead to safety risks[30].

Safety Integrity Levels

ISO 26262 defines several Automotive Safety Integrity Levels (ASILs), which are used to classify the necessary safety measures required to handle potential risks. The levels range from ASIL A (lowest) to ASIL D (highest), depending on the severity, exposure, and controllability of the hazard. However, quality management practices are essential to ensure that the system is reliable, user-friendly, and does not distract the driver or interfere with the operation of safety-critical systems.

Consider an infotainment system in an autonomous vehicle. This system, primarily used for navigation, entertainment, and vehicle settings, might be classified as QM (or level A) because its failure is unlikely to result in a direct safety hazard. However, quality management practices are essential to ensure that the system is reliable, user-friendly, and does not distract the driver or interfere with the operation of safety-critical systems[30].

Implementing ISO 26262

Implementation of ISO 26262 in infotainment systems involves:

- Hazard analysis and risk assessment to identify potential safety issues.
- Development of safety requirements to mitigate identified risks.
- Integration of safety mechanisms during the hardware and software design phases to ensure robust error handling and fault tolerance.
- Continuous validation and verification processes to confirm safety standards are met throughout the development cycle.

3 Integration of Automated Testing

The integration of automated testing into the IVI system development process represents a significant improvement in the pursuit of quality, reliability and efficiency. This chapter explores the key role of automated testing, with a particular focus on the screen life cycle and workflow within the GUI development process.

3.1 GUI Development Process in In-Vehicle Infotainment Systems

A specialised external entity within the VW Group developed the core software that powers the infotainment units. This entity is responsible for delivering updates and new versions of the software, commonly referred to as the ”model”, to all members of the VW Group at pre-determined intervals. Subsidiaries, such as Škoda Auto, take this basic software and customise it to reflect their unique brand identity and user experience expectations through a process known as ”skinning”. This customisation is critical to differentiating each brand within the VW Group and is meticulously carried out by Digiteq Automotive for Škoda Auto.

Skinners focus on the aesthetic and interactive aspects of the infotainment system's GUI. Their expertise lies in strategically placing interface elements such as icons, text, buttons, checkboxes and sliders to ensure an intuitive and responsive user experience. Unlike developers, who may focus on the underlying functionality of the system, skinners prioritise how these elements look and feel to the user, ensuring that every interaction is fluid and seamless. The culmination of their efforts results in a distinctive ”skin” for the infotainment system, representing the visual and interactive blueprint of the model[21].

3.2 Life Cycle of the Screen

This section introduces the concept of the GUI screen life cycle in the context of IVI systems. It emphasizes the critical phases that each screen undergoes from conception to completion. The life cycle is integral to ensuring that each graphical interface meets the rigorous usability and aesthetic standards required in today's automotive industry.

Overview

- The life cycle begins with the Initial Concept where the screen is conceptualized and basic functionality is outlined.
- Following this, the screen enters various Development and Testing Phases, which include skinning, iterative reviews, and multiple testing rounds to refine functionality and aesthetics.
- The final phases involve Validation and Deployment, where the screen is rigorously tested in real-world scenarios and prepared for integration into the final product.

This overview sets the stage for a detailed exploration of each stage in the text below, where we will visually map these processes using Figure 3.1 to provide a clearer understanding of the intricate processes involved.

Graphical Visualization of Screen Life Cycle in GUI Development
When developing graphical user interfaces for IVI systems, the screen life cycle is a critical process that ensures each screen is designed, tested and refined to meet stringent quality standards. This life cycle can be complex, involving several stages from initial concept to final testing. Figure 3.1 illustrates these stages in flowchart. Let us examine each stage to understand its role in the GUI development process.

New State
The life cycle of a screen begins in the "new" state, where the initial setup is created. This stage is crucial in laying the groundwork for what will become a fully functional GUI element. At this point, the screen contains only placeholders with no functional elements.

Open State

After completing the basic setup, the screen transitions to the ”open” state and is assigned to a skinner, as previously mentioned. Skinners are specialists who are known for their expertise in the aesthetic aspects of the GUI. They focus on customizing the screen by rigorously applying predefined design guidelines to enhance functionality and visual appeal. This includes integrating elements such as icons, buttons, and other interactive components that are essential for a user-friendly interface.

Waiting States

During the skinning process, the screen may enter various waiting states if additional customisation or resources are required:

- **Waiting for Model:** The screen is waiting for further model refinement.
- **Waiting for Resources:** Additional resources, such as high-resolution images or custom icons, are required.
- **Waiting for Supplier:** External suppliers may need to provide components or information.

These wait states highlight potential bottlenecks in the development process where delays may occur.

Skinning State

After receiving the necessary inputs, the screen returns to the skinning phase. This iterative approach allows for continuous refinement and ensures that every element on the screen is perfectly integrated and aligned with the overall design objectives.

Ready for Testing

Once the skinning is complete, the screen enters the ”ready for testing” phase. The screen is rigorously tested for functionality, usability and consistency with other GUI components. Any issues identified at this stage are critical as they can have a significant impact on the user experience.

Reopened

If errors are found, the screen status is changed to ”reopened” and sent back for further adjustments. This stage is crucial for quality control, ensuring that no faulty screens make it into the final product.

Tested

Screens that pass initial tests are labelled as ”tested” and progress to a second testing phase with updated software and HMI versions. Retesting frequently takes place in a real car prototype to verify their functionality in real-world conditions. This ensures that GUIs remain reliable following system updates.

Test Completed

Successfully tested screens are marked as ”test completed”, indicating that they are ready to be included in the final version of the infotainment system. This label signifies that the screen has met all required specifications and is expected to perform reliably in the field.

Cancelled

Occasionally, a screen may be cancelled due to various reasons such as changing project scopes or technical infeasibility. This status ensures resources are efficiently allocated by discontinuing work on elements that no longer meet the project’s needs.

This figure depicts the sequential stages each GUI screen undergoes from conception to completion, highlighting key processes such as skinning, testing, and final approval.

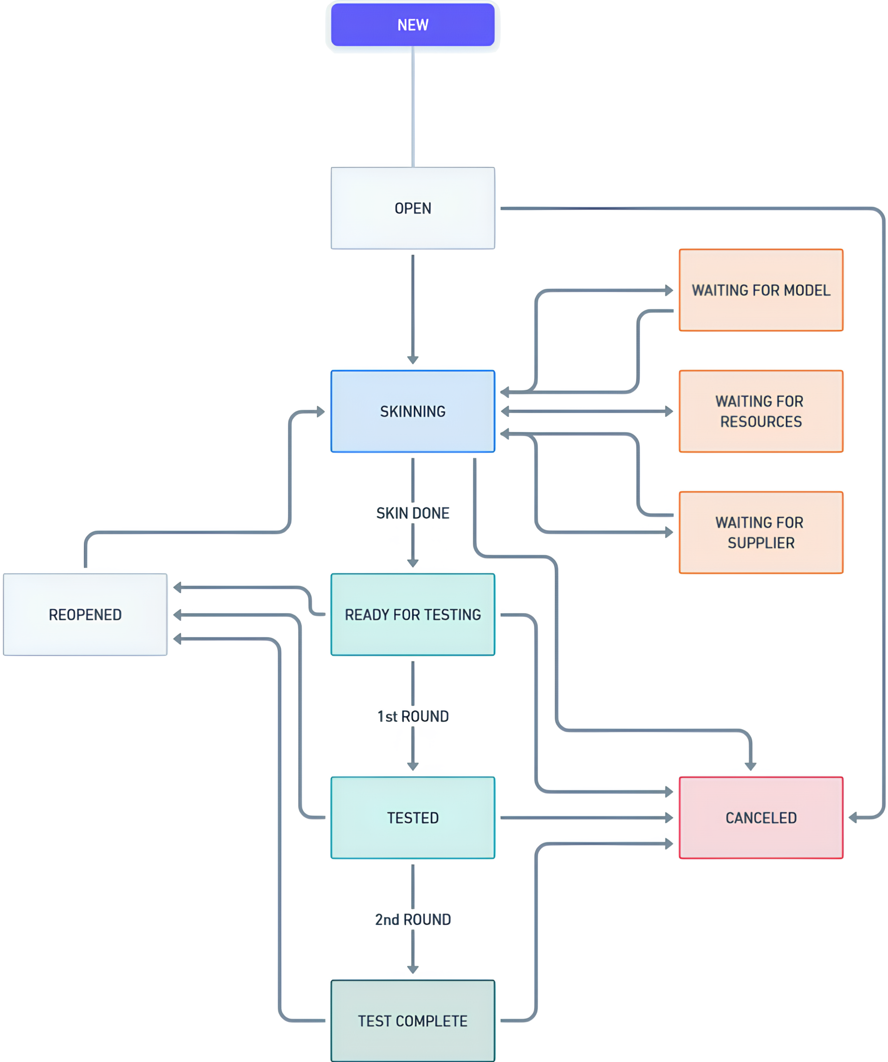

Figure 3.1: Life cycle of a GUI screen in infotainment system development

Understanding and managing the screen life cycle is crucial for developing effective and reliable IVI systems. Each stage of the process serves a specific purpose and contributes to the overall quality of the final product. This ensures that the system not only looks good but also functions seamlessly, enhancing the in-vehicle experience for users.

3.3 Motivation for Improving GUI Development Processes

The workflow for developing and validating GUI components is generally robust, taking each screen from inception to a fully tested state. However, this process faces its greatest challenges not during the initial development or testing phases but afterwards. Once a screen has completed its testing cycle, it transitions to a ”test complete” status, at which point skinners and testers typically complete their work on that particular component. However, the evolution of the GUI does not stop there; it requires ongoing updates to the software and HMI to meet evolving requirements and incorporate new features.

The process becomes particularly complex when updates are introduced by an external entity within a larger automotive group, such as the VW Group, which includes a variety of brands with their own specific requirements, such as Škoda Auto, Seat, Volkswagen and others. While these updates are intended to improve the system, they can inadvertently introduce discrepancies or bugs into screens that have previously been verified and marked as complete. Such issues can manifest themselves following updates to the HMI or software, potentially altering screen elements in unintended ways. Whether it is a noticeable discrepancy, such as missing text or icons, or more subtle issues, such as slight misalignments or shifts in graphical elements, detecting these issues post-update is a significant challenge.

Due to the complexity of the GUI and the numerous screens, each with its unique settings and configurations, it is impractical and time-consuming to manually retest each screen for potential issues after every update. This highlights the need for refining GUI development and testing methodologies.

The progress of software and HMI requires a more agile and responsive approach to GUI testing and development. It highlights the need for an integrated testing framework that can dynamically adapt to software updates, ensuring that previously completed screens remain error-free despite changes elsewhere in the system. The motivation for improving GUI development processes arises from the need to streamline update and testing cycles, reduce manual testing efforts, and enhance the overall quality and reliability of the GUI in infotainment systems[21].

3.4 Benefits of Integrating Automated Testing in GUI Development

The integration of automated testing into the GUI development process represents a significant change in the way infotainment systems are tested, optimised, and maintained. Unlike traditional manual testing methods, which typically involve a limited number of test rounds and sporadic user testing sessions, automated testing provides a consistent, systematic approach to identifying and resolving issues. This section explores the profound benefits of adopting automated testing in GUI development, highlighting the frequency of testing screens once a week as opposed to the two rounds of manual testing and occasional user testing iterations.

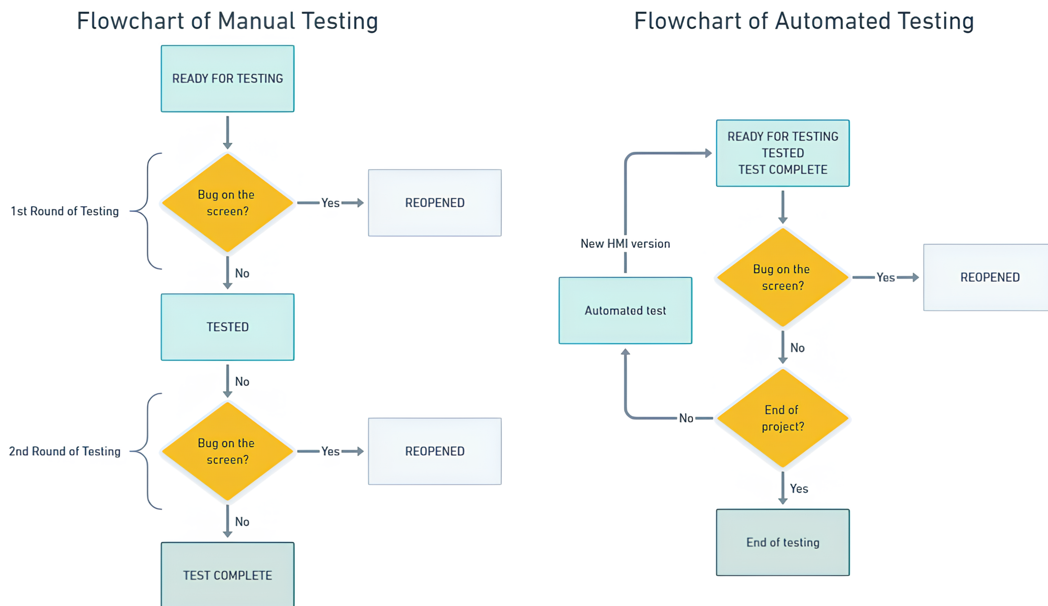

Figure 3.2: Flowcharts showing the different screen life cycle processes of manual and automated testing in the GUI development

Benefits of Automated Testing in Comparison with Manual One

In the dynamic field of IVI systems, automated testing is a method that stands out due to its ability to streamline development and ensure high-quality outputs. This method not only accelerates the testing process but also enhances the focus on innovation and user experience, aligning perfectly with the industry's goals for rapid development and robust software performance.

- **Consistent Quality Assurance:** Automated testing performs evaluations of GUI screens weekly, ensuring that any changes or updates are regularly scrutinised. This consistency leads to the early detection of bugs and issues that might otherwise go unnoticed until later stages of development or worse, after deployment.
- **Efficiency and Time Savings:** Manual testing is time-consuming and labour-intensive, often requiring significant human resources to conduct two rounds of testing in addition to user testing phases. Automated testing significantly reduces the manpower and time required, by performing assessments weekly, freeing up resources for other critical development tasks.
- **Comprehensive Coverage:** Automated testing can test every screen and interaction within the GUI every week, providing a level of coverage that is virtually impossible to achieve with manual testing. This thoroughness ensures that even the smallest inconsistencies or errors are identified and fixed.
- **Objective Results:** Automated testing is not subject to human error or bias, providing objective, consistent results week after week. This objectivity is essential for maintaining high standards of quality and functionality in GUI development.

4 Foundations of Automotive Testing

Developing and testing IVI systems requires the use of appropriate tools and protocols to ensure product reliability and functionality. This chapter examines two essential components of automotive testing: the CAN bus and the Tcl programming language.

The Role of CAN Bus

Infotainment testing requires the use of the CAN bus to simulate real-world scenarios where multiple vehicle subsystems communicate simultaneously. The CAN bus is used to simulate various CAN messages that are vital for operational tests, such as displaying infotainment content or managing vehicle states. For example, the signal "Klemme 15" is critical for indicating the ignition status of the vehicle. If the Klemme states (for example, Klemme 15 or Klemme 30) indicate that the engine is off, some infotainment screens may not be visible or operational. This can affect the user experience and functionality tests. To stay ahead of the curve and gain a deeper understanding, it is beneficial to refer to section 4.1 Introduction to CAN Bus, which covers the foundational aspects of CAN bus technology.

The Importance of Tcl Programming

We use Tcl as our primary scripting language for creating TestCases and functions within our testing framework, alongside the CAN bus. Tcl was chosen for its simplicity and effectiveness in handling string-based operations and automated task execution, which are essential in test scenario creation. We use Tcl to create automated tests that evaluate all aspects of the infotainment system's UI and backend. This ensures that all components work seamlessly together.

4.1 Introduction to CAN Bus

CAN bus is a robust vehicle bus standard designed to facilitate communication between various vehicle systems without the need for a central computer. This revolutionary technology, developed by Bosch in the 1980s, has become a fundamental component of modern automotive design, enabling various electronic control units (ECUs) within a vehicle to communicate efficiently. Unlike traditional wiring systems, the CAN bus allows the reduction of complex wiring harnesses, resulting in improved vehicle reliability, easier repair processes, and enhanced functionality.

The introduction of the CAN bus was primarily driven by the need for a more efficient and reliable way to distribute control functions and diagnostic information between vehicle systems. As vehicles became more complex, incorporating more electronic features such as advanced engine controls, IVI systems, and various sensors, the limitations of conventional point-to-point wiring became apparent. The CAN bus emerged as a solution to these challenges, not only reducing the complexity and cost of wiring but also facilitating real-time data exchange between ECUs[35].

4.1.1 CAN Bus Architecture

This section explains the detailed architecture of the CAN bus system. It describes the essential components and configurations that enable robust communication capabilities within automotive networks.

Physical Layer

The physical layer of the CAN bus system is critical in defining the electrical characteristics and physical connections that enable reliable data transmission over the network. The specifications of this layer ensure that the CAN bus can operate effectively in the demanding conditions common to automotive and industrial environments[4].

Transmission Medium

The choice of transmission medium is vital for the performance of the CAN Bus system. Generally, a twisted pair cable is used for its ability to reduce electromagnetic interference, which is a common issue in electrically noisy environments found primarily in vehicles and industrial machinery. The twisted pair design helps in canceling out noise that might be induced onto the wires, ensuring that the integrity of the transmitted signals is maintained[4].

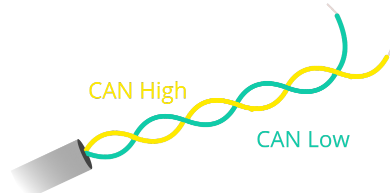

Figure 4.1: CAN bus twisted pair cable[13]

Signal Levels

The CAN bus uses differential signalling with two wires, CAN high and CAN low, to transmit information. This method is particularly effective in automotive environments where electrical noise and interference are prevalent. Differential signalling allows the CAN bus to achieve a high level of noise immunity by cancelling out electrical noise common to both wires. As a result, the integrity of the signal is maintained even in the harsh electrical environment found in vehicles.

The relationship between the CAN high and CAN low lines is quantified by the differential voltage (V_{diff}), defined by Equation 4.1:

$$V_{diff} = V_{CAN_H} - V_{CAN_L} \tag{4.1}$$

This formula is of paramount importance, as it determines the logical state transmitted across the network. A higher differential voltage (approximately 2 volts) indicates a dominant state (logical '0'), which is a critical state for asserting control in the network. Conversely, a lower or negligible differential voltage (close to 0 volts) corresponds to a recessive state (logical '1'), which signifies that no node is actively trying to dominate the bus.

The voltage differential approach reduces the risk of signal level misinterpretation by receiving nodes, ensuring accurate and robust data transmission. Additionally, this method helps to decrease electromagnetic emissions from the cable, contributing to the overall electromagnetic compatibility of the system.

The CAN bus is widely adopted in automotive and industrial networks due to its robustness to external interference and its ability to maintain signal integrity

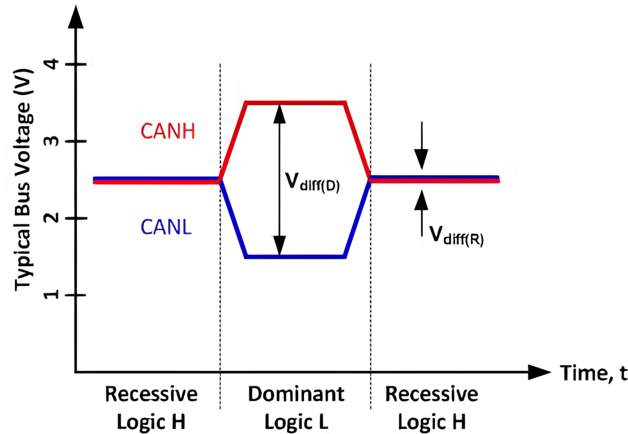

Figure 4.2: Signal levels in a CAN Bus[15]

under challenging conditions. The design considerations for signal levels, as shown in the equation and figure, highlight the sophisticated engineering behind the CAN protocol that enables reliable communication in environments subjected to significant electrical noise[4, 5].

Connectors and Termination Resistors

The connectors used in the CAN Bus system are standardized to ensure compatibility and reliability across different devices and manufacturers. It is essential to use high-quality and robust connectors to maintain secure physical connections, which are critical for the uninterrupted operation of the network.

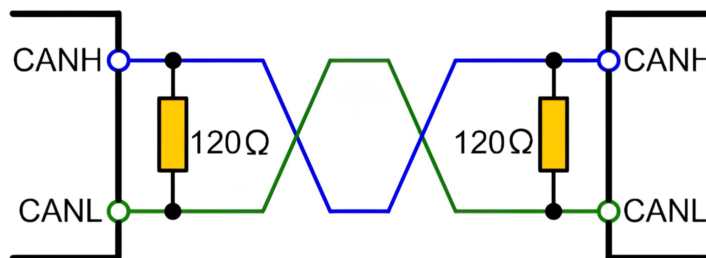

Figure 4.3: CAN bus termination resistors[10]

Termination resistors play a crucial role in the physical layer by preventing signal reflections at the ends of the transmission medium. A 120-ohm resistor is usually connected at each end of the CAN network to match the characteristic impedance of the twisted pair cable. This ensures that signals do not reflect along the cable, which could cause interference and degrade communication quality[4].

Physical Layer Standards

The ISO 11898 standard, which governs the CAN Bus, specifies two different physical layer options: high-speed CAN (ISO 11898-2) and low-speed, fault-tolerant CAN (ISO 11898-3). The high-speed variant is designed for systems where rapid data transmission is essential, supporting speeds up to 1 Mbps. In contrast, the low-speed, fault-tolerant version is optimized for applications where communication integrity and fault tolerance are prioritized over transmission speed[4].

Importance of the Physical Layer

The design and implementation of the physical layer are foundational to the overall performance and reliability of the CAN Bus system. By specifying the electrical and physical parameters for data transmission, this layer ensures that the network can operate effectively under various conditions, providing a reliable communication backbone for complex electronic systems[4].

4.1.2 CAN Bus Protocols

This section explores the details of CAN message frames, including their structure, types, and the difference between Standard CAN and Extended CAN frames.

The Structure of a CAN Message

A CAN message frame can be divided into several fields, each of which serves a specific purpose in the data communication process. The CAN frame structure comprises several fields: the Start of Frame, the ID Field, the Remote Transmission Request Field, the Control Field, the Data Field, the CRC field, the Acknowledge field and the End of Frame. The Arbitration Field, which includes the identifier and RTR, is crucial for determining the priority of messages on the bus.

- **SOF:** The Start of Frame is a ”dominant 0” to tell the other nodes that a CAN node intends to talk

- **ID:** The ID is the frame identifier - lower values have higher priority

- **RTR:** The Remote Transmission Request indicates whether a node sends data or requests dedicated data from another node

- **Control:** The Control contains the Identifier Extension Bit which is a ”dominant 0” for 11-bit. It also contains the 4 bit Data Length Code that specifies the length of the data bytes to be transmitted (0 to 8 bytes)

- **Data:** The Data contains the data bytes aka payload, which includes CAN signals that can be extracted and decoded for information

- **CRC:** The Cyclic Redundancy Check is used to ensure data integrity

- **ACK:** The ACK slot indicates if the node has acknowledged and received the data correctly

- **EOF:** The EOF marks the end of the CAN frame[13, 5]

Standard CAN vs. Extended CAN

The CAN protocol, central to automotive and industrial communications, operates under two different specifications: Standard CAN (2.0A) and Extended CAN (2.0B). These specifications differ primarily in the length of their identifiers, which has a direct impact on message prioritisation and network capacity.

Standard CAN uses an 11-bit identifier, providing up to 2,048 different message IDs. This version is widely used in applications where network complexity is moderate and the number of devices is relatively small[5].

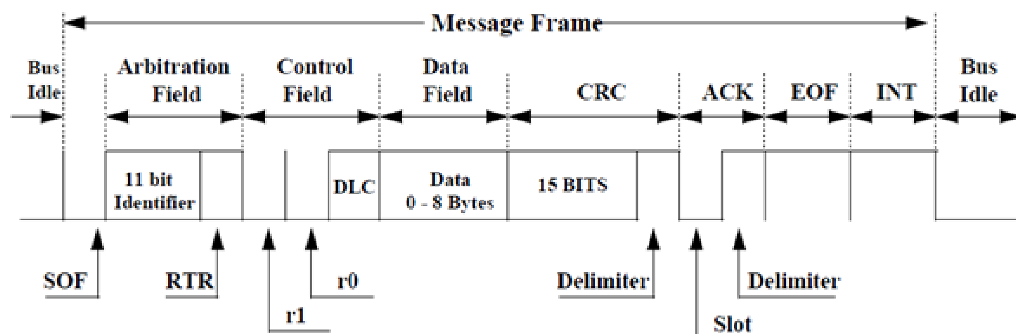

Figure 4.4: The figure shows the structure of a CAN 2.0A (Standard Format) message frame, detailing the fields involved in data transmission[20]

Extended CAN extends the identifier to 29 bits, significantly increasing the address space to over 537 million possible message IDs. This extension is designed for more complex systems with a higher number of nodes, ensuring that each message can be uniquely identified and prioritised appropriately. Despite the larger identifier size, Extended CAN maintains compatibility with Standard CAN through careful design, allowing both message types to coexist on the same network[5].

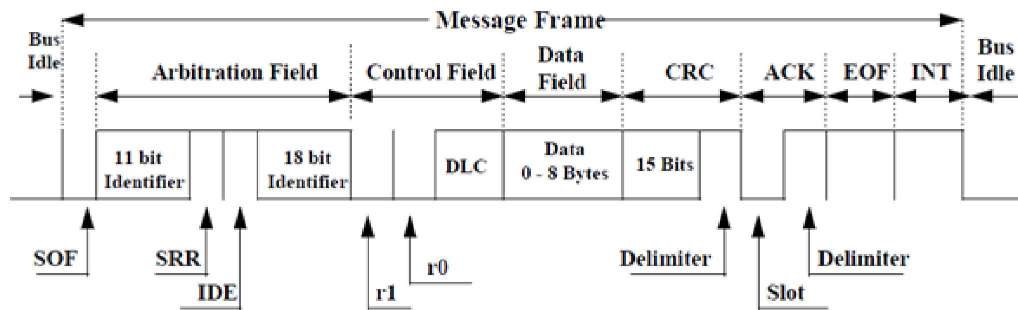

Figure 4.5: The figure shows the structure of a CAN 2.0B (Extended Format) message frame, detailing the fields involved in data transmission[20]

CAN Identifier and Arbitration Process

The CAN uses a unique method of message prioritisation and collision resolution through its identifier and arbitration process. This system is fundamental to CAN's efficiency and reliability, particularly in environments where timely data delivery is crucial, such as automotive and industrial applications[4].

- **CAN Identifier (ID):** Each message transmitted over a CAN network is assigned a unique identifier. This identifier is crucial for identifying the message's content and determining its priority on the bus. The identifier is a standard 11 bits long. The lower the numerical value of the ID, the higher the priority of the message.

- **Priority-Based Messaging:** The prioritization mechanism in CAN networks is based on the binary value of the message ID. Messages with lower ID values are given higher priority and are therefore transmitted first. This system ensures that critical messages, such as those related to vehicle safety systems, are given precedence over less urgent communications.

- **Arbitration Process:** The arbitration process is a non-destructive method used by CAN to manage message collisions. When multiple devices begin transmitting messages simultaneously, the message with the lower ID value,

and therefore higher priority, is granted bus access while the others cease transmission. This process ensures that data is transmitted without any corruption or loss.

- **Role of Recessive and Dominant States:** In CAN communication, the dominant state (logical 0) always overwrites the recessive state (logical 1). This physical layer property is pivotal during arbitration, as it allows nodes to passively withdraw when a higher priority message is detected without affecting the ongoing transmission.

- **Collision Resolution:** The arbitration mechanism efficiently resolves collisions without the need for retransmission mechanisms typical in other networking protocols. This significantly reduces latency and ensures real-time communication, which is critical in control and monitoring applications[5, 13].

Enhancing CAN Protocol Reliability with Cyclic Redundancy Check

The Cyclic Redundancy Check (CRC) is an essential feature of the CAN protocol and serves as a powerful tool for detecting errors in transmitted messages. Within a CAN frame, the CRC field follows the data field and consists of a predetermined number of bits that are used to calculate a checksum of the frame contents before transmission. This checksum is recalculated by the receiver to verify the integrity of the received message. If the calculated CRC matches the received CRC, the message is considered error-free; otherwise, it indicates that the message was corrupted during transmission[5].

Data Transmission and Reception

In a CAN network, each node is capable of reading any message sent by any other node. The key to this capability lies in the CAN bus architecture, which is based on a broadcast communication mechanism. Here's how it works:

1. **Message Broadcasting:** When a node sends a message, it transmits it to all nodes on the network at the same time. This is done using a differential signaling method over the two-wire bus, which improves signal reliability even in noisy environments.

2. **Message Detection and Filtering:** Each node on the network has filters that analyze incoming messages to determine their relevance. The filters use the message identifier, which indicates its priority and type.

3. **Receiving Messages:** When a message is received, it is accepted and processed only if its identifier matches the criteria set by the node's filters. This ensures that only relevant messages are processed, optimizing network efficiency and reducing processing overhead.

4. **Error Handling:** The CAN protocol has error detection and handling mechanisms. If a node detects an error in a message, it flags the error to all other nodes. The erroneous message is then discarded, and it may be retransmitted depending on the system configuration to ensure system reliability[5].

For further information on detecting and managing errors in the CAN bus network, please see the subsequent section on 'Error Signalling' below, which elaborates on the mechanisms and protocols involved in maintaining data integrity across the network.

Error Signalling

Error signalling in the CAN protocol is a sophisticated mechanism designed to ensure the integrity and reliability of data communications across the network. Within the CAN framework, any participating node can detect errors and immediately signal them to all other nodes on the network. This is achieved by the transmission of an Error Frame, which consists of two different fields: the Error Flag and the Error Delimiter.

The Error Flag is a sequence of six consecutive dominant or recessive bits, depending on the type of error detected, which interrupts the normal flow of data on the bus and alerts all nodes to the presence of an error. Following the Error Flag, the Error Delimiter, consisting of eight recessive bits, marks the end of the Error Frame and the resumption of normal bus activity. When an error is detected, nodes will automatically attempt to retransmit the erroneous message, ensuring that no critical information is lost[5].

4.2 Basics of CAN Bus in Automotive Testing

The CAN bus is indispensable in the field of automotive testing, playing a crucial role in the diagnosis, evaluation and validation of electronic systems in vehicles. Its robustness and communication efficiency enable a wide range of test applications critical to ensuring vehicle reliability and performance.

4.2.1 Comprehensive Testing Methods for CAN Bus Systems

In order to guarantee the resilience and dependability of CAN bus systems in automotive applications, a series of comprehensive testing methods are employed. These methods, which encompass diagnostic testing, performance analysis, and regression testing, are fundamental for assessing system integrity, performance, and compatibility with new updates. Each plays a crucial role in identifying potential issues and optimising system operations to meet the rigorous automotive standards.

Diagnostic Testing

The ability of the CAN bus to provide access to real-time data from ECUs simplifies diagnostic testing. Testers leverage this capability to execute diagnostic commands directly to ECUs, retrieving fault codes and monitoring the operational status of various subsystems. This real-time data acquisition allows for the immediate identification and analysis of simulated faults or conditions, facilitating rapid troubleshooting and rectification processes[25].

Performance Analysis

The analysis of CAN traffic provides valuable insights into the performance and interaction of ECUs across different vehicle systems. Testers can observe how messages exchanged between ECUs affect the behavior and efficiency of vehicle operations under various conditions, such as acceleration, braking, and environmental changes. This analysis identifies bottlenecks and inefficiencies in data transmission, as well as potential areas for optimization in software algorithms and system integration. The aim is to improve overall vehicle performance by ensuring that systems such as adaptive cruise control or lane departure warning work together seamlessly[25].

Regression Testing

As vehicle software continues to evolve for improved functionality and safety, regression testing becomes essential. By monitoring CAN communications, testers can ensure that updates or newly introduced ECUs integrate seamlessly with existing systems without causing regression or compatibility issues. This rigorous validation process is critical to maintaining the integrity of the vehicle's electronic systems, ensuring that updates enhance functionality without compromising the vehicle's established operating standards[25].

4.2.2 Integration of On-Board Diagnostics in CAN Bus Systems

On-Board Diagnostics (OBD) provides a standardised method for vehicle systems to conduct self-diagnostics and report issues. OBD's functionality has expanded beyond its initial purpose of monitoring vehicle emissions, now offering comprehensive diagnostic capabilities that support maintenance and troubleshooting.

The introduction of OBD-II in the mid-1990s was a significant advancement as it standardised all vehicles sold in the United States. This standardsation enables diagnostic tools to interface with any vehicle, regardless of the manufacturer, through a universal data link connector. The integration of OBD systems with the CAN bus allows for real-time monitoring and diagnostics of various vehicle subsystems, enhancing the efficiency and accuracy of identifying malfunctions.

When the OBD system detects a fault, it logs a DTC and may activate the "Check Engine" light to alert the driver to potential issues. These DTCs can be accessed through OBD-II scanners, providing technicians with specific insights into the vehicle's operational status[23].

In addition to diagnostics,OBD systems play a vital role in communicating with and controlling vehicle units. The OBD interface allows for flashing of the units, effectively updating or modifying their software. This functionality is crucial as it enables on-demand changes to vehicle configurations. Flexibility is crucial for testing infotainment screens under different configurations, including the activation of features such as lane assist or adaptive cruise control.

4.2.3 Types of CAN Buses in Vehicles

In the field of automotive design, the Controller Area Network (CAN) bus represents a fundamental component for inter-device communication within the vehicle. Various types of CAN buses have been developed to cater to specific functionalities, with the objective of enhancing vehicle performance, safety, and entertainment features. The major types of CAN buses typically include:

- **Powertrain CAN (Antrieb CAN):** This bus handles critical functions related to engine management, transmission control, and other powertrain components. Its high-speed operation (up to 1 Mbps) ensures rapid response times necessary for these vital systems.
- **Vehicle Speed CAN (Fahrwerk CAN):** Specifically focused on vehicle dynamics, this bus manages systems such as braking, steering, and suspension.

It is crucial for safety and stability control technologies that require real-time execution.

- **Comfort CAN:** This type manages systems not critical to the vehicle's immediate operational safety or performance, such as air conditioning, seat adjustment, and lighting. Typically running up to 125 kbps.
- **Infotainment CAN:** Dedicated to the IVI systems, this bus connects components that provide entertainment and information services, such as audio and video playback, navigation systems, and connectivity modules. It ensures that data flow for entertainment systems does not interfere with the critical control buses.

The CAN buses converge at a central unit, designated as the Gateway. This configuration permits the efficient management of data and a reduction in the network load, as each bus handles only the relevant subsystems. By dividing the buses, manufacturers can isolate systems in order to prevent failures in one network from affecting others. This enhances vehicle safety and performance. Furthermore, this separation allows for simpler troubleshooting and maintenance, as issues can be localized to specific areas of the vehicle's network, thereby facilitating diagnostics[49]. An illustrative example of types of CAN buses can be found in subsection 5.2.9.

4.3 Introduction to Tcl Programming

As we begin to explore automated testing for IVI systems, it is important to have the necessary tools and knowledge. Tool command language (TCL) is a programming language that stands out for its simplicity, flexibility, and suitability for scripting automated tests. This chapter, "Introduction to Tcl Programming," is dedicated to explaining the basics of Tcl, starting with an overview of the language. This section explores the origins, core principles, and unique features of Tcl that make it an invaluable asset in automated testing. The aim is to build a strong foundation for subsequent sections, which delve deeper into Tcl programming syntax, data structures, and practical applications.

4.3.1 Overview of Tcl Language

Tcl, also known as **Tool command language**, is a dynamic scripting language that is highly valued for its simplicity, extensibility, and wide applicability across various domains. Its design concept emphasises ease of use and straightforward

syntax, making it an accessible tool for both novice programmers and seasoned professionals. This overview explores the key attributes, design principles, and versatility of Tcl that make it an essential tool in software development and testing[7].

Origins and Evolution

Tcl was developed in the late 1980s by John Ousterhout as an embeddable command language for applications. Its utility quickly expanded to encompass script automation, rapid prototyping, and even full-scale application development, thanks to its companion graphical toolkit, **Tk**. Over the years, Tcl has evolved to include advanced features such as network support, enhanced performance mechanisms, and tools for interfacing with various programming languages and environments[7].

Philosophy and Design Goals

Tcl's design is based on the principle that code should be easy to write, adapt and maintain. It achieves this through:

- **Simplicity:** Tcl's syntax is intentionally minimalist, avoiding the complexities that are often present in other scripting languages.
- **Flexibility:** Tcl scripts can run on multiple platforms without modification, demonstrating true write-once, run-anywhere capabilities.
- **Extensibility:** The language may be expanded with extra commands and libraries that are customized to meet specific project requirements or domains.

Key Features

- **Dynamic Typing:** Variables in Tcl are not bound to any specific data type, which enhances the language's flexibility and simplifies script development.
- **Command-Based Structure:** Tcl considers nearly all operations as commands, including assigning variables and calling procedures, contributing to its consistent syntax.
- **Powerful String Processing:** Due to its string-centric design, Tcl provides a wide range of tools for string manipulation, making it highly proficient in handling text-heavy tasks.
- **Comprehensive Standard Library:** Tcl's standard library is extensive, enabling a wide range of scripting activities.

The Tcl/Tk Combination

Tk is the standard GUI toolkit for Tcl, enabling developers to design and implement graphical user interfaces with minimal effort. The combination of Tcl and Tk enables the development of user-friendly interfaces for test scripts. However, this is beyond the scope of this paper.

4.3.2 Advantages of Using Tcl for Infotainment System Testing

Tcl is particularly advantageous for scripting automated tests of automotive infotainment systems due to its flexibility, ease of use, and robust integration capabilities. The following paragraphs will outline the key benefits Tcl brings to infotainment testing.

1. **Ease of Scripting Interactive Tests:** Tcl's straightforward syntax and dynamic nature render it an optimal choice for scripting complex interactive tests that are essential for infotainment systems. These systems, which feature multimedia playback, navigation, and connectivity functions, benefit from Tcl's capacity to rapidly script and modify tests for these interactive features.

2. **Integration with Infotainment Testing Tools:** Tcl is designed to integrate seamlessly with tools such as CANoe, which is widely used for simulation and testing in automotive infotainment development. This integration allows for direct control and testing of infotainment protocols and hardware interfaces through Tcl scripts, enhancing the thoroughness and efficiency of tests.

3. **Rapid Development and Iteration:** The capacity to rapidly develop and modify test cases is of paramount importance in the context of the rapid evolution of infotainment technology. Tcl's interpretative execution enables the implementation of changes without the need for lengthy recompilations, thus facilitating the acceleration of iteration and adaptation to new testing scenarios as infotainment systems evolve.

4. **Cross-Platform Functionality:** The cross-platform nature of Tcl ensures that testing scripts can be developed and executed across various development environments without modification, thus facilitating the development of robust and reliable software. This is particularly advantageous in the context of infotainment systems, where software may be developed in disparate operating system environments.

The aforementioned advantages render Tcl a preferred scripting language for automating the testing of infotainment systems in vehicles. This ensures that these

complex systems are thoroughly tested for functionality, reliability, and user experience.

4.3.3 Programming in Tcl

As mentioned above, Tcl offers a versatile scripting platform for automotive testing, particularly beneficial in the realm of infotainment systems. The simplicity and comprehensive functionality of Tcl enable testers to create flexible and dynamic test scripts in a time-efficient manner. For those engaged in automotive testing, Tcl offers an accessible yet potent toolset that is well-suited to the complexities of modern infotainment systems.

For further detailed information on Tcl programming constructs such as syntax, loops, functions, dictionaries, namespaces, lists, and arrays, please refer to the section A.1. The appendices provide comprehensive guidance and illustrative examples that demonstrate the effective utilisation of these Tcl features in the context of automotive infotainment testing. The objective of these resources is to assist both novice and experienced programmers in fully utilising the potential of Tcl in their testing frameworks.

5 Test Bench

This chapter is devoted to the intricacies of the test bench, a key component of automotive testing, and provides a comprehensive overview of the equipment, methods and environments used to test IVI systems.

The test bench is at the heart of automotive testing, serving as a melting pot where hardware and software components are meticulously evaluated under simulated real-world conditions. This chapter is divided into four distinct sections, each dedicated to unveiling a facet of the test bench, painting a holistic picture of its importance, functionality and impact on the development of IVI systems.

Figure 5.1: Test bench for automated testing

5.1 Overview of the Test Bench

The test bench is the backbone of today's approach to validating and ensuring the optimal performance of IVI systems. Its creation and meticulous design is based on the need to bridge the gap between theoretical development and real-world application. This section aims to explain the overarching framework, the functionalities and the essential role of the test bench in the life cycle of IVI systems.

The test bench is a combination of advanced hardware and software components designed to simulate a comprehensive automotive environment. The simulation is a rigorous testing ground that subjects the infotainment systems to a range of scenarios, from the mundane to the extreme. It is not a mere facsimile of real-world conditions. The test bench is designed to replicate the complexities of automotive operation, including variations in environmental conditions, user interactions, and vehicle dynamics. It provides a reliable environment for testing.

The test bench concept is both iterative and comprehensive, ensuring that every component of the infotainment system is scrutinised under a wide range of conditions. This process employs a dual approach, utilizing both manual oversight and automated testing protocols. The automated aspect is particularly noteworthy, utilizing scripts and scenarios designed to mimic a wide range of user interactions and system responses. This automation is crucial not only for its efficiency but also for its ability to consistently replicate test conditions, ensuring the reliability of the testing outcomes.

Furthermore, the test bench is important in enabling the creation of scalable and adaptable testing scenarios. This flexibility is crucial due to the rapid evolution of infotainment technologies and their expanding functionalities. The test bench, therefore, is not static but evolves in concert with the systems it tests, incorporating new testing methodologies, hardware upgrades, and software updates to address and anticipate the needs of emerging technologies.

5.2 Hardware Components

The test bench is a crucial element in the ecosystem of IVI testing. It is supported by a diverse array of hardware components, each playing a specific role in creating a comprehensive environment for testing infotainment units under conditions that closely simulate real-world use. This section explores the hardware components that make up the test bench, highlighting their functionalities and integration into the testing process. For a visual representation of the interconnections between these components, please refer to Figure 5.15. Additionally, Figure 5.14 illustrates the configuration of power supply connections, crucial for understanding the energy distribution within the test bench.

5.2.1 Windows PC

The Windows PC acts as the central control unit of the test Bench, orchestrating the execution of test scenarios, data collection, and analysis. It hosts the software tools and scripts necessary for automated testing, serving as the interface through which testers can interact with the test bench.

Figure 5.2: High-performance Windows PC[17]

The Windows PC, that is chosen for the test bench, is typically a high-performance computer with a multi-core processor and ample RAM to handle the demanding computational tasks associated with automated testing. The choice of a multi-core processor is dictated by the need to run multiple software tools and scripts simultaneously without performance degradation. Similarly, sufficient memory ensures smooth operation when processing large data sets, running complex simulations or managing multiple applications.

The operating system of choice is a version of Windows known for its stability, security and broad support for automotive test software and hardware interfaces. This ensures compatibility with a wide range of tools and facilitates easy integration with other test bench components such as the grabber and CAN case.

The primary function of the Windows PC within the test bench is to act as a command centre from which tests are initiated, monitored and analysed. It hosts the test automation software responsible for initiating test sequences, simulating user input and collecting responses from the infotainment system.

Moreover, the Windows PC often includes tools for remote access and control, allowing test engineers to monitor and adjust test processes remotely. This capability is particularly valuable in extended test scenarios or when the test bench is used across different projects or locations.

5.2.2 Grabber

Digiteq Automotive's Modular FrameGrabber (MGB) is the European automotive industry's in-car video stream capture device. The MGB specialises in capturing video streams directly from the video interface of infotainment central units and transferring them via TCP stream over Gigabit Ethernet for analysis. It supports multiple video formats, including H.264 and Motion JPEG, and can capture screenshots in PNG format. It also allows targeted streaming through the Area of Interest feature, which increases frame rates and reduces baud rates for specific video segments.

Figure 5.3: Modular FrameGrabber (MGB)[29]

5.2.3 CAN Case

The CAN Case by Vector is essential for automotive testing, especially when used in conjunction with software such as CANoe. It excels at simulating CAN messages, which is essential for testing the communication and functionality of IVI systems and other networked components within a vehicle. The ability to simulate CAN messages allows developers and testers to meticulously evaluate system response under different scenarios, ensuring robustness and reliability before deployment.

Figure 5.4: Vector CAN Case[14]

5.2.4 Manson Power Supply with Remote Control

The Manson remote power supply is essential for testing, supplying power to vehicle units and allowing them to be remotely restarted. This feature is critical in providing the benefit of remote unit management. It also connects to a Windows PC via USB, allowing software-based automation and control of the power supply's behaviour, improving test efficiency and realism.

Figure 5.5: Power supply of Manson company[11]

5.2.5 12V Power Supply

The 12V Power Supply is a component of the test bench. Its purpose is to power devices such as the Grabber, Quido, or LED Bar signalization. Unlike remote-controlled power supply units, this 12V power source provides a steady and reliable supply of electricity to support the operation of these critical testing components. This ensures they function optimally throughout the testing process. Its role is crucial in maintaining the continuity and efficiency of the testing environment by providing a stable power foundation for the various devices that are integral to automotive testing.

Figure 5.6: 12V power supply[44]

5.2.6 PCAN-PCI Express

The PCAN-PCI Express is an interface card that enables a connection between PCI Express slots in computers and CAN networks. This hardware is crucial for automotive testing and development, as it allows direct access and communication with CAN systems within a vehicle from a desktop or workstation. It is especially beneficial in settings that require reliable data exchange and network diagnostics, guaranteeing smooth integration and communication with the vehicle's CAN network for diagnostic, programming, and testing purposes.

5.2.7 LED Bar Signalisation

The test bench setup includes the LED Bar Signalisation, which uses a simple yet effective LED strip connected through four wires: Ground, Red(**R**), Green(**G**), and Blue(**B**). The system employs relays for the Red, Green, and Blue cables,

allowing for dynamic color alterations. The apparatus primarily uses a binary color signaling system. Green indicates that the test bench is available for use, signaling availability to those unfamiliar with testing routines. In contrast, red alerts indicate that the test bench is currently in use, and others should avoid interacting with the system during testing processes.

5.2.8 Two Phones (One Android and One with iOS)

Integrating an Android and an iOS smartphone into the test bench significantly enhances the testing capabilities for IVI systems. This setup is essential for evaluating different phone functions, including compatibility and functionality with SmartLink features such as Android Auto and Apple CarPlay. By using features from the two main mobile operating systems, testers can ensure that the infotainment system offers a smooth and user-friendly experience for connecting and entertaining with a wide range of devices.

5.2.9 Front Panel with All CAN Buses

Our test bench features a fully modular front panel design, distinguished by its use of standardised D-Sub connectors. This modular approach contrasts sharply with typical in-car systems that require direct unit connections, offering greater flexibility for our testing setup.

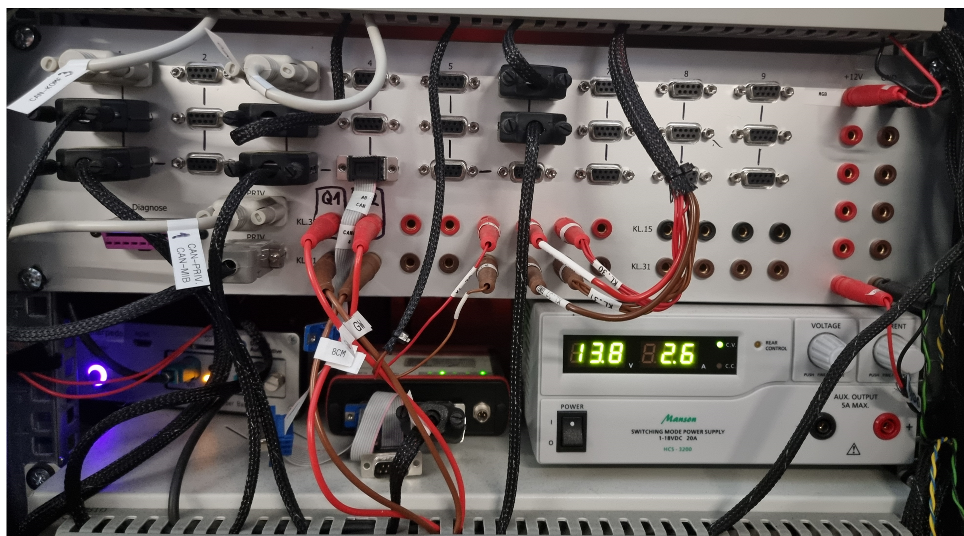

Figure 5.7: Example of the front panel implemented in a test bench

The front panel acts as the central hub for all vehicle unit connections, accommodating various CAN buses like Comfort, Infotainment, and Fahrwerk. For detailed descriptions of the specific functionalities and components associated with these CAN buses, see the subsection 4.2.3. This configuration facilitates seamless plug-and-play connectivity, allowing for comprehensive testing of multiple systems either concurrently or individually. Key advantages of the test bench are:

- **Ease of Rebuilding:** The modular nature of the front panel facilitates the reconfiguration of the test bench for new projects or different vehicles, thereby ensuring its long-term utility and adaptability.
- **Simplified Troubleshooting:** Centralized connections enhance accessibility, easing maintenance and troubleshooting efforts.
- **Scalability:** Easily accommodates additional modules, facilitating updates with minimal adjustments.
- **Reliability:** Standardized connectors minimize wiring errors and reduce hardware risks, enhancing overall test accuracy.

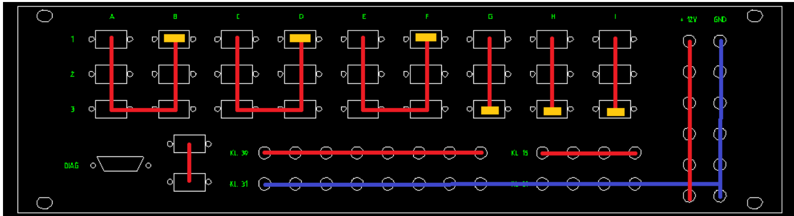

Figure 5.8: Front panel connection design

In summary, the front panel represents a significant enhancement to our testing procedures, streamlining the setup processes and enabling efficient and flexible testing across a range of vehicle systems.

5.2.10 Quido by Papouch

Papouch's Quido, while not exclusively designed for automotive applications, has been effectively adapted for use in automotive test environments within our test bench configuration. As a versatile relay board, Quido allows precise control of the power supply to various vehicle units such as the Gateway and BCM. This control is critical as it allows power to be selectively switched off or on as required during test sequences.

In addition to managing the power supplies, Quido also facilitates the control of LED signalling, which is used to indicate the status of the test bench visually. Its functionality also extends to enabling remote restart of units such as the infotainment system, improving test efficiency by reducing the need for direct manual intervention. Quido is also used to manage the ventilation system within the test bay, ensuring that equipment is maintained at optimum operating temperatures.

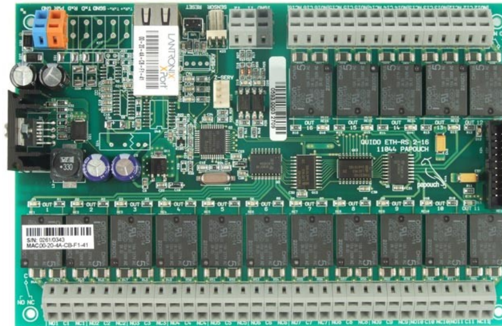

Figure 5.9: Quido ETH 2/16: 2 inputs, 16 outputs and thermometer[24]

5.2.11 UPS

The Uninterruptible Power Supply is a crucial element of the test bench, providing backup power during short-term outages. It guarantees that the entire test bench, including vital testing processes and equipment, remains operational for 6-9 minutes without external power. This capability is essential for maintaining testing continuity and safeguarding against data loss or equipment malfunction due to sudden power interruptions.

Figure 5.10: Uninterruptible power supply (UPS)[36]

5.2.12 Vehicle Units

Infotainment Unit

The Infotainment Unit is the central hub of the in-vehicle digital experience, combining entertainment and information for both the driver and passengers. It is a multifaceted system that includes navigation, media playback, and connectivity features such as smartphone integration and internet access. This unit not only enhances the driving experience through multimedia and navigation, but also plays a crucial role in vehicle safety and diagnostics by displaying vital information and alerts. The integration and testing of the system within the vehicle ecosystem are crucial for ensuring functionality, user interface usability, and overall customer satisfaction.

Figure 5.11: Škoda Enyaq infotainment unit[3]

Infotainment Display (ABT)

The ABT (Anzeigebedienteil), which stands for Display Control Unit, is a central component of the vehicle's infotainment system. It enables users to interact with various infotainment features through touch, providing control over navigation, media, connectivity, and more. This touch interface improves the user experience by providing an intuitive way to access the infotainment systems' extensive functionalities.

Figure 5.12: Škoda Enyaq 13" ABT[2]

Gateway Unit

The Gateway unit in a vehicle serves as a crucial network hub, enabling communication and data exchange between different ECUs across various vehicular subsystems. It guarantees the smooth operation of the vehicle's complex networks, such as CAN, LIN, and others. The Gateway unit is a crucial component in modern automotive architecture as it manages the flow of information, ensuring the vehicle's overall performance, safety systems, and infotainment functionality. Its importance cannot be overstated.

Figure 5.13: Škoda Enyaq Gateway unit[12]

5.3 Software Components

The section on Software Components discusses the tools and technologies necessary for automated testing of IVI systems. These components are crucial for testing methodologies that guarantee the functionality, reliability, and performance of infotainment systems.

5.3.1 CANoe

The CANoe software is crucial for simulating other units in automotive testing environments. Testers can use CANoe to ensure that all potential screens and functionalities are accessible, avoiding the scenario where certain features might be unavailable or missing on the screen due to the absence of real ECUs. This simulation capability is crucial for comprehensive testing and verification of IVI systems and their interactions within the networked automotive ecosystem.

5.3.2 Grimr

Grimr, created by Digiteq Automotive, is a sophisticated tool that translates data captured by the Grabber into visual content on a PC. This feature greatly assists in the detailed evaluation of infotainment systems, enabling testers to interact remotely as if they were physically present. Grimr has proven to be a valuable asset in the testing process over the years, enhancing the depth and scope of testing methodologies.

5.3.3 TestAut2

TestAut2, also developed by Digiteq Automotive, is the main software suite used to manage the test process within vehicle infotainment systems. This tool is tailored to the specific internal needs of Digiteq Automotive. Unlike generic testing solutions, TestAut2 allows users to initiate testing in selected contexts or projects, providing a tailored approach to meet specific testing requirements.

One of the outstanding features of TestAut2 is its ability to compile and present comprehensive test results. This functionality is critical for evaluating the reliability of the system under test and is essential for detailed documentation and evaluation processes.

For more information on how TestAut2 is integrated and used in the test setup, see section 6.2

5.3.4 Git Extensions

As a distributed version control system, Git allows multiple developers to work on the same project without interfering with each other's work. Git Extensions builds on this capability by providing an easy-to-use interface that makes it easier for team members to share changes, track project history and collaborate efficiently.

Backing up code is critical in software development to prevent data loss due to unforeseen circumstances. Git Extensions makes this easier by automating the backup process. Every commit serves as a backup point, allowing developers to revert to previous versions if necessary.

Conflicts are inevitable in collaborative development environments, especially when multiple developers are making changes to the same code base. Git Extensions provide powerful conflict resolution tools, making it easier to identify and resolve conflicts.

5.4 System Architecture of Test Bench

The diagram below illustrates the power connection scheme for a test bench setup used in the testing of IVI systems. The configuration shows a 220V power source connected to an UPS, which stabilises the power and provides backup in case of outages. Two distinct power supply units are illustrated: a 12V power supply that energises the Grabber and Quido, and a 14V power supply with remote control that supplies the infotainment display, Gateway Unit, and Infotainment Unit. This configuration ensures that all components receive a stable and controlled power supply, which is essential for the reliable operation of the test bench.

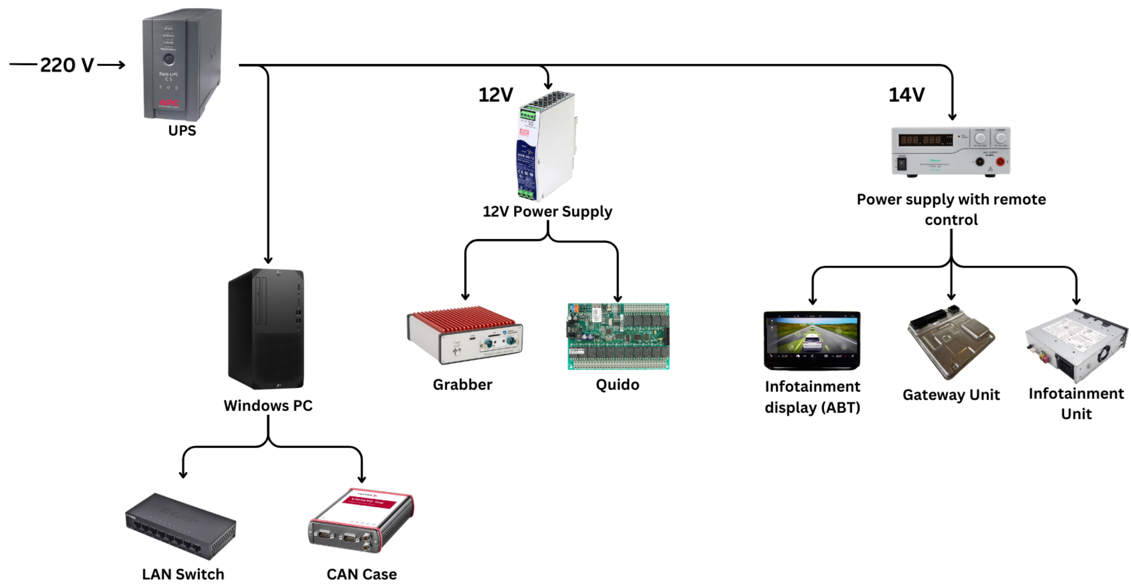

Figure 5.14: Diagram of the power source of the test bench components

The second diagram below illustrates the hardware connection schematic, which depicts the configuration of a test bench. It provides a detailed overview of the interconnection between the various hardware components.

The third and final diagram illustrates the workflow of a software testing setup, demonstrating the interaction between Tcl scripts, CANoe simulation, and the infotainment unit. A Grabber device captures the display, with Grimr software analysing the output, all under the coordination of the TestAut2 system.

Figure 5.15: Hardware connection schematic

Figure 5.16: Software connection diagram

6 Implementation

This chapter is devoted to the practical implementation of automated testing frameworks for IVI systems, with a particular focus on the Škoda Enyaq, which is based on the Volkswagen Group's **MEB** (Modularen Elektrifizierungsbaukasten) platform and is designed exclusively for electric vehicles.

Due to confidentiality agreements, we will be focusing our detailed exploration on the already released and serially produced version of the Enyaq, which is associated with the project **ICAS3**, known internally as the F327. The latest iteration of this platform, known as **ICAS3GP**, is still under development and is covered by a non-disclosure agreement, so we will not be discussing it in this thesis.

This chapter aims to provide a comprehensive overview of the older generation MEB platform through practical examples and methodological insights into the automated test processes used in these systems.

6.1 Understanding Code Structure

To develop and maintain a scalable and efficient automated testing framework for IVI systems, a well-organized code hierarchy is necessary. The **HMI** library serves as the foundation of our testing scripts, incorporating a graded hierarchy of loss of functionality tailored to accommodate the diversity inherent in infotainment systems across different projects and platforms. This hierarchy ensures that code is modularized and reusable, enabling streamlined updates and customizations across various infotainment system versions.

The code hierarchy within the **HMI** library is structured into four primary subdivisions, each representing a layer of specificity in the functionality of the infotainment systems.

Figure 6.1: The code hierarchy within the HMI library

Common Functions (COMMON_FUNC.tcl)

At the top of the hierarchy, common functions are universally applicable across all infotainment systems, regardless of the model or configuration. These functions leverage the standardization within hardware components and communication protocols to provide foundational capabilities. Examples of such functions include:

- **`proc SendCanMsg {msg {msgDelay 20}}`:** Utilizes the standardized CAN protocol to send messages, reflecting the protocol's universal applicability across automotive systems.

- **`proc SetColorLEDBar {color}`:** A function to change the color of the LED bar, exploiting the uniform hardware connection of LED strips across all test benches.

Skin Style Functions

In project F327, the Skin Style layer was not included in the infotainment system's architecture, as it was specifically utilised for Enyaq models on the MEB platform during that period. This layer was designed to meet unique design requirements that were not applicable to the F327 project.

It is important to note the recent shift in the VW Group's strategy towards standardising infotainment systems. In the present era, a unified infotainment platform, including the Skin Style layer, is being implemented across various vehicle platforms with the objective of streamlining the user experience and system maintenance. This represents a significant departure from the past practices observed during the F327 project, where a unified approach was not necessary. This historical insight helps contextualise the project's decisions within the broader technological advancements in automotive infotainment systems.

Platform Functions

This layer focuses on the chassis platforms and tailors functions to the technical and architectural diversity of different vehicle platforms. It acknowledges that variations in hardware and system architecture across platforms can affect how infotainment functions are executed and tested.

ABT (Display Size)

This layer is centered around the physical sizes of the display units within the vehicles. Functions here are designed with awareness of how display dimensions can influence user interface elements, such as the extent and manner of scrolling actions required to navigate the system.

Skin (Regional or Functional Variations)

The lowest level of the hierarchy deals with regional variations, such as left-hand drive (LHD) versus right-hand drive (RHD), or specific functional adaptations, such as language-specific skins. To illustrate, the ARAB skin caters to Arabic-speaking users by adjusting the user interface to accommodate right-to-left reading, thereby ensuring usability and intuitive navigation. These adaptations enhance both the comfort and safety of users by aligning the system's functionality with local driving styles and linguistic norms. An example of a skin-specific function is:

- **`proc ScrollUpOnePage{{ctx 'DEF'}}`:** This function depends on the ABT size and skin configuration, accounting for the pixel dimensions of a page and the positioning of scroll bars relative to the vehicle's drive orientation.

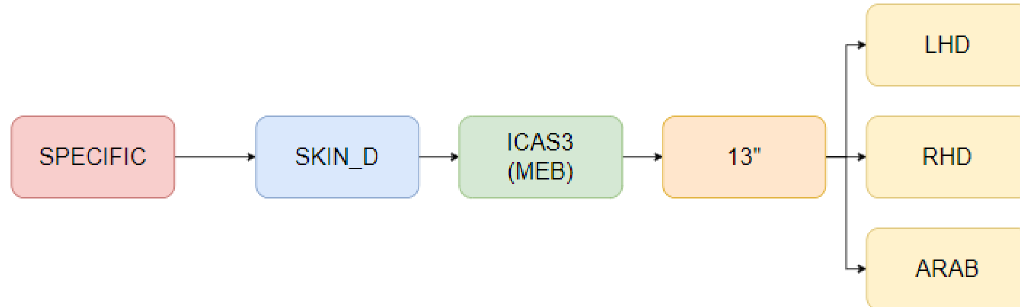

Figure 6.2: Example of code hierarchy for F327 project (Enyaq)

6.1.1 Implementation Strategy

The use of hierarchical organization enables a layered approach to the development and maintenance of test scripts. This is achieved by segmenting functions according to their applicability and specificity, allowing for:

- **Maximize Code Reusability:** Using common functions provides a shared toolkit that can be leveraged across all testing scenarios, minimising the need for duplication.
- **Enhance Maintainability:** Updates to universal features or protocols can be implemented in the common functions layer, automatically propagating across all tests.
- **Streamline Customization:** The subdivision-specific layers allow for targeted customization of test scripts, accommodating the unique aspects of different infotainment systems with minimal impact on the overall codebase.
- **Improve Test Precision:** By taking into account the specificities at each level of the hierarchy, test scripts can more accurately reflect the real-world operation and user interaction scenarios of each infotainment system version.

The hierarchical grading system in the **HMI** library demonstrates a strategic approach to managing the complexity and diversity of automated testing for IVI systems. This structured framework enhances the adaptability, efficiency, and accuracy of the testing process, enabling it to keep pace with the rapid evolution and diversification of infotainment technologies in the automotive industry.

6.2 Test Environment Setup

The establishment of a comprehensive test environment is of paramount importance for the validation of IVI systems. The TestAut2 software provides a robust platform for this purpose, enabling testers to select specific projects and types of tests to run.

Figure 6.3: Interface of TestAut2

Selecting the Project

The process commences with the selection of an appropriate project from a drop-down menu located in the top left corner of the interface. This drop-down menu contains all available projects, thereby ensuring that testers can readily identify and select the project most pertinent to their testing objectives.

Preparing the Infotainment System

Before running tests, it is essential to ensure that the infotainment system is prepared in the correct state. This involves a series of initialization functions, such as sourcing the latest function versions, setting the appropriate voltage, and defining platform variables contingent on the selected project. This step is crucial as it lays the groundwork for reliable and consistent test results.

Choosing Test Contexts

The left panel of the TestAut software presents a series of pre-defined sets of tests, categorised into two primary folders: `HMI_FULL` and `HMI_DEVELOPMENT`.

The `HMI_FULL` folder encompasses all context areas, with TestCases grouped according to the infotainment area they assess. For instance, parking-related screens are grouped under a specific context, as are the vehicle appearance screens (referred to as skin0). This organisation facilitates the generation of clear reports and the streamlining of the setup of separate testing environments.

The `HMI_DEVELOPMENT` folder contains the ”Simple test” section, which enables the validation of TestCases during the development phase. This acts as a sandbox for testers, allowing them to ensure that everything functions correctly before proceeding to full-scale testing. The ”Showoff” subfolder provides a suite of TestCases tailored for demonstrating the infotainment system's capabilities, which is particularly useful for presentations to stakeholders, such as management.

Timing and Execution

The TestAut interface is characterised by a timer that displays both the current time and the accumulated test time. This feature is of particular benefit to testers, as it allows them to manage their testing schedules and provides an estimate of the duration taken to run a particular set of tests.

6.3 Writing and Preparation of Test Scenarios

The quality of IVI depends heavily on thorough testing procedures. To ensure this, meticulously crafted test scenarios that are robust, repeatable, and reflective of the myriad of real-world situations that drivers and passengers encounter are paramount.

6.3.1 Test Scenario Structure

In the domain of automated testing for IVI systems, structuring test scenarios involves defining the parameters within a TestCase. The template structure of a TC serves as the blueprint, guiding the testing of specific functionalities within the IVI.

Defining a TestCase

A TestCase is a fundamental unit in test automation that encapsulates the conditions under which a test is conducted and the expected result. In the context of IVI systems, the TestCase template comprises several key parameters:

Table 6.1: TestCase parameters description

Parameter	Description
context	The context to which the tested screen belongs and will be executed
area	A screen snippet that is conceptually relevant to the TC
mode	The mode in which we want to record the screen (e.g. classic, scroll)
screenName	The identifier of the screen to be tested, as it is named in the model
refVersion	The reference version of the screen
endCondition	The end condition for which the screenshot is to be taken

- ***path***: Denotes the sequence of steps necessary to navigate to the target screen within the infotainment system.
- ***prepare***: Encompasses actions required to bring the system into the desired state before capturing the screen.
- ***cleanUp***: Outlines the steps to revert the system back to a baseline state, ensuring consistency for subsequent TestCases.

To begin the execution process, navigate to the relevant screen (screenName) using the specified path. Once on the correct screen, initiate the prepare phase to configure the necessary settings or inputs. After the test execution, the cleanUp phase ensures that the system is restored to a default state for consistent appearance and behavior for future TestCases.

Example of a TestCase

This TestCase is designed to assess the functionality of the headlight controls via the infotainment system's graphical user interface. It ensures that users can effectively operate the headlights using the touchscreen controls provided by the infotainment system.

```
1  # Verify that the GUI of headlight controls are functional within the
     infotainment system
2    dict set views CARSETUP_HD_LIGHT_OFF context "car"
3    dict set views CARSETUP_HD_LIGHT_OFF area "SCREEN_INSIDE"
4    dict set views CARSETUP_HD_LIGHT_OFF mode "scroll"
5    dict set views CARSETUP_HD_LIGHT_OFF screenName "CARSETUP_HD_LIGHT"
6    dict set views CARSETUP_HD_LIGHT_OFF refVersion "H42.100.0"
7    dict set views CARSETUP_HD_LIGHT_OFF path {
8      {CANOE::RunCANoeMacro "LightExteriorOn"}
9      {HMI::EnterStageArea "CAR"}
10     {HMI::EnterTab 1}
11     {HMI::TextClick "ROW_0" "Exterior" 28 80}
12     {HMI::SwipeSeekText "SCREEN_INSIDE" "Headlights" 28 "RIGHT"}
13     {HMI::TextClick "SCREEN_INSIDE" "Headlights" 28}
14   }
15   dict set views CARSETUP_HD_LIGHT_OFF prepare {
16     {HMI::CheckBoxSetScrollAll "OFF"}
17   }
18   dict set views CARSETUP_HD_LIGHT_OFF cleanUp {
19     {CANOE::RunCANoeMacro "LightExteriorOff"}
20   }
```

Listing 6.1: Testing the GUI of the headlight controls on the IVI's touchscreen

6.4 Setting Up the Screen

The stage of 'Setting Up the Screen' represents a pivotal step in the test execution phase for IVI systems. It involves navigating through the interface of the Škoda Enyaq, and configuring various elements, including dropdowns, sliders, and checkboxes, to their required states for testing. This section will elucidate the structured approach to accessing and preparing the screen within the infotainment system, as per the TestCase requirements set out in the subsection 6.3.1.

Accessing the Screen

In order to access the screen, or "path" in a TestCase, a sequence of function calls must be executed. These direct the system from the home screen to the target screen. Each function must be meticulously designed to interact with the infotainment system's interface. This is to ensure that the test can navigate through the menus and options without human intervention.

Preparing the Screen

Once the desired screen has been accessed, the next step is to bring it into the requested state, which is termed "prepare". For instance, if a dropdown menu needs to be set to its first or last item, a function will execute this specific command. Similarly, sliders are adjusted to their values, and checkboxes are toggled on or off, depending on the test scenario. The preparation of the screen ensures that the initial conditions of the test are met prior to the commencement of the actual testing.

The detailed steps for each action, including the underlying Tcl scripting for "path", "prepare" and "cleanUp" sequences, will be illustrated with examples. The functions that select items based on index position will be discussed in relation to dropdowns. Functions that set values using absolute positions will be the focus of the discussion in relation to sliders. The toggle functions will be discussed in relation to checkboxes, ensuring that they reflect the expected states.

Images of the Škoda Enyaq's infotainment system, which illustrate elements such as dropdown menus, sliders, and checkboxes, will be provided to supplement the explanations, thus facilitating a visual understanding of the configurations. Correctly setting up the screen is a prerequisite that can have a significant impact on the outcome of subsequent tests. Therefore, it is essential to pay close attention to detail and to have a comprehensive understanding of the capabilities of the HMI library when setting up the screen effectively.

6.4.1 Click Functions

The ability to click is a fundamental aspect of GUI testing in automated infotainment systems. It enables testers to simulate user interactions with the touchscreen interface. This section provides an overview of various click functions and discusses their implementations and usage in the context of sending CAN messages to replicate touch inputs.

Basic Click Function: ClickAt

The '`ClickAt`' function represents the fundamental method employed to simulate pressing actions at specific coordinates on the infotainment display. It operates by generating two CAN messages: one for the "Press" action and another for the "Release" action. The process involves a straightforward mathematical conversion where x and y coordinates are translated into CAN messages directly associated with the touch points on the screen.

Although '`ClickAt`' provides the fundamental functionality for a click operation, it is not the most robust method for interacting with GUI elements. This is because it requires exact coordinates as inputs and does not account for dynamic content or changes in element positions.

Advanced Click Functions: TextClick and ImgClick

To enhance the reliability of click operations, functions such as '`TextClick`' and '`ImgClick`' are employed. These functions improve upon '`ClickAt`' by incorporating a search mechanism to automatically locate the elements on the screen.

- `TextClick`: This function searches for a text element on the screen. Once found, it retrieves the coordinates of this text and calls '`ClickAt`' to perform the click action at the appropriate location.
- `ImgClick`: Similar to '`TextClick`', but it searches for an image element. After locating the image, it uses its coordinates to simulate a click.

These functions are particularly useful in scenarios where the positions of elements might change or when exact coordinates are not known beforehand. In order to test elements on scrollable screens, it is necessary to utilise the '`TextClickInList`' and '`ImgClickInList`' functions, which have been designed to handle dynamic content that may not be visible in the initial viewport.

6.4.2 Checkbox Controls

Checkbox controls represent an essential component of IVI system interfaces, enabling users to toggle settings on or off with a simple touch interaction. In the context of automated GUI testing, it is imperative to test the functionality and state accuracy of Checkbox controls. This section will discuss the functions of '`CheckBoxSetAll`' and '`CheckBoxSetScrollAll`' and their role in automating the testing process.

Checkbox controls provide a visual and interactive method for users to make binary choices in the infotainment system. The Checkbox typically displays two distinct visual states:

- **OFF State**: Usually represented by an empty box or an icon with a contrasting background indicating that a particular feature or setting is disabled.
- **ON State**: Indicated by a filled-in box or highlighted icon, signaling that the feature or setting is active.

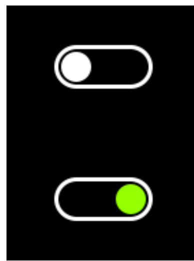

Figure 6.4: Illustration of the 'OFF' and 'ON' status of checkboxes

The transition from the OFF to the ON state (or vice versa) is accompanied by a visual cue, such as a checkmark or a colour change, and, in some cases, an auditory feedback to confirm the action.

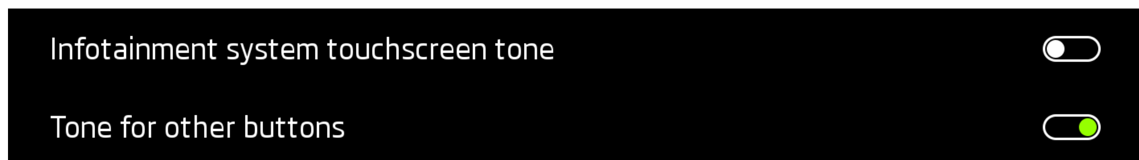

Figure 6.5: Displaying the transition states of checkboxes for system preferences

The implementation of these functions involves the identification of checkbox controls within the GUI through visual representation and the determination of their current state. In the event that a checkbox is in the OFF state and the target state is ON, the function initiates a click event at the checkbox's coordinates,

effectively toggling it to the ON state. Conversely, the reverse is done for the setting of checkboxes to OFF.

6.4.3 Dropdown Controls

Dropdown controls offer a user-friendly interface for selecting from a list of options. These controls are crucial for settings that offer multiple selections, where the user must pick one. This element typically displays a default selection and, when interacted with, presents a list of options for the user to choose from. The user's choice can trigger various behaviours in the system, from changing settings to commanding actions.

Figure 6.6: View of the dropdown control in the Enyaq infotainment system

Automated testing of dropdown items requires functions that can navigate and interact with the dropdown list. This includes selecting specific items, whether it's the first, last or any other item in the list. The '**`DropDownSet`**', '**`DropDownSetAll`**' and '**`DropDownSetScrollAll`**' functions facilitate this process.

The DropDownSet functions typically work by first triggering the dropdown to expand and display its options. They then identify the desired choice, either by index or text, and simulate a click event to make the selection.

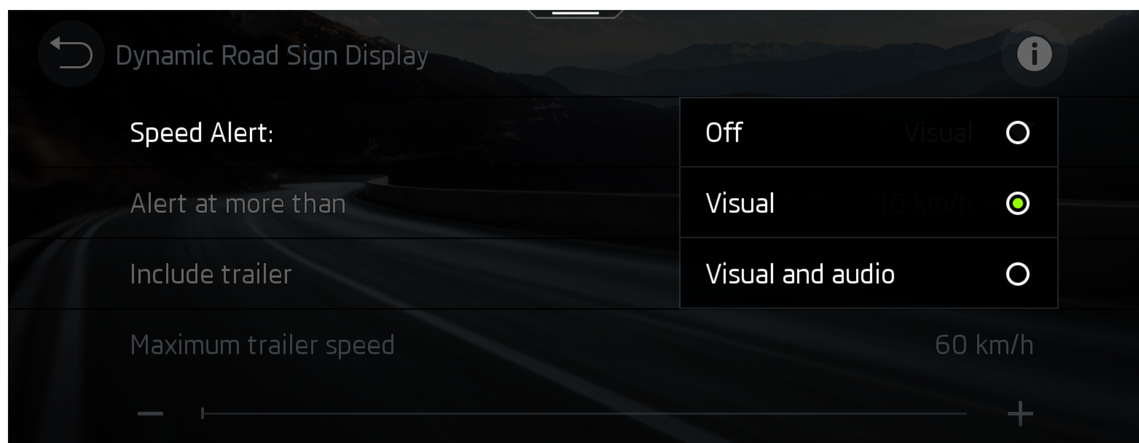

Figure 6.7: Expanded view of the Speed Alert settings dropdown

6.4.4 Slider Controls

Slider controls represent a versatile class of user interface elements that permit users to adjust values within a predefined range. They are frequently employed in IVI systems for the purpose of modifying settings such as volume adjustment or setting alert thresholds. Sliders can be oriented horizontally or vertically and are typically employed to represent a value. They provide a rapid and intuitive method for users to input values, rendering them a user-friendly option for adjusting settings such as sound volume or system preferences.

In the context of automated testing, the functionality of a slider is tested by identifying its ends and calculating its range. This process typically involves the following steps:

1. **Detection**: The test script searches for the '*minus*' and '*plus*' symbols that denote the ends of the slider. These symbols represent the minimum and maximum values, respectively.

2. **Calculation**: Once detected, the script calculates the length of the slider track between the '*minus*' and '*plus*' symbols.

3. **Position Setting**: Subsequently, the script calculates the coordinates for the minimum, midpoint, and maximum positions on the slider track.

4. **Interaction**: Finally, the script simulates click events at the calculated coordinates to set the slider to minimum, midpoint, and maximum values.

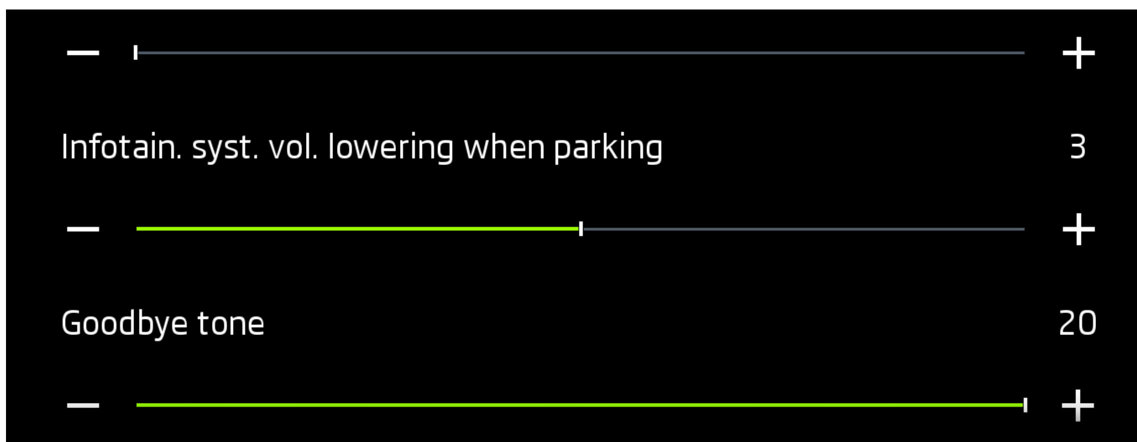

Figure 6.8: Audio settings slider controls

It is important to note that in automated testing scripts, sliders are often set to only their minimum, midpoint, or maximum values due to the complexity involved in setting arbitrary values. This approach provides a reasonable assurance of functionality across the slider's range.

6.5 Screen Capture

Once the screen has been configured in accordance with the requisite specifications, the subsequent crucial stage in the assessment of the new HMI version is the screen capture process. This entails the precise imaging of the display in order to analyse the layout and functionality under a range of conditions. Screen capture is not merely a recording of the content displayed on the screen; it is also about capturing the interface in a manner that reflects how end-users will interact with the system.

This section will examine the various techniques and tools employed for screen capture. The process of screen splitting will be discussed, which allows the user to isolate and capture specific areas of interest on the screen. This is particularly useful for focusing on elements that require detailed evaluation or are critical to the user experience.

Additionally, the different modes in which screens can be captured will be explored. These modes are designed to accommodate various scenarios, such as scrolling, which is necessary when the content extends beyond the visible screen area. They also allow for the capture of screens during different states, such as loading sequences or when dropdown menus are active. It is, therefore, essential to understand these modes in order to ensure that all dynamic and static elements of the HMI are accurately documented.

6.5.1 Screen Splitting

Screen splitting is a strategy employed in automated testing that focuses on the examination of specific areas within an infotainment display. This targeted approach enables testers to concentrate on discrete components of the interface, streamlining the setup process and reducing complexity. It eliminates the need to account for visual settings beyond the area of interest, enhancing testing efficiency.

Key Areas:

- `DISPLAY`: Represents the entire screen.
- `SCREEN_INSIDE`: Refers to the central section of the display where main interactions occur.
- `MAINBAR`: The bottom bar area, often containing navigation or system controls.
- `TABBAR`: The section dedicated to tab navigation, crucial for functionality but isolated for focused testing.
- `SCREEN_LIST`: A modified `SCREEN_INSIDE` where the `TABBAR` is absent, thus the central screen is wider.
- **PopUp Areas**: Targeted regions intended for popup dialog testing.
- **Homescreen Tiles**: Specific tiles on the homescreen can be isolated to test individual features or alerts.

The utilisation of screen splitting enables testers to automate the capture and analysis of designated regions, such as the `TABBAR`, without the distraction of surrounding elements. This approach is particularly beneficial for regression testing, where changes in one area should not affect the overall interface.

The Figure 6.9 exemplifies the `MAINBAR` (red), `TABBAR` (green), and `SCREEN_INSIDE` (orange) areas in the context of a 'Vehicle status' screen. The `MAINBAR` spans horizontally at the bottom, the `TABBAR` is situated above it, and the `SCREEN_INSIDE` covers the central display section showcasing the vehicle's illustration and status information.

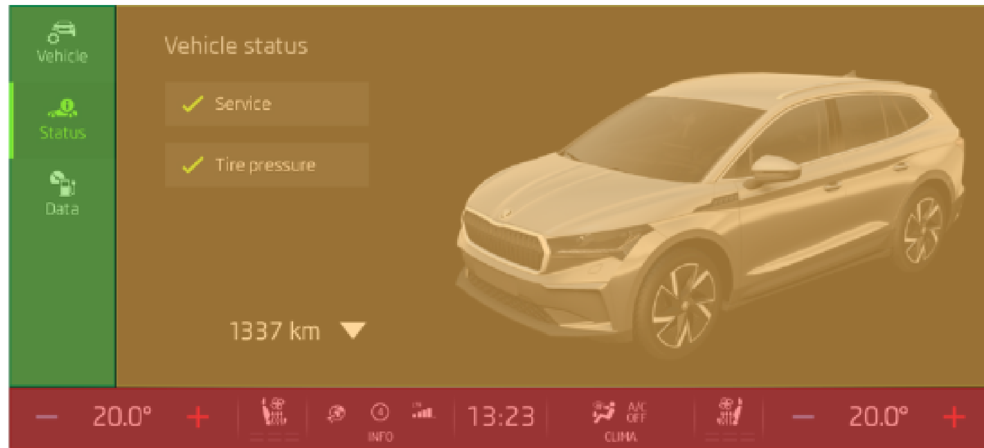

Figure 6.9: Infotainment display segmentation

The Figure 6.10 illustrates the concept of capturing only a single homescreen tile, highlighted in green, concerning 'Driving data'. It demonstrates how, in a report, only this specific part of the screen would be reported, focusing solely on the content within the green frame. This would be used for testing elements such as efficiency, speed, or mileage data, without regard to adjacent areas.

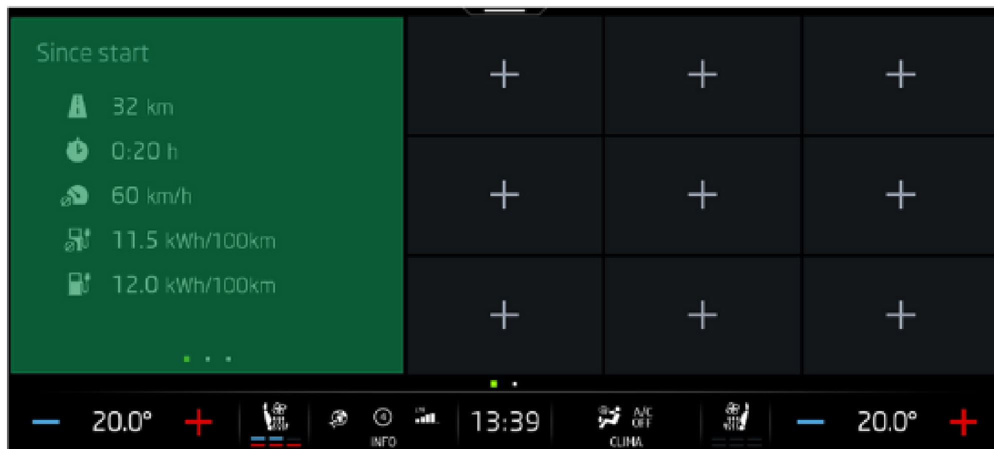

Figure 6.10: Focused testing of the driving data homescreen tile

The two images serve as visual aids in understanding how screen splitting can be used to enhance the efficiency and accuracy of automated testing for IVI systems. The focused areas are highlighted to denote the regions of interest during the automated testing process, thereby providing clarity and precision in the testing results.

6.5.2 Capturing Modes

Automated testing of infotainment systems employs a variety of capturing modes with the objective of enhancing the efficiency and thoroughness of the validation process. These modes are designed to enable testers to effectively record the state of the infotainment system under a range of conditions. By utilising these specific capturing modes, testers can focus on particular elements within the system's interface, ensure consistency in repeated tests, and trigger certain conditions that might not be easily observable during manual testing.

To facilitate comprehension and utilisation of these capturing modes, a comprehensive table is provided below. The table outlines each mode, describes its primary function, and notes any specific conditions or parameters that are relevant to the mode. This organised representation allows testers to quickly reference and select the most appropriate capturing mode for their current testing requirements, thus facilitating a streamlined and targeted testing process.

Table 6.2: Overview of capturing modes

Mode	**Description**
Classic	Captures a single image, cropping based on 'area'.
Classic+Drop	Captures as classic, including all expanded dropdowns.
Scroll	Captures the entire scrollable list, with a maximum of 25 images.
Scroll+Drop	Like scroll, but includes expanded dropdowns.
SideScroll	Captures while horizontally scrolling, with various 'endCondition' options, up to 25 images.
TabBar	Captures all tab bar positions.
TabBarPress	Captures all tab bar positions in a pressed state.
ScrollPress	Captures scrollable screens with touches on each line.
ClassicPress	Captures screen with elements in pressed state as specified by 'endCondition'.
PopUp	Like classic, retains the popup after "path" execution with direct popup name.
PopUpPress	Like classicPress, but maintains the popup post "path" execution.

6.5.3 Understanding the 'endCondition' Parameter

The '`endCondition`' parameter plays a pivotal role in the process of automated screen capturing, functioning as a dynamic criterion that determines when a screen should be captured. It is the defining condition that guides the automated system in determining the precise moment to take a snapshot of the infotainment screen during testing.

1. **Cursor:** This condition is used when screens have a blinking cursor. Multiple snapshots are taken during the test, and the one with the active (visible) cursor is saved.

2. **Loading:** Applicable to screens with a rotating "loading" indicator. The condition involves capturing several images, comparing the quantity of identically colored (green) pixels, and saving the one with the highest count (indicating the largest segment of the loading icon).

3. **Animation:** Used for screens with changing animations. The capture process involves taking several images during the animation sequence and comparing each with a reference image to determine the optimal snapshot to save. This ensures that the captured screen reflects the expected state of the animation at a specific point in time.

4. **Numerical Value (e.g., 5):** Useful for 'scroll' and 'sidescroll' modes, it signifies the capture of a specified number of screens.

5. **String:** When '`endCondition`' is a string, the capture continues until a specific text is found on the screen.

```
1 dict set views SYSTEM_UPDATES endCondition {"loading"}
```

Listing 6.2: Implementation example for endCondition

The '`endCondition`' parameter plays a pivotal role in the fine-tuning of the automated testing process, enabling the precise capture of interactive elements and dynamic content within infotainment systems.

6.6 Comparison of New Images with Reference and Detection of Differences

This section outlines an automated comparative analysis framework that leverages image recognition to validate HMI graphics against predefined standards. The cornerstone of this framework is the establishment of a reference image, which serves as the benchmark for graphical fidelity. This image is thoroughly vetted by testers and confirmed to align with graphical specifications. The reference image encapsulates the desired appearance of the HMI, including iconography, text placement, and colour schemes.

The system automatically compares the reference image with newly captured screens from the latest HMI or software versions on a weekly basis. Employing ImageMagick software, the comparison is executed, scanning for deviations pixel by pixel. This rigorous check ensures any alterations, whether intentional or inadvertent, are identified and assessed. For an in-depth discussion on the functionalities and applications of ImageMagick in automated testing, see section A.2.

The output of this comparison is a color-coded report reflecting the degree of similarity between the tested image and the reference:

- **Green:** Denotes an exact match, signifying no discernible difference from the reference image.

Figure 6.11: Exact match verification: Green indicates identical screens

- **Yellow:** Represents a minor discrepancy with less than 1% pixel difference detected.

Figure 6.12: Minor discrepancy detected: Yellow for under 1% pixel difference

- **Red:** Signals a significant difference exceeding 1% pixel variation.

Figure 6.13: Significant variation: Red for over 1% pixel difference

- **Blue:** Indicates the absence of a reference image, prompting a manual check to establish a new standard if the screen meets quality criteria.

Figure 6.14: Manual review required: Blue when reference is missing

In addition to the aforementioned primary indicators, the framework also identifies error states, which may arise from syntax errors in the TestCases or unexpected test duration, leading to an automatic termination. These error states are highlighted distinctly to alert testers to anomalies that could affect the testing process.

Figure 6.15: Error indication: Highlighted when a test issue occurs

The implementation of this image comparison framework enables a systematic and efficient approach to HMI graphic verification. It reduces the potential for human error, standardises quality control, and ensures consistency across software updates. By facilitating the quick identification of discrepancies and errors, it supports developers in maintaining the high standards expected in modern IVI systems.

6.6.1 Case Study: Detecting and Analyzing Interface Discrepancies in HMI Updates

In the ongoing evolution of IVI systems, each software update may result in intended enhancements as well as unintended discrepancies. The following case study illustrates the process of identifying and evaluating changes within the GUI to distinguish deliberate modifications from potential bugs.

Reference Image Analysis

The reference image serves as a standard for graphical accuracy in the HMI interface. In this case, the reference image is that of a phone dial screen within the infotainment system. It features a clear and user-friendly layout with essential functions like number input and quick access to emergency and voicemail services.

Analysis of Recent HMI Update

In the revised version of the HMI, the new image displays a modification to the voicemail label, which now reads as ”Voicemail” in place of the original ”Mailbox” text. This change, while seemingly minor, represents a graphic update that must be verified to ensure it aligns with the intended design revisions. The rest of the dial interface remains unchanged, maintaining the integrity of the design and user interaction experience.

Differential Analysis

The differential image highlights the specific area of change between the reference and the new image. Utilizing red to indicate discrepancies, the altered ”Voicemail” label is distinctly marked, contrasting against the unaltered elements which blend seamlessly with the reference background.

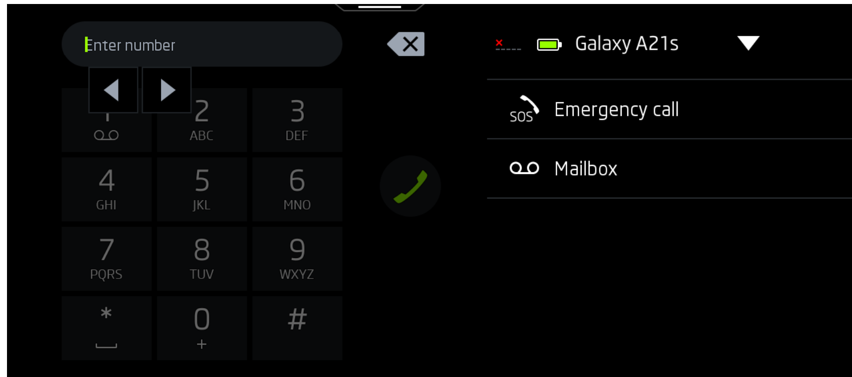

Figure 6.16: Reference image

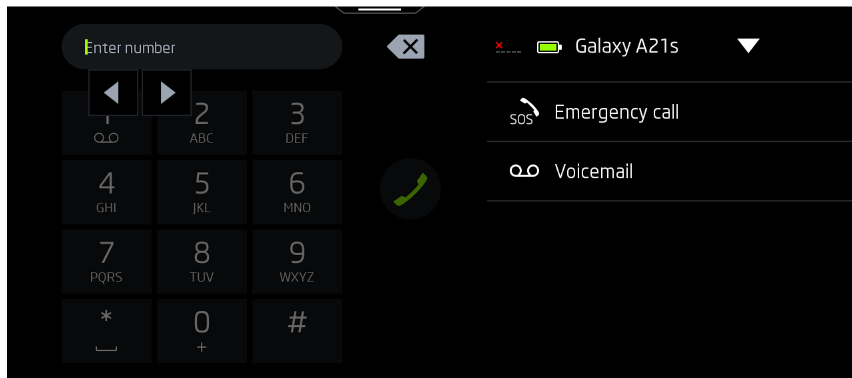

Figure 6.17: Image of new HMI update

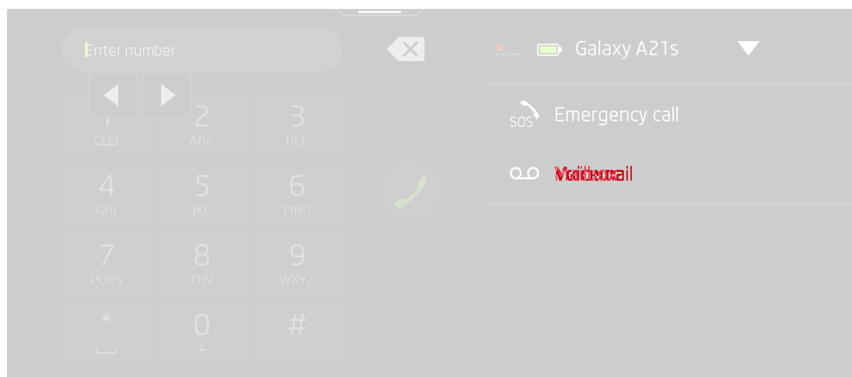

Figure 6.18: Differential image

Upon the detection of a variation, the subsequent crucial stage is verification. This process entails consulting the change logs, design documents or communicating with the design and development teams to ascertain whether the modification was intentional. Only through this thorough examination can a change be validated or flagged for correction.

6.6.2 Case Study: Identifying and Resolving Interface Bugs in HMI Updates

The current case study examines the detection of a significant interface bug, which deviates from the established HMI design guidelines and functional expectations.

Reference Image Analysis

The standard interface layout for radio settings is presented in the reference image. It comprises a well-structured arrangement of clearly defined options, each accompanied by a standardised checkbox. This design consistency is of critical importance for both aesthetic appeal and user interaction.

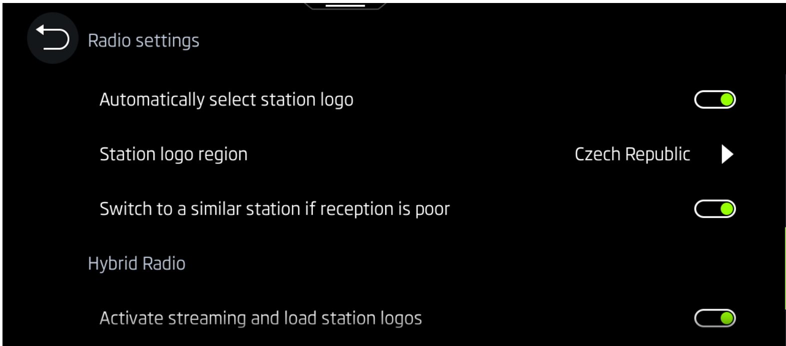

Figure 6.19: Reference image

Analysis of Recent HMI Update

Upon examination of the latest iteration of the HMI, an anomalous row is immediately apparent. This comprises an additional setting with an unconventional checkbox, devoid of any corresponding title. This irregularity deviates from the established design language of the interface, suggesting the possibility of an oversight or bug rather than an intended update.

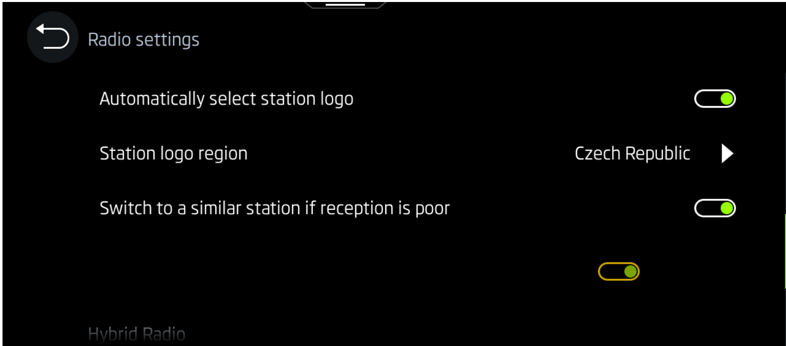

Figure 6.20: Image of new HMI update

Differential Analysis

The differential image is stark, with the unlabelled row and non-standard checkbox vividly contrasted against the standard backdrop. This visual cue is crucial for developers and testers alike, as it draws attention to specific areas requiring immediate remediation.

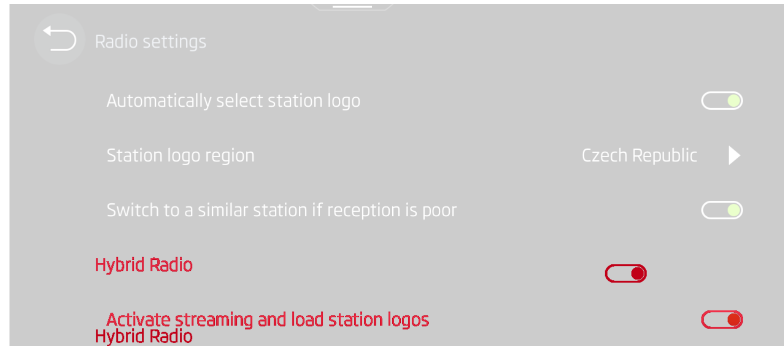

Figure 6.21: Differential image

Upon identifying this deviation as a bug, the subsequent step is to document and report it to the skinning team. By providing a detailed report, it is possible to implement targeted corrections, ensuring that the interface adheres to the project's standards and maintains the continuity of the user experience.

This case study serves as an example of proactive bug detection and resolution in the ongoing refinement of IVI systems. This case study illustrates the necessity for a meticulous approach to interface verification post-updates, assuring that any anomalies are swiftly identified and resolved.

6.7 Final Report on Automated Infotainment System Testing

The final report represents the culmination of an automated testing session for an infotainment system. It is a comprehensive and visually organised document that articulates the status and graphical representation of the system's interface. This final report is not merely a collection of data; it is a synthesised presentation of results, offering clear insights into the aesthetics of the infotainment system's GUI. The report typically commences with a graphical summary, which provides an overview of the test outcomes in a concise manner. As illustrated in the provided images, this encompasses a count of various test result categories, including:

- Number of TestCases (TCs)
- Total screens captured
- Summary of tests passed, failed, or with errors

Test results count

Sums of elements in report	
NO_DATA	118
PASSED	1166
MIDDLE_NOK	41
FAILED	44
SYNTAXERROR	0
MIDDLE_NOK + FAILED	85
TOTAL SCREENSHOTS	1369
TESTCASES ACTIVE	839
TESTCASES ALL	867
JIRA ACTIVE	444
JIRA ALL	455

Figure 6.22: An example of the numerical result of one of the tests report

This graphical representation enables stakeholders to rapidly assess the efficacy of the testing process and identify areas in need of attention. It constitutes a fundamental component of the report, frequently presented in HTML format to integrate detailed data with user-friendly visuals.

A significant aspect of the final report is the collection of new images captured during testing, including those that demonstrate discrepancies from the anticipated outcomes. These images are frequently resized to align with the report's format, thereby providing clear evidence of the system's current state in comparison to the baseline or reference images.

Comprehensive logs are also part of the final report, capturing the session's intricate details:

- `ConsoleLog.txt`: A log file from the Tcl console, which records the commands and outputs during the test execution.
- `Log.html`: A log from TestAut2, the automated testing tool, providing an HTML formatted view of the testing process.
- `Test.xml`: A settings file for TestAut2 that details the configuration and parameters used in the testing session.

Moreover, the report includes an archive of every screen from previous reports (versions HMI), stored externally. This historical archive is indispensable for comparative analysis, allowing testers to observe changes over time and assess the progress or regression of the system's graphical interface.

02:30:03 ---------- CARSETUP_MAIN_EXTERIOR_1_0303010623000000 ---------- CARSETUP_MAIN_EXTERIOR_1	CARSETUP_MAIN_V2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:30:24 ---------- CARSETUP_MAIN_EXTERIOR_2_0303010623000000 ---------- CARSETUP_MAIN_EXTERIOR_2	CARSETUP_MAIN_V2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:30:44 ---------- CARSETUP_MAIN_EXTERIOR_3_0303010623000000 ---------- CARSETUP_MAIN_EXTERIOR_3	CARSETUP_MAIN_V2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:31:18 ---------- CARSETUP_MAIN_EXTERIOR_4_0303010623000000 ---------- CARSETUP_MAIN_EXTERIOR_4	CARSETUP_MAIN_V2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:32:17 ---------- FAS_VIEW_HOME_ON_0303010623000000 ---------- FAS_VIEW_HOME_ON	CAR_FAS_2_HOME ---------- notes:	H55.49.4				0px ~ 0.0000%
02:33:27 ---------- HOMESCREEN_VEHICLE_3_REPORTS_0303010623000000 ---------- HOMESCREEN_VEHICLE_3_REPORTS	HOMESCREEN_2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:34:43 ---------- HOMESCREEN_VEHICLE_3_TYRES_0303010623000000 ---------- HOMESCREEN_VEHICLE_3_TYRES	HOMESCREEN_2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:35:31 ---------- HOMESCREEN_NAVIGATION_2x2_0303010623000000 ---------- HOMESCREEN_NAVIGATION_2x2	HOMESCREEN_2 ---------- notes:	H55.48.2				0px ~ 0.0000%
02:35:45 ---------- CARSTATUS_MAIN_0303010623000000 ---------- CARSTATUS_MAIN	CARSTATUS_MAIN ---------- notes:	H55.48.2				0px ~ 0.0000%
02:36:11 ---------- CARSTATUS_SERVICE_0303010623000000 ---------- CARSTATUS_SERVICE	CARSTATUS_MAIN_FAVORITE_SERVICE ---------- notes:	H55.48.2				3046px ~ 0.4249%
02:36:36 ---------- CARSTATUS_TYREPRESSURE_0303010623000000 ---------- CARSTATUS_TYREPRESSURE	CARSTATUS_MAIN_FAVORITE_TIRESSTATUS ---------- notes:	H55.48.2				0px ~ 0.0000%
02:36:54 ---------- CARBOARDCOMPUTER_0303010623000000 ---------- CARBOARDCOMPUTER	CARBOARDCOMPUTER_MAIN ---------- notes:	H55.48.2				0px ~ 0.0000%
02:37:16 ---------- CLIMATE2PRECONDITIONING_0303010623000000 ---------- CLIMATE2PRECONDITIONING	CLIMATE2PRECONDITIONING ---------- notes:	H55.48.2				0px ~ 0.0000%
02:37:35 ---------- SOUND_POSITION_FOCUS_FRONT_0303010623000000 ---------- SOUND_POSITION_FOCUS_FRONT	SOUND_POSITION ---------- notes:	H55.48.2				0px ~ 0.0000%

Figure 6.23: Sample part of the final report

7 Results and Analysis

Automated testing is a crucial aspect of ensuring the reliability and quality of IVI systems, such as the one found in the Škoda Enyaq, which represents one of the most comprehensive systems offered by the manufacturer. This chapter examines the capabilities of automated testing, the current scope of the test bench in use, and the outcomes of these testing sessions.

7.1 Test Automation Results

The current test bench provides extensive testing, often exceeding 1000 screenshots that capture a broad range of scenarios within the infotainment system. These screenshots verify the visual and interactive aspects of the system's GUI, ensuring that all visual elements are rendered correctly and that interactions lead to the expected outcomes.

The number of unique screens and test cases depends heavily on the specific project at hand. In the case of the Enyaq infotainment system, the scale is significantly larger due to the system's complexity and breadth of features.

Over the lifetime of the Enyaq project, this methodical approach to testing has led to the discovery of hundreds of bugs. Each bug identified is a step towards refining the user experience, contributing to a more stable, functional, and user-friendly infotainment system.

In conclusion, the automated testing of the Enyaq infotainment system is a comprehensive undertaking. By encompassing a comprehensive array of screens and test cases, it ensures that the system meets the high standards expected of a leading automotive brand such as Škoda. The automation process not only expedites the detection of bugs but also supports the development team in delivering a robust and reliable product for the end-users.

7.2 Limitations

Automated testing, while transformative in its scope and capabilities, is not without its limitations. Despite advanced technology and methodologies, certain aspects of IVI systems remain beyond the reach of full automation. This section explores the limitations of automated testing, the challenges of achieving full test coverage, and the areas that require further refinement.

Incomplete Test Coverage

One of the main limitations of automated testing is the inability to test all screens automatically. For example, critical functions that rely on online connectivity or real-time data often escape comprehensive automated testing due to the absence of certain devices or live data streams. In particular, screens associated with online services remain untested because they require connectivity to external networks that the test bed may not be equipped to simulate.

Testing Voice Assistants

The interactive nature of voice-activated features presents another significant hurdle. Automated test systems cannot physically reproduce spoken commands, which is essential for validating voice assistant functionality. The lack of this capability leaves a gap in system verification, especially as voice commands become an increasingly common form of user interaction in modern vehicles.

CANoe Simulation Challenges

The limited availability of real vehicle units on the test bench necessitates the use of simulators such as CANoe to mimic the remaining systems. However, achieving harmony between the real and simulated units is a complex task. Synchronising CANoe simulations with physical components to achieve seamless, representative testing is often challenging and can affect the reliability of test results.

Test Duration and Optimization

Another critical issue is the length of the testing process, with test runs of up to 16 hours. Such long test times indicate an urgent need for optimisation. Efficiency improvements are needed not only to reduce test time, but also to minimise resource utilisation and reduce the potential for errors that can result from prolonged test cycles.

Lack of Automated Flashing via API

The lack of automated flashing of units via an API also limits test flexibility. Currently, units are pre-coded with as many features as possible to maximise screen availability. However, this approach does not address the need to dynamically update or flash units to test different software versions or configurations, leaving some screens inaccessible for testing.

Capturing Complex Animations

Lastly, the automated testing system's capacity to capture screens with complex animations is not foolproof. While the "animation" `endCondition` enhances the ability to record screens during animated sequences, capturing the nuances of intricate animations remains a challenge. Ensuring that the captured screen matches the reference precisely, particularly with sophisticated animations, is an area where automated testing can falter.

In summary, while automated testing has made significant progress in advancing the validation of IVI systems, it has yet to overcome certain inherent limitations. Overcoming these challenges will require continued development of the test infrastructure, incorporation of more sophisticated simulation techniques, refinement of test optimisation processes, and improved integration of real and simulated components. As the complexity of IVI systems increases, so must the ingenuity and capability of automated test methodologies.

7.3 Test Automation Cost

The implementation of automated test methods in IVI systems requires a significant up-front investment. When assessing the financial requirements for the deployment of these systems in 2024, it is important to note that these costs are estimates based on current market conditions and are subject to change.

The main costs of setting up an automated test environment are the purchase of specialised hardware components and software licences. The estimated total investment required to set this up is approximately **€26,000**. This estimate is based on market analysis and includes the essential components listed for a fully functional test environment.

A significant portion of this budget is allocated to the CANoe software licence, which represents almost half of the total cost.

7.3.1 Exclusions in Cost Estimation

This financial estimation exclusively accounts for the tangible assets required for the automated testing setup. It is important to highlight that all vehicle units utilised during the testing phases are provided by Škoda Auto at no cost. These development units, while challenging to quantify in price, represent a significant value addition and reduce the overall financial burden of the testing process.

It should be noted that the initial cost estimate does not include operational costs such as salaries for employees, as well as electricity and other utilities necessary for system operations.

7.3.2 List of Components and Their Costs

To provide further clarity on the financial aspects of setting up an automated testing system, a list of key items and their approximate prices is provided below.

- **CANoe License and CAN Case:** €12,000 - A comprehensive tool for ECU testing and network simulation.
- **Test Racks and Fixtures:** €3,000 - Custom-designed fixtures and racks for mounting and interfacing with infotainment units.
- **High-Performance PCs:** €2,000 - Computers equipped with advanced specifications to handle the software and testing data.
- **Networking Equipment:** €1,500 - Includes routers, cables, and interfaces for establishing a robust testing network.

While the initial outlay for automated test infrastructure may appear considerable, the potential for significant improvements in test efficiency, accuracy and consistency justifies the investment.

8 Future Directions

As the automotive industry continues to evolve, the technologies embedded within vehicles, particularly IVI systems, are becoming increasingly sophisticated. This chapter examines potential future directions for automated testing in this domain, taking into account current trends in technology development across various automotive manufacturers.

8.1 Transition to Android-Based Infotainment Systems

One of the most prominent trends in the automotive industry is the adoption of Android-based interfaces for IVI systems. Companies such as Audi, a member of the Volkswagen Group, have already initiated the integration of infotainment systems based on Android applications. This transition towards Android provides a more flexible platform for app development and integration, potentially offering a more enriching user experience through customisable interfaces and a broader range of applications.

Given the close relationship and technology sharing within the Volkswagen Group, it is reasonable to anticipate that Škoda Auto may also adopt Android-based infotainment systems in the future. Typically, the technology seen in Audi vehicles trickles down to Škoda models within a span of three to five years. This would suggest a probable shift in Škoda's approach to infotainment solutions in the near to mid-term future.

8.2 Future Research and Development

The introduction of Android interfaces requires a significant change in the approach to automated testing. Currently, testing focuses primarily on hardware reliability and the integration of native software within the infotainment units. However, with

Android at the heart of these systems, automated testing would need to evolve to validate Android applications specifically designed for use in vehicles.

To accommodate these changes, research and development efforts in the field of automated testing will need to focus on creating more dynamic and flexible testing frameworks. These frameworks should not only support existing protocols but also adapt to the ever-changing software landscape introduced by Android and other mobile platforms.

8.3 Speculative Outlook

The specific technological trajectory of Škoda Auto is a complex matter to predict. However, observing trends in sister companies like Audi, which has integrated Android-based systems into its infotainment offerings, suggests a potential direction for Škoda. While it seems plausible that Škoda might adopt similar technologies in the coming years, this remains an educated guess rather than a certainty.

The dynamic nature of the automotive industry, with its continuous innovations and changing market forces, means that any predictions must be viewed as tentative. This highlights the need for automated testing frameworks to remain flexible and capable of adapting to new technologies as they emerge.

9 Conclusion

This thesis is centred around the implementation of automated testing for IVI systems. With the increasing complexity and functionalities of modern automotive software, automated testing offers a scalable solution for ensuring system reliability and user satisfaction.

A fundamental aspect of implementing automated testing is the understanding of the technical foundations, including the CAN protocol and the Tcl script language. These technologies are of paramount importance for the creation and management of tests, due to their inherent robustness and flexibility in handling device communication and script automation.

The workflow of automated testing significantly enhances the efficiency of the testing process. By automating repetitive tasks and facilitating regression tests, the process allows for continuous improvements and faster development cycles, ensuring that each version of the software meets high-quality standards before deployment.

An in-depth exploration of the hardware and software components used in the testing environment is documented in the thesis. Detailed connection diagrams provide a clear visualisation of how these components interact, offering insights into the complexities of the testing setup.

Furthermore, the thesis introduces a comprehensive library of HMI functions. This library contains hundreds of functions that are essential for testing various aspects of the infotainment system. Only a selection of these functions is documented within the thesis, reflecting a selection of the most critical features necessary for comprehensive testing.

A specific case study on the Škoda Enyaq is included to demonstrate the application of these methodologies in a real-world context. The appendix of the thesis contains all the test cases developed for this project, providing a practical perspective on how automated testing is applied to specific vehicle models.

The results of these tests are compiled into detailed reports, complete with visualisations that highlight the performance and outcomes of the testing process. These reports serve as a crucial tool for developers to identify and address potential issues swiftly.

Despite the advancements, the thesis does not shy away from discussing the limitations encountered. Issues such as scalability across different hardware configurations and occasional inconsistencies in test outcomes are highlighted, underscoring areas for future improvement.

Currently, Digiteq Automotive is utilising these automated testing methodologies to evaluate the successors of Škoda Octavia, Superb, Kodiaq, and Enyaq. This practical application across five test benches illustrates the industry's confidence in automated testing as a critical component of automotive software development.

References

[1] AL DALLAL, Jehad. Automation of object-oriented framework application testing. In: 2009, pp. 425–434. ISBN 978-1-4244-3885-3. Available from DOI: 10.1109/IEEEGCC.2009.5734312.

[2] AUTO, Skoda. *Nový virtuální kokpit, 13" displej* [online]. Skoda Auto, 2021 [visited on 2024-05-05]. Available from: https://cdn.skoda-storyboard.com/2021/03/72_ENYAQ_iV_DPL-1920x1281.jpg.

[3] BILLIGER, Carparts. *SKODA NAVI DAB MULTIMEDIA MIB3 INFOTAINMENT NAVIGATION 5E3035816-B* [online]. Carparts Billiger, [n.d.] [visited on 2024-05-05]. Available from: https://www.carparts-billiger.de/155-Niara_thickbox/skoda-navi-dab-multimedia-mib3-infotainment-navigation-5e3035816-b2x.webp.

[4] BOLAND, Hannah M. et al. An Overview of CAN-BUS Development, Utilization, and Future Potential in Serial Network Messaging for Off-Road Mobile Equipment. In: AHMAD, Fiaz and SULTAN, Muhammad (eds.). *Technology in Agriculture*. Rijeka: IntechOpen, 2021, chap. 25. Available from DOI: 10.5772/intechopen.98444.

[5] BOSCH, Robert. *Can specification, version 2.0*. Bosch, 1991.

[6] CHOI, Dong-Kyu et al. In-Vehicle Infotainment Management System in Internet-of-Things Networks. In: *2019 International Conference on Information Networking (ICOIN)*. 2019, pp. 88–92. Available from DOI: 10.1109/ICOIN.2019.8718192.

[7] CONTRIBUTORS, Wikipedia. *Tcl* [online]. 2023. [visited on 2024-05-05]. Available from: https://en.wikipedia.org/wiki/Tcl.

[8] *dict manual page - tcl.tk* [https://www.tcl.tk/man/tcl/TclCmd/dict.htm]. [N.d.]. [Accessed 26-03-2024].

[9] *Digiteq automotive* [online]. 2023. [visited on 2023-11-21]. Available from: https://www.digiteqautomotive.com/en.

[10] ELECTRIC, Bueno. *Can termination resistors-vital part* [online]. 2022. [visited on 2024-05-05]. Available from: https://www.buenoptic.net/encyclopedia/item/544-can-termination-resistors-vital-part.html.

[11] ELESHOP.EU. *Manson HCS-3602-USB power supply* [online]. 2024. [visited on 2024-05-05]. Available from: https://eleshop.eu/manson-hcs-3602-usb-power-supply.html.

[12] EUROTEIL. *ICAS1 GW* [online]. 2024. [visited on 2024-05-05]. Available from: https://www.jllautoparts.com/product/1ea-937-012-n.

[13] FALCH, Martin. *Can bus explained - a simple intro [2023]* [online]. 2021. [visited on 2024-05-05]. Available from: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial.

[14] GMBH, Vector Informatik. *VN1600* [online]. Vector Informatik GmbH, 2010 [visited on 2024-05-05]. Available from: https://cdn.vector.com/cms/content/products/VN16xx/graphics/VN1630log_Liegend_Unten_web_3200x2000px.jpg.

[15] GRIFFITH, John. *What do can bus signals look like?* [online]. 2023. [visited on 2024-05-05]. Available from: https://www.ti.com/document-viewer/lit/html/SSZTCN3.

[16] HOGSTROM, Christopher and Christopher HOGSTROM. *TCL Loops - Gritty engineer* [online]. 2019. [visited on 2024-05-05]. Available from: https://grittyengineer.com/tcl-loops/.

[17] HP. *HP Z1 G9 Tower Workstation* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://www.hp.com/gb-en/shop/Html/Merch/Images/c08195534_1750x1285.jpg.

[18] IMAGEMAGICK. *Image Magick* [online]. 2024. [visited on 2024-05-05]. Available from: https://imagemagick.org/.

[19] KAN, Hongxing et al. A method of minimum reusability estimation for automated software testing. *Journal of Shanghai Jiaotong University (Science)*. 2013, vol. 18, pp. 360–365. Available from DOI: 10.1007/S12204-013-1406-1.

[20] KARAMBUNAI, Kota. *Controller Area Network (CAN) basic and technical overview* [online]. 2015. [visited on 2024-05-05]. Available from: http://kotakarambunai.blogspot.com/2015/11/controller-area-network-can-basic-and.html.

[21] KUBÁT, Jan. *Automatické testování infotainment jednotek*. 2020. MA thesis. ČVUT.

[22] MACARIO, Gianpaolo, Marco TORCHIANO, and Massimo VIOLANTE. An in-vehicle infotainment software architecture based on google android. In: *2009 IEEE International Symposium on Industrial Embedded Systems*. 2009, pp. 257–260. Available from DOI: 10.1109/SIES.2009.5196223.

[23] OSBORNE, Ben. *What does OBD stand for?* [online]. 2023. [visited on 2024-05-05]. Available from: https://www.noregon.com/what-is-obd/.

[24] PAPOUCH. *Quido ETH 2/16: 2 vstupy, 16 výstupů a teploměr* [online]. Papouch, 2024 [visited on 2024-05-05]. Available from: https://papouch.com/quido-eth-2-16-2-vstupy-16-vystupu-a-teplomer-p4642/?vid=1785.

[25] PFEIFFER, Olaf, Andrew AYRE, and Christian KEYDEL. *Embedded networking with can and Canopen*. First. Copperhill Technologies Corporation, 2008.

[26] POINT, Tutorials. *TCL - bitwise operators* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://www.tutorialspoint.com/tcl-tk/tcl_bitwise_operators.htm.

[27] POINT, Tutorials. *TCL - relational operators* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://www.tutorialspoint.com/tcl-tk/tcl_relational_operators.htm.

[28] RAFI, Dudekula et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012, pp. 1–42. ISBN 978-1-4673-1821-1. Available from DOI: 10.1109/IWAST.2012.6228988.

[29] S.R.O, Digiteq automotive. *Modular FrameGrabber (MGB)* [online]. Digiteq automotive s.r.o, 2024 [visited on 2024-05-05]. Available from: https://www.digiteqautomotive.com/sites/default/files/2020-08/_DSC1645__jpg__2500px_0-min.jpg.

[30] SALEM, Patrick. *Understanding ASIL Decomposition for Functional Safety in Automotive Applications* [online]. 2024. [visited on 2024-04-15]. Available from: https://www.linkedin.com/pulse/understanding-asil-decomposition-functional-safety-automotive-salem-bbcwc/.

[31] SAXENA, ANSHUL. *Everything You Need to Know About In-Vehicle Infotainment Systems* [online]. 2023. [visited on 2023-11-14]. Available from: https://www.einfochips.com/blog/everything-you-need-to-know-about-in-vehicle-infotainment-system/.

[32] SHENK, Geoffrey. *Automotive Infotainment System Testing Automation* [online]. Functionize Inc., 2018 [visited on 2024-05-05]. Available from: https://www.functionize.com/blog/testing-automation-for-infotainment-systems.

[33] SHIN, Yeonghun et al. Digital Forensic Case Studies for In-Vehicle Infotainment Systems Using Android Auto and Apple CarPlay. *Sensors (Basel, Switzerland)*. 2022, vol. 22. Available from DOI: 10.3390/s22197196.

[34] *ŠKODA presents new Digital Assistant: "Okay, Laura!"* [online]. 2023. [visited on 2023-12-06]. Available from: https://www.skoda-storyboard.com/en/press-releases/skoda-presents-new-digital-assistant-okay-laura/.

[35] SMITH, Grant Maloy. *What is Can bus (controller area network)* [online]. 2024. [visited on 2024-05-05]. Available from: https://dewesoft.com/blog/what-is-can-bus.

[36] STOCKINTHECHANNEL. *APC SMC1000IC uninterruptible power supply* [online]. 2023. [visited on 2024-05-05]. Available from: https://www.stockinthechannel.co.uk/Product/APC-SMC1000IC-uninterruptible-power-supply-UPS-Line-Interactive-1000-VA-600-W-8-AC-outlet-s-/42348216.

[37] SYNOPSYS, Inc. *What is ASIL?* [online]. 2024. [visited on 2024-04-15]. Available from: https://www.synopsys.com/automotive/what-is-asil.html.

[38] *Tcl - Arrays* [https://www.tutorialspoint.com/tcl-tk/tcl_arrays.htm]. [N.d.]. [Accessed 31-03-2024].

[39] *Tcl - Lists* [https://www.tutorialspoint.com/tcl-tk/tcl_lists.htm]. [N.d.]. [Accessed 31-03-2024].

[40] *Tcl - Namespaces* [https://www.tutorialspoint.com/tcl-tk/tcl_namespaces.htm]. [N.d.]. [Accessed 31-03-2024].

[41] *Tcl - Procedures* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://www.tutorialspoint.com/tcl-tk/tcl_procedures.htm.

[42] *Tcl Built-In Commands* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://wiki.tcl-lang.org/page/switch.

[43] TEAM, Tcl Core. *Welcome to the TCLER's wiki!* [online]. 2018. [visited on 2024-05-05]. Available from: https://wiki.tcl-lang.org/.

[44] TME.EU. *WDR series 12 V power supply* [online]. 2024. [visited on 2024-05-05]. Available from: https://www.tme.eu/en/news/library-articles/page/52591/din-rail-mounted-power-supply-modules-from-mean-well/.

[45] TUTORIALSPOINT. *TCL - logical operators* [online]. [N.d.]. [visited on 2024-05-05]. Available from: https://www.tutorialspoint.com/tcl-tk/tcl_logical_operators.htm.

[46] VAIBHAV. *Automotive Infotainment Testing Best Practices | IVI System — embitel.com* [https://www.embitel.com/blog/embedded-blog/infotainment-testing-success-mantras-that-your-automotive-development-team-should-ace/]. 2019. [Accessed 15-01-2024].

[47] WIKIPEDIA. *Electronic Control Unit — Wikipedia, The Free Encyclopedia* [http://cs.wikipedia.org/w/index.php?title=Electronic%20Control%20Unit&oldid=23626234]. 2024. [Online; accessed 25-April-2024].

[48] YIN, Ning. *Automated Testing for Automotive Infotainment Systems*. 2018. MA thesis. Chalmers University of Technology and University of Gothenburg.

[49] ZAMAN, Najamuz. *Automotive Electronics Design Fundamentals*. Springer, 2015.

A Appendices

A.1 Tcl Syntax

The syntax of Tcl is designed to be simple and expressive, enabling rapid script development and execution. This section explores the essential elements of Tcl syntax, including its command structure, variable usage, control flow constructs, and procedures, providing a solid foundation for effective scripting.

Comments

In Tcl, comments are utilised to include explanatory remarks in the code, enhancing its readability and maintainability. Single-line comments begin with the # symbol and continue until the end of the line. They can be placed on their own line or at the end of a line of code[43].

When adding a comment after a command on the same line, it is essential to separate the command and the comment with a semicolon (;) followed by the # symbol. This ensures that the Tcl interpreter correctly interprets the comment. Here is an example of how to use single-line comments:

```
# This is a standalone comment
set speed 100 ;# This comment follows a command
```

Listing A.1: Comments example

Command Structure

At the heart of Tcl is its minimalist yet powerful command syntax. Each command is a concise line of instructions, beginning with a command name and followed by its associated arguments. These arguments are separated by spaces and can be as few as none or as many as the task requires. A command ends with a new line or a semicolon, whichever suits the programmer's flow.

By way of illustration, the anatomy of a standard Tcl command is shown below:

```
commandName argument1 argument2 ... argumentN
```

Listing A.2: Command example

This simplicity ensures that commands are both readable and easy to write. For example, a code to print *"Hello Tcl World"* is as simple as:

```
puts "Hello Tcl World"
```

Listing A.3: Command to print "Hello Tcl World" in Tcl

It also uses the dollar sign ($) to denote variables. When a command is executed, variable names prefixed with $ are replaced with their corresponding values before the command is executed. This substitution mechanism is an integral part of dynamic command generation and execution[43].

```
#Example 1
set brand "Škoda"
puts "Vehicle Brand: $brand"
# Output: Vehicle Brand: Škoda

#Example 2
set speed 60
puts "Current Speed: $speed kmph"
# Output: Current Speed: 60 kmph
```

Listing A.4: Variable substitution

Tcl's command substitution feature allows for dynamic computation and interaction within scripts, making it a crucial tool for complex operations. By enclosing commands within square brackets ([]), Tcl first evaluates the enclosed command and then substitutes its result into the outer command. This mechanism is especially useful in scenarios that require data manipulation based on the outcome of embedded commands[43].

```
set radius 4
set area [expr 3.14 * $radius * $radius]
puts "Area of the circle: $area"
```

Listing A.5: Example of command substitution

In the provided example, the '**expr**' command calculates the area of a circle using the radius stored in the '*radius*' variable. The result is assigned to the '*area*' variable, which is then printed to the standard output using the '*puts*' command within square brackets. This fragment illustrates how command substitution can facilitate the execution of calculations or operations that depend on the results of other commands, seamlessly integrating them into the flow of the script.

The unset command is used to delete variables, removing them from the script's environment. When a variable is unset, it no longer exists in the namespace, and

attempting to access it afterward will result in an error unless it is redefined. This command is particularly useful for freeing up memory or ensuring that outdated or unnecessary data does not linger in the script, potentially causing incorrect behavior or conflicts[43].

```
1 unset radius
```

Listing A.6: Example of removing variable

Mathematical Operations

Although Tcl is primarily known for its string manipulation capabilities, it also provides support for mathematical operations. This makes it a versatile tool for applications requiring numerical computations, ranging from simple arithmetic to complex mathematical expressions. The language's ability to handle both string and numerical data types makes it a valuable asset for developers.

In Tcl, mathematical operations are carried out using the expr command. This command evaluates an expression and returns its value. The basic syntax is:

```
1 set result [expr {operation}]
```

Listing A.7: Example of expr command in Tcl

where operation can be any arithmetic calculation, such as addition (+), subtraction (-), multiplication (*), division (/), and more complex mathematical functions. Tcl supports a wide range of mathematical functions, including trigonometric, logarithmic, and exponential functions[43].

```
# Assume a and b are predefined variables with some values
set a 10
set b 5

set addition [expr {$a + $b}]
puts "Addition: $addition"  ;# Result: 15

set subtraction [expr {$a - $b}]
puts "Subtraction: $subtraction"  ;# Result: 5

set multiplication [expr {$a * $b}]
puts "Multiplication: $multiplication"  ;# Result: 50

set division [expr {$a / $b}]
puts "Division: $division"  ;# Result: 2

set modulus [expr {$a % $b}]
puts "Modulus: $modulus"  ;# Result: 0
```

Listing A.8: Implementing basic arithmetic operations

To illustrate the use of mathematical operations in Tcl in the context of automated testing in automotive, consider a scenario where we need to test the volume control functionality of an infotainment system. The system adjusts the volume based on the speed of the vehicle, increasing the volume as the speed increases. We can simulate this behaviour and calculate the expected volume level at different speeds.

```
# Calculate volume level based on speed
set baseVolume 20 ;# Base volume level at 0 km/h
set speed 80 ;# Current speed in km/h
set volumeAdjustmentFactor 0.1 ;# Increase per km/h
set expectedVolume [expr {$baseVolume + ($speed *
    $volumeAdjustmentFactor)}]
puts "Expected volume at $speed km/h: $expectedVolume"
```

Listing A.9: Simulating volume adjustment based on vehicle speed

Table A.1: Overview of fundamental operations in Tcl

Function	Syntax	Description
Addition	+	Adds two numbers
Subtraction	-	Subtracts the second number from the first
Multiplication	*	Multiplies two numbers
Division	/	Divides the first number by the second
Modulus	%	Returns the remainder of division
Increment	++	Increases a variable's value by 1
Decrement	--	Decreases a variable's value by 1

Logical Operations

Logical operations are crucial in programming languages as they allow for decision-making based on specific conditions. In Tcl, logical operations are primarily used in conditional statements and loops to control the flow of execution Tcl supports the fundamental logical operators found in many programming languages: **&&** (logical AND), || (logical OR), and **!** (logical NOT). These operators are utilised to combine or invert boolean expressions that evaluate to either true or false[45].

- The logical AND operator (&&) returns true only if both operands are true.
- The logical OR operator (||) returns true if at least one of the operands is true.
- The logical NOT operator (!) inverts the truth value of its operand.

Logical operators in Tcl are used within expressions, typically inside if statements, while loops, or for loops. Here's the basic syntax:

```
if {$condition1 && $condition2} {
    ;# Code to execute if both condition1 and condition2 are true
}

if {$condition1 || $condition2} {
    ;# Code to execute if either condition1 or condition2 is true
}

if {!$condition} {
    ;# Code to execute if condition is not true
}
```

Listing A.10: Implementing conditional logic

Bitwise Operations

Bitwise operations are crucial for low-level programming tasks, such as manipulating data at the binary level. In Tcl, bitwise operations enable the manipulation of bits within integer values, allowing for efficient data processing and control.

Tcl supports several bitwise operators for performing operations on integer operands at the bit level. These operators include:

- Bitwise AND (&): Performs a logical AND operation on each pair of bits in two integers.
- Bitwise OR (|): Performs a logical OR operation on each pair of bits in two integers.
- Bitwise XOR (^): Performs a logical XOR (exclusive OR) operation on each pair of bits in two integers.
- Bitwise NOT (~): Performs a logical NOT operation, inverting each bit in an integer.
- Left Shift (<<): Shifts the bits of an integer to the left by a specified number of positions.
- Right Shift (>>): Shifts the bits of an integer to the right by a specified number of positions[26].

Bitwise operators are used within the expr command in Tcl. Here's how you can use these operators:

```
set a 12 ;# Binary: 1100
set b 5  ;# Binary: 0101

# Bitwise AND
set andResult [expr {$a & $b}] ;# Result is 4 (0100)

# Bitwise OR
set orResult [expr {$a | $b}]  ;# Result is 13 (1101)

# Bitwise XOR
set xorResult [expr {$a ^ $b}] ;# Result is 9 (1001)

# Bitwise NOT
set notResult [expr {~$a}]     ;# Result depends on the system's word
    size

# Left Shift
set leftShift [expr {$a << 2}] ;# Result is 48 (110000)

# Right Shift
set rightShift [expr {$a >> 2}] ;# Result is 3 (0011)
```

Listing A.11: Example of using bitwise operators

Relational Operations

Relational operations in Tcl are utilised to compare values, which is fundamental in control flow structures such as if-conditions and loops. This section covers relational operators and their usage in Tcl scripts.

Tcl provides standard relational operators to compare numerical values and strings:

- Equal (==): Returns true if two operands are equal.
- Not Equal (!=): Returns true if two operands are not equal.
- Greater Than (>): Returns true if the left operand is greater than the right operand.
- Less Than (<): Returns true if the left operand is less than the right operand.
- Greater Than or Equal To (>=): Returns true if the left operand is greater than or equal to the right operand.

- Less Than or Equal To (<=): Returns true if the left operand is less than or equal to the right operand[27].

Relational operations are commonly used in conditional statements to make decisions:

```
set speed 60

if {$speed > 55} {
    puts "Exceeding speed limit!"
}

set temperature 35

if {$temperature >= 30} {
    puts "High temperature warning!"
}
```

Listing A.12: Example of implementing relational operations

The Tcl programming language features a straightforward syntax that emphasises ease of use and flexibility. In Tcl, each command is comprised of a command name followed by arguments separated by whitespace. The language supports various types of data without the need for specific declarations, making it highly adaptable for different scripting scenarios.

Key aspects of Tcl syntax include:

- **Command Structure:** Commands are the fundamental building blocks in Tcl, where a typical command can include the command name and various arguments. The end of a command is denoted by a newline or a semicolon.
- **Variable and Command Substitution:** Tcl allows for dynamic content within scripts through variable and command substitution. Variables are prefixed with **$** to denote substitution, while square brackets [] are used to execute commands within commands.

For those seeking a more detailed understanding of Tcl's syntactic elements and a more comprehensive examination of its application, further coverage can be found in Appendix A.1 of this document. This appendix not only provides a more in-depth analysis of the nuances of Tcl syntax but also offers extensive discussions and examples of Tcl's mathematical, logical, bitwise, and relational operations. These sections offer insights into how Tcl manages complex scripting tasks effectively.

A.1.1 Loops in Tcl

Tcl loop constructs, including '**for**', '**foreach**' and '**while**', are crucial for automating testing in IVI systems. These loops enable the repeated execution of a code block, which is essential for executing complex functions.

'For' Loop

The '**for**' loop in Tcl is similar to its counterpart in C and other programming languages. It is typically used to iterate over a sequence of numbers, making it suitable for scenarios where the exact number of iterations is known in advance. This could be particularly useful for testing a sequence of input values for an infotainment system function.

```
for {initialization} {condition} {iteration} {
    # Code block to be executed
}
```

Listing A.13: Template of a Tcl 'for' loop

- **initialisation:** This step is executed before the loop starts. It is commonly used to initialize a counter variable, but it can also include any Tcl command.
- **condition:** Before each iteration of the loop, this expression is evaluated. If it is true (non-zero), the loop continues. If it is false (zero), the loop terminates.
- **iteration:** At the end of each loop iteration, this step is executed. It is typically used to increment or decrement a counter variable, but it can execute any Tcl command.

For example, to print numbers from 1 to 10 in Tcl using a '**for**' loop, you can write:

```
for {set i 1} {$i <= 10} {incr i} {
    puts $i
}
```

Listing A.14: Example of 'for' loop in Tcl

This loop initializes a variable '**i**' to 1, continues to execute as long as '**i**' is less than or equal to 10, and increments '**i**' by 1 in each iteration. Inside the loop, the 'puts' command prints the current value of '**i**' to the standard output.

'Foreach' Loop

The '**foreach**' loop in Tcl is a powerful construct designed for iterating over lists and arrays, making it an essential tool for scenarios requiring sequential processing of elements. Unlike the for loop, which is generally used for executing a block of code a specific number of times based on a counter, the '**foreach**' loop simplifies the iteration over items in a collection without manual index management. This feature can greatly improve the readability and efficiency of code, particularly when working with data collections.

The basic syntax of the '**foreach**' loop in Tcl is as follows:

```
foreach varName list {
    ;# Code block to execute for each element in the list
}
```

Listing A.15: Template of a Tcl 'foreach' loop

- **varName:** A variable that is set to each element of the list in turn.
- **list:** A list of elements that will be iterated over.

The Tcl '**foreach**' loop is known for its clarity, convenience, and flexibility, making it the preferred choice for iterating over collections of data. One of its primary advantages is the enhanced clarity it brings to the code. By explicitly indicating that the operation involves processing each item in a *list* or *array*, it makes the developer's intention clear, improving code readability. Clarity is particularly important in complex scripts where understanding the flow of data is crucial.

Additionally, the convenience offered by the '**foreach**' loop cannot be overstated. It eliminates the cumbersome need for manual index management, which is prone to errors. This feature ensures that each element in the collection is processed without the need for explicit tracking of indices, thereby simplifying the coding process.

Flexibility is another significant benefit of the '**foreach**' loop. Tcl is proficient in managing not only basic lists but also associative arrays, referred to as *dictionaries*. This feature enables the iteration over key-value pairs in dictionaries, making complex data manipulation tasks effortless. Furthermore, Tcl's '**foreach**' loop can simultaneously iterate over multiple lists, which is ideal for parallel processing of related datasets[16].

```
set carModels {"Audi A4" "BMW 3 Series" "Mercedes C-Class"}
set infotainmentVersions {"MMI 10.1" "iDrive 7.0" "MBUX 2020"}
foreach carModel $carModels infotainmentVersion $infotainmentVersions {
    puts "$carModel is equipped with infotainment system version:
    $infotainmentVersion"
}
```

Listing A.16: Example of 'foreach' loop in Tcl

The output of the previous code will be displayed as follows:

```
Audi A4 is equipped with infotainment system version: MMI 10.1
BMW 3 Series is equipped with infotainment system version: iDrive 7.0
Mercedes C-Class is equipped with infotainment system version: MBUX
    2020
```

Listing A.17: Output of 'foreach' example

'While' Loop

The '**while**' loop is a crucial control structure in Tcl that enables the execution of a block of code repeatedly as long as a specified condition remains true. This type of loop is particularly useful for situations where the number of iterations is not known before the loop starts. It is ideal for tasks such as reading data until an end-of-file marker is reached or for polling a device status until a certain state is detected.

The basic syntax of the while loop in Tcl is as follows:

```
while {condition} {
    ;# Code block to be executed as long as condition is true
}
```

Listing A.18: Template of a Tcl 'while' loop

- **condition:** An expression that is evaluated before each iteration of the loop. If the condition evaluates to true (non-zero), the loop continues with another iteration. If the condition evaluates to false (zero), the loop terminates, and execution continues with the next statement following the loop[16].

In the following excerpt, we explore the use of a '**while**' loop, a fundamental control structure in the Tcl programming language, through a whimsical yet instructive example that humorously mirrors the iterative process of writing a thesis itself.

```
# Initialize the thesis progress
set thesisProgress 0
set totalRequiredPages 80

# Loop until the thesis is complete
while {$thesisProgress < $totalRequiredPages} {
    ;# Print a message about the current state
    puts "You've completed $thesisProgress pages of your Bachelor
   thesis. Keep going!"

    ;# Add some humour with coffee breaks
    if {$thesisProgress % 10 == 0} {
        puts "Time for a coffee break! You've earned it!"
    }

    ;# Add a funny message for the halfway point
    if {$thesisProgress == $totalRequiredPages / 2} {
        puts "Halfway there! Imagine the graduation scarf drapping over
    your shoulders."
    }

    ;# Sleep for a second to simulate work being done
    after 1000

    ;# Increment the thesis progress
    incr thesisProgress
}

# Print a message once the thesis is complete
puts "Congratulations! Your Bachelor thesis is finally complete. Time
   to celebrate!"
```

Listing A.19: Programming your way to graduation: A Bachelor's thesis progress simulator in Tcl

This Tcl code represents a light-hearted simulation of writing a bachelor's thesis, tracking progress in increments towards an 80-page goal. As pages are 'completed', encouraging messages encourage the user to keep going, interspersed with cues for coffee breaks after every tenth page and a special note at the halfway point. At the end, a congratulatory message marks the completion of the simulated dissertation journey.

The ’**while**’ loop’s versatility makes it suitable for a wide range of applications in Tcl scripting, including:

- **Data Processing:** Iterating over a data stream or a file line by line until no more data is available.
- **Timing Loops:** Implementing delays or waiting for a specific condition to be met, useful in polling operations or timeouts.
- **Interactive Prompts:** Repeatedly asking for user input until a valid response is provided or the user chooses to exit.

When using ’**while**’ loops, it’s important to ensure that the loop condition will eventually become false; otherwise the loop could become an infinite loop, causing the script to hang or consume excessive resources. Careful management of the condition and loop variables is essential to avoid such problems[16].

Understanding the ’switch’ Command in Tcl

The ’**switch**’ command in Tcl provides a powerful mechanism for toggling program control between a number of options based on the value of an expression. It is similar to the switch statement found in many other programming languages, but comes with features that are uniquely tailored to the dynamic nature of Tcl. This control structure improves the readability and efficiency of code that requires conditional execution of multiple branches[42].

The basic syntax of the ’**switch**’ command can be expressed as follows:

```
switch options? string pattern body ... ?default body?
```

Listing A.20: Template of a Tcl ’switch’ command[42]

- **string:** The string against which patterns are matched.
- **pattern:** The pattern to match against the string. Patterns can be literals, glob-style patterns, or regular expressions, depending on the options used.
- **body:** The script to execute when a match is found.

The Tcl ’**switch**’ command is a versatile conditional control structure that enables the execution of different code blocks based on the match between a value and a set of patterns. Unlike the simpler if-elseif-else construct, ’**switch**’ can significantly simplify code, especially when dealing with complex conditional logic. It

stands out for its ability to handle not only exact matches but also more complex pattern matching scenarios.

The 'switch' command enables three main modes of pattern matching, making it useful for various use cases:

- **Exact Matching:** By default, the '**switch**' function performs exact matching by comparing the given string against a series of patterns to find an exact match. This mode is straightforward and covers many basic use cases.
- **Glob-Style Matching:** When the '**-glob**' option is activated, patterns can include glob-style wildcards (such as * and ?). This mode is especially useful for matching strings against patterns that follow a predictable format but may contain variable parts.
- **Regular Expression Matching:** The '**-regexp**' option allows for the interpretation of patterns as regular expressions, providing the greatest flexibility and power for matching complex string patterns. This mode is particularly useful for scenarios that require advanced pattern-matching capabilities[42].

```
set filename "archive.zip"

switch -glob $filename {
    *.txt {
        puts "Processing a text file."
    }
    *.jpg|*.png {
        puts "Processing an image file."
    }
    *.zip {
        puts "Processing a compressed file."
    }
    default {
        puts "Unsupported file type."
    }
}
```

Listing A.21: Practical example of a Tcl 'switch' command

In this example, the -glob option allows the use of wildcard patterns to match file extensions, providing a concise method to route processing logic based on the file type.

A.1.2 Mastering Functions in Tcl using 'proc'

Functions, which are referred to as *procedures* in Tcl, are essential building blocks that enable the encapsulation and reuse of code within scripts and applications. Procedures in Tcl are defined using the '**proc**' command and allow for the creation of complex, modular, and maintainable codebases. This subsection explores the creation, usage, and advanced features of procedures in Tcl, providing programmers with the knowledge to utilize their full potential.

A procedure is defined using the '**proc**' command, followed by the procedure name, a list of parameters, and the procedure body. The syntax is as follows:

```
proc procedureName {parameterList} {
    # Procedure body
}
```

Listing A.22: Template of a Tcl 'proc' command

- **procedureName:** The name of the procedure.
- **parameterList:** A list of parameters the procedure accepts, enclosed in braces **{}**.
- **Procedure body:** The Tcl code to execute when the procedure is called[41].

Parameters can be defined to accept default values, making them optional during calls. This is achieved by specifying the parameter name followed by the default value in the parameter list.

To call a procedure, simply use its name followed by any required arguments:

```
procedureName arg1 arg2
```

Listing A.23: Template of calling procedures

Arguments are passed to the procedure in the order they are listed in the parameter list[41].

Example: A Simple Procedure

Consider a procedure that greets a user:

```
proc greet {name} {
    puts "Hello, $name!"
}

greet "Ekaterina"
```

Listing A.24: Example of a simple procedure

This greeting procedure takes a single parameter, '*name*', and uses it within a '*puts*' command to print a greeting. When calling the 'greet' function with the argument '*Ekaterina*', it will print "*Hello, Ekaterina!*"

Working with Return Values

Procedures in Tcl return the result of the last command executed in their body by default. To return a specific value, use the '**return**' command[41]:

```
proc sum {a b} {
    return [expr {$a + $b}]
}

set result [sum 3 12]
puts $result  ;# Outputs: 15
```

Listing A.25: Example of a simple procedure with 'return' command

A.1.3 Harnessing the Power of Dictionaries for Efficient Data Management in Tcl

In Tcl, dictionaries are one of the powerful data structures designed to store and manage collections of elements in key-value pairs. This structure is particularly useful for organising related data, making it easily accessible by reference to a unique key.

Basics of Dictionaries

A dictionary in Tcl is an unordered collection of key-value pairs, where each key is unique. Dictionaries are ideal for storing associative arrays, where each key maps to a value. They are created using the '**dict create**' command, followed by key and value pairs. Here's a simple example[8]:

```
set myDict [dict create key1 "value1" key2 "value2"]
```

Listing A.26: Example of creating a dictionary

To access a value in a dictionary, the **'dict get'** command is used, specifying the dictionary and the key:

```
puts [dict get $myDict key1] ;# Outputs: value1
```

Listing A.27: Example to access a value in a dictionaryl

Modifying Dictionaries

Tcl offers several commands for modifying dictionaries, including '**dict set**' for adding or updating key-value pairs and '**dict unset**' for removing them. Modifying a dictionary does not change the original; instead, it returns a new modified version[8].

```
# Adding/updating a key-value pair
set myDict [dict set myDict key3 "value3"]

# Removing a key-value pair
set myDict [dict unset myDict key1]
```

Listing A.28: Example of 'dict set' and 'dict unset' commands

Iterating Over Dictionaries

To iterate over a dictionary, the '**dict for**' command is used. It allows you to loop through each key-value pair, performing operations as needed.

```
dict for {key value} $myDict {
    puts "Key: $key, Value: $value"
}
```

Listing A.29: Example of 'dict for' command

Advanced Features

- **Nested Dictionaries:** Tcl supports nesting dictionaries within dictionaries, enabling the representation of complex data structures.
- **Dictionary Keys:** While typically strings, dictionary keys can be any value, offering flexibility in how data is structured.
- **Efficiency:** Dictionaries are implemented efficiently, providing fast access to data, which is crucial for performance-sensitive applications.

The table below summarises the various **'dict' command options** available in Tcl, along with a brief description for each. This table provides a concise overview of the functionality that the **'dict'** command offers for manipulating dictionaries in Tcl.

Table A.2: Summary of Tcl 'dict' command options[8]

Command Option	Description
dict create	Creates a new dictionary with optional key-value pairs.
dict get	Retrieves the value for a given key from the dictionary.
dict set	Sets the value for a given key, creating the key if necessary.
dict unset	Removes a key (and its value) from the dictionary.
dict update	Temporarily updates keys with variables for a script block.
dict append	Appends string values to the value of a key.
dict lappend	Appends list elements to a list in the dictionary.
dict replace	Replaces or adds key-value pairs in the dictionary.
dict remove	Removes one or more keys and their values from the dictionary.
dict merge	Merges two or more dictionaries, with later keys overriding.
dict incr	Increments the value of a key by a given amount.
dict with	Updates the dictionary with variables scoped within a script.
dict for	Iterates over the dictionary, assigning keys and values to variables in a loop.
dict keys	Returns a list of all keys in the dictionary.
dict values	Returns a list of all values in the dictionary.
dict size	Returns the number of key-value pairs in the dictionary.
dict exists	Checks if a key exists in the dictionary.
dict info	Returns a human-readable string with information about the internal representation of the dictionary.
dict map	Transforms the dictionary according to the script, returning a new dictionary.
dict filter	Filters the dictionary based on keys, values, or script criteria, returning a new dictionary.

A.1.4 Using Namespaces in Tcl for Modular Programming

In Tcl, '**namespaces**' are a fundamental concept used to encapsulate and organise code into distinct modules or packages. They act as containers for grouping related commands, variables, and other '**namespaces**', thus avoiding name collisions and improving code reusability and maintainability. By using '**namespaces**', developers can create modular applications where components can be developed, tested and deployed independently[40].

Creating a 'namespace' in Tcl is straightforward and can be achieved using the 'namespace' command. Here's a simple example:

```
namespace eval MyNamespace {
    # Define variables and procedures within the namespace
    variable myVar "Hello, Namespace"
    proc myProc {} {
        return "This is a procedure in MyNamespace"
    }
}
```

Listing A.30: Example of creating a namespace in Tcl

In this example, the namespace '**MyNamespace**' contains a variable called '**myVar**' and a procedure called '**myProc**'. To access these elements from outside the namespace, you must use their fully qualified names, which include the namespace name followed by two colons (**::**) and the element name. For instance, '**MyNamespace::myVar**'.

Importing Namespaces

As software systems grow in complexity, modularisation becomes increasingly important. Modularisation not only helps organise code but also enables code reuse across different modules or projects. However, accessing commands or variables defined in other '**namespaces**' can be cumbersome, as it requires prefixing the full '**namespace**' path to each call. Tcl's importing mechanism simplifies this access, making code more readable and maintainable.

Tcl offers the '**namespace import**' command, which allows importing commands from one namespace to the current or another specified namespace. This enables invoking commands without the need to prefix them with the namespace path[40].

```
# Assume 'myUtilities' namespace defines a procedure 'performCalculation'
namespace import myUtilities::performCalculation

# Now 'performCalculation' can be directly called
performCalculation args
```

Listing A.31: Example of importing a namespace

A.1.5 Leveraging Lists in Tcl

Lists are a frequently used data structure in Tcl, providing a simple and powerful way to store and manipulate ordered collections of items. The syntax is straightforward, and the built-in commands for list manipulation are extensive, making them an essential tool in Tcl programming.

Lists can be created in Tcl either by directly specifying the elements within braces or by using the '**list**' command. This flexibility allows for easy list construction and modification[39].

```
# Creating a list with braces
set myTestCases {TestCase1 TestCase2 TestCase3}

# Creating a list with the list command
set myParameters [list Parameter1 Parameter2 Parameter3]
```

Listing A.32: Examples of creating a list

Tcl provides the '**lindex**' command to access elements at specific indices in a list. Tcl lists are zero-indexed, making the first element accessible at index 0.

```
# Accessing the first testcase
set firstTestCase [lindex $myTestCases 0]
```

Listing A.33: Example of accessing element in list

Tcl provides a comprehensive range of commands for managing lists, such as **linsert**, **lreplace**, **lappend**, and **lremove**, catering to various needs for dynamic list management[39].

```
# Appending a new TestCase
lappend myTestCases TestCase4

# Inserting a new parameter at the beginning
set myParameters [linsert $myParameters 0 NewParameter1]
```

Listing A.34: Example of lappend and linsert commands

A.1.6 Arrays

Tcl '**arrays**', also known as associative arrays, allow for the association of keys with values, providing a more convenient way to access data using meaningful identifiers instead of numeric indices. This feature is particularly useful in automated testing scenarios, where parameters and results can be directly linked to specific TestCases or configurations.

The creation and modification of arrays in Tcl are straightforward. Using the '**set**' command with the array name and key, one can easily assign values to specific keys within an array[38].

```
# Assigning a value to a key in an array
set testResults("TestCase1") "Pass"
set testResults("TestCase2") "Fail"
```

Listing A.35: Initializing test outcomes in an associative array

To obtain a value from an array, you need to specify the array name and key. The '**array get**' command can be used to retrieve all key-value pairs, which makes it easy to iterate over the contents of an array[38].

```
# Retrieving a value using its key
set result1 $testResults("TestCase1")

# Getting all key-value pairs from an array
array get testResults
```

Listing A.36: Example of accessing array elements

A.2 ImageMagick: A Powerful Tool for Image Processing in Automated Testing

ImageMagick is a versatile, open-source software suite that is widely recognised for its ability to create, edit, compose, or convert bitmap images. It supports over 200 image formats, including popular ones such as JPEG, PNG, TIFF, and GIF, making it an indispensable tool in software testing where image manipulation and analysis are crucial[18].

Functionality and Features

ImageMagick offers a broad range of functionalities that are particularly useful in automated testing of IVI systems:

- **Conversion and Transformation:** ImageMagick can convert images between formats, resize, rotate, apply various effects, and adjust image colours, which is essential for preparing test artefacts and simulating different screen scenarios.
- **Image Comparison:** One of its most powerful features is the ability to compare images. This is invaluable in regression testing, where verifying the consistency of UI elements after updates or changes is necessary. ImageMagick

can highlight differences between images down to the pixel level, providing a visual aid to identify unexpected changes.

- **Text and Graphics Handling:** It can annotate images with text or overlay graphics. This is particularly useful for adding labels or instructions directly onto test images or for visualising test results[18].

Application in Automated Testing

In the context of automated testing for infotainment systems, ImageMagick is used to automate several crucial tasks:

1. **Automated Screenshots Verification:** ImageMagick can automatically process screenshots taken during tests to verify the correctness of the graphical user interface against a baseline. This process is vital for ensuring that all visual elements are displayed correctly across different system versions or after software updates.

2. **Batch Processing:** The tool can handle batch processing of images, which enables the testing framework to process large numbers of screenshots in an automated and efficient manner. This capability significantly reduces manual effort and speeds up the testing cycle.

Benefits in Infotainment Testing

Utilizing ImageMagick in infotainment system testing brings several benefits:

- **Enhanced Accuracy:** Automated image comparison helps in detecting UI discrepancies that might be overlooked during manual testing.

- **Efficiency:** Automates repetitive tasks such as image conversions and adjustments, allowing testers to focus on more complex test scenarios.

- **Scalability:** Supports handling large datasets of images, which is typical in extensive testing phases of infotainment systems where numerous screen states need to be validated.

B Appendices

B.1 Attached Files

readme.txt	Description of attached files
Enyaq_Report	Directory with sample report
TCs_ICAS3_SK_MEB13_EU_LHD	Directory with TestCases
source	
|_ Bachelor Thesis Sojka.zip	Zip file of LATEX project
|_ Bachelor_Thesis_Sojka.tex	Text of the work in LATEX format