CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Requirement-centric Framework to Build Tool Integration for Traceability and Consistency

Master's thesis in Computer science and engineering

Dongying Cao

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

A Requirement-centric Framework to Build Tool Integration for Traceability and Consistency

Dongying Cao

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

A Requirement-centric Framework to Build Tool Integration for Traceability and Consistency
DONGYING CAO

Supervisor: Eric Knauss, Department of Computer Science and Engineering
Advisor: Daniel Larsson, Shiva Kumar Gone, Volvo Car
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 736582601

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2024

A Requirement-centric Framework to Build Tool Integration for Traceability and Consistency
Dongying Cao
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Many automotive companies use the ISO 26262 and Automotive SPICE (ASPICE) standards to develop software. These standards highlight the importance of traceability and consistency. However, in practice, different companies use various software tools for different tasks such as software testing and requirement management. Current methods to achieve traceability often depend heavily on specific suppliers or tools. Some methods require a lot of manual work, like maintaining Excel sheets, while others automate the process but are not efficient enough and stick to certain standards. The issue of ensuring consistency has received less attention. A more generic and flexible approach is needed to establish traceability and identify inconsistencies using data, not relying on specific tools.

This thesis study was conducted at Volvo Car Corporation with a software integration and release team. Not just this team, but other departments also face a challenge: there is no effective workflow to establish traceability for automated HIL (Hardware in the Loop) testing. Requirements are managed in a tool called CarWeaver, while test cases are executed either automatically in dSPACE AutomationDesk or manually in dSPACE ControlDesk. Currently, traceability and visualization are only managed for manual testing using Excel sheets. As automated HIL testing becomes more common, a method to also report the status of requirements is needed. Furthermore, the consistency required by ASPICE is lacking in the current approach.

This thesis proposes a requirement-focused framework to create a tool integration framework for establishing traceability and checking consistency. This framework does not depend on any specific software vendor in the automotive industry. It involves collecting data on test results and requirements from various software tools and then processing and analyzing this data centrally. The study provides a practical example, including adjusting the test workflow to capture the right data and using APIs to transform requirement data. The central processing unit uses Python scripts for data handling and Power BI for visualization.

This framework could also be useful in other fields where requirement traceability and consistency are issues, especially when different tools are used for various purposes.

Design science research is adopted as the research methodology with two cycles of iteration, each involving problem investigation, solution design, design validation, solution implementation, and implementation evaluation.

Keywords: tool integration, requirement traceability, consistency, automotive indus-

try, Automotive SPICE.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Eric Kanuss, for his invaluable guidance, patience, and expert advice throughout this project. His encouragement and support have been cornerstones of my journey. I am also immensely thankful to Jennifer Horkoff, my examiner, for her insightful comments and suggestions, which have significantly contributed to the refinement of this work.

A special thanks goes to Shiva Kumar Gone, Jesper Fritsch, and Daniel Larsson at Volvo Car Corporation. Their industry insights and expertise were vital in bridging the gap between academic research and practical application. Their contributions have been crucial to the success of this project. I am also grateful to the entire Integration and Release team and the Complete Software team at Volvo Car Corporation. Working alongside such talented groups has been both inspiring and rewarding. Their collaboration and willingness to share their knowledge and experiences have enriched this experience.

Dongying Cao, Gothenburg, 2024-06-18

Contents

1

Introduction

In today's automotive industry, software plays an increasingly critical role. The volume of software integrated into vehicles is expanding and the size reaches 100 Million lines of code [1]. At the same time, software complexity and its application in controlling safety-critical functions have surged as well [2]. During the development process, numerous artifacts are produced, such as requirements, test cases, implementation code, and so on. Therefore traceability should be managed and achieved effectively [3]. The automotive industry adheres to standards like ISO 26262 and Automotive SPICE (ASPICE) throughout the development cycle [4], which often highlight the significance of traceability and consistency. Well-established traceability facilitates the verification of requirements [5], thereby ensuring the high quality and safety of the software and promoting product delivery.

However, due to the large number of artifacts in modern automotive systems and various software in use, establishing and interpreting traceability has become a challenging task. For interpretation, visualization tools aid in presenting the complicated relationships between requirements and test cases in a more understandable way [6].

Software should be tested thoroughly before the delivery to stakeholders. Testing activities in the automotive industry often require interaction with external elements like signals. These activities are typically carried out in simulated environments to ensure accuracy and safety before they are implemented in an actual vehicle. Volvo Car Corporation employs hardware-in-the-loop (HIL) testing for this purpose, using AutomationDesk for most of the automated testing and ControlDesk for manual testing. In manual HIL testing, a process is in place to record test results and track requirement traceability using Excel sheets and manual work. However, automated HIL testing lacks a similar process. The software systems do not automatically connect, resulting in missing data. This creates a gap in traceability, leading to issues such as no visualization of requirements and no checks for inconsistencies. Also, reports from AutomationDesk do not include links between test cases and requirements. If stakeholders want to know which requirements are linked to failed test cases, they must undergo a time-consuming manual search. Therefore, it is necessary to develop a new solution to simplify access to traceability.

Due to the complexity of software, automated testing has become essential [7]. Volvo Car Corporation is shifting its focus from manual to automated HIL testing. The lack of traceability in automated testing is becoming a major problem. This study uses design science research methodology to explore traceability challenges in auto-

mated HIL testing at Volvo Car Corporation. It suggests a framework for integrating tools to establish requirement traceability and ensure consistency checks, which can be applied in broader contexts. This framework could help other companies improve traceability and consistency across different software platforms, such as dSPACE AutomationDesk for automated HIL testing and SystemWeaver for managing requirements. This framework differs from existing methods for integrating tools to track requirements, such us using software product provided by particular vendors or filling Excel templates, as it places few restrictions on the software or architecture used and does not depend heavily on specific suppliers or tools. It also moves away from the widespread yet inefficient practice of using Excel to manage traceability in complex, multi-software environments.

1.1 Case Company

Volvo Car Corporation, with its research and development department headquartered in Sweden, is a leading international manufacturer of luxury vehicles. The company has adopted a hybrid agile methodology called the Volvo Cars Agile Framework (VCAF). At the heart of VCAF's organizational architecture is the Product Stream concept. A Product Stream is comprised of multiple Agile Release Trains (ARTs), which are long-term, dedicated, cross-functional teams working toward a unified goal.

This thesis is conducted within the Integration and Release (IAR) Team in the Power Electronics Software ART (PESW), which is part of the Electric Propulsion Product Stream under the Engineering department. PESW is responsible for developing optimal drive control software for electric and hybrid vehicles to ensure superior customer satisfaction. The IAR team mainly works with the integration and delivery of inverter software.

The IAR team integrates inverter software developed within the application layer of the Autosar stack as described in [8]. At this stage, automated HIL testing is performed as smoke testing, which is a quick and preliminary test of a software build, focusing on basic functionality to ensure the major components are working before deeper testing is performed. The team then delivers the integrated software to external suppliers, who build the complete inverter software. Upon receiving the finalized complete software, comprehensive manual HIL testing, known as acceptance testing, will be performed to confirm that all relevant requirements defined in a software called CarWeaver are met. The complete inverter software will then be sent to the internal stakeholders of the company.

Currently, HIL smoke testing is automated through Jenkins pipelines, running test cases with dSPACE AutomationDesk. In contrast, HIL acceptance testing is manually performed using dSPACE ControlDesk. Both processes use Excel sheets to build and visualize traceability. However, as the company shifts from manual to automated HIL testing, there is a growing need for an automated traceability system. This new system will support quicker and smoother software development and release cycles.

1.2 Purpose of the Study

The purpose of this study is to develop a method to automate traceability in complex environments where different software is used for various purposes, particularly in the automotive industry. This will be achieved by creating a framework that focuses on data integration and requirements.

The main beneficiaries are all stakeholders involved in the software integration and delivery process at Volvo Car Corporation. This includes integration and delivery engineers, developers, testers, suppliers, and project managers. The study seeks to offer a flexible and broadly applicable method that can be used by various companies across different industries to establish traceability between different systems and software. For Volvo Car Corporation, this research is expected to support more strategic decision-making and improve the efficiency of software delivery.

1.3 Problem Statement

In the automotive industry, HIL testing is typically conducted before real vehicle testing. The case company, Volvo Car Corporation, is transitioning from manual to automated HIL testing before software release. However, they face challenges due to the lack of an effective method for establishing requirement traceability in automated HIL testing. While there is a workflow for building traceability in manual HIL testing, it is unsuitable for automated testing because it requires significant manual effort. This lack of traceability has led to issues such as missing information in test reports, the inability to automatically analyze project requirements, and the absence of proper visualization to aid software release.

Many companies, including Volvo Car Corporation, struggle with traceability management and maintaining consistency among software artifacts. Factors contributing to this include distributed teams, frequent software updates, changing trace links, and varied stakeholder backgrounds. The limitations of current tool integration practices make it challenging for companies to address these issues.

Therefore, a more generic and flexible method is required to build requirement-centric traceability in a complex environment where various software interact.

1.4 Research Questions

The proposed study aims to address three research questions.

RQ1 - What is the problem with the current status of requirement traceability in the software build and automated test workflow?

This question aims to investigate the challenges and limitations regarding requirement traceability in the existing workflow faced by the case department.

RQ2 - What improvements in test reporting and visualization are sought by stakeholders?

This question aims to understand stakeholders' expectations for traceability within the test report and the following visualization of test case execution results.

RQ2.1 - How should traceability information be integrated into the HIL testing report?

This question aims to determine the specific traceability details between HIL test cases and corresponding requirements that stakeholders want to see included in the test reports. It also considers the presentation of this traceability data, including style and layout preferences.

RQ2.2 - In what way would the stakeholders prefer traceability data to get visualized?

This question aims to discover the preferred methods and formats for visualizing the execution results of automated HIL test cases.

RQ3 - How can improved traceability in test reports and visualizations enhance the efficiency of software integration and release processes?

1.5 Scope and Limitations

HIL testing at Volvo Car Corporation is categorized into four levels: Component, Domain, Complete, and Vehicle HIL testing. Discussions with the team highlighted their desire to incorporate traceability information in reports and export execution results in JSON format across all testing levels. However, given the limited timeframe of this master's thesis project, this study will concentrate on Component HIL testing.

The case company frequently updates its vehicle platforms and introduces new tools at the onset of new platform development, leading to variations in working methods. This thesis investigates a specific platform project where CarWeaver is used for managing requirement information and test case definitions. It's important to note that while previous projects may have employed different tools for requirement management, this study is limited to projects utilizing CarWeaver and does not address the transition of requirement management tools for other projects.

Moreover, adding a traceability table into HIL test reports requires modifications to existing test cases. It needs to be clarified that this study does not update test case details such as requirement IDs and test case IDs in CarWeaverthis is a responsibility that remains with the test case developers.

The framework suggested in this study can work with any software that can share its data, such as through an API or direct export. However, if the software does not have a method to export data, the framework will not be completely automated. In such cases, manual effort to export data will be necessary.

1.6 Significance of the Study

In this design science research, the author contributes to both academia and industry by proposing an innovative and automated framework centered on data and requirements for developing a tool integration framework that is independent of any specific software vendor or standard. This framework offers a more universal, adaptable, and automated solution compared to existing solutions, facilitating traceability establishment and consistency maintenance. Therefore, it is applicable not only to automotive companies but also to other industries facing similar challenges with traceability.

Volvo Car Corporation is the biggest beneficiary here since the study bridges the gap in requirement validation for future fully automated HIL testing processes. This improvement will bring smoother software delivery. The framework developed in this study can be applied across Agile Release Trains (ARTs), product streams, and departments at Volvo Car Corporation. With the tool's source code available in-house, it also serves as a benchmark for future tools designed to extract or augment data from AutomationDesk and CarWeaver for advanced analytics.

1.7 Outline

The rest of this report is structured as follows: It begins with a section on background information, which describes the main tools and techniques used by the case team. This is combined with a review of related literature that provides support for the thesis. Next, the research methodology used in this study is detailed. The findings of the two iterative phases carried out during the thesis are then presented, leading to a discussion of these results. The report concludes with a summary and perspectives on future research directions.

2

Background

This chapter introduces the key concepts, techniques, and tools relevant to the study.

2.1 CI/CD and Automated Testing

Continuous Integration (CI) is a widely established development practice in the software development industry [9]. It refers to the practice of merging all developers' working copies to a shared mainline several times a day [10]. Automated software building and testing are usually included in CI practice [11]. As summarized by Shahin et al. [12], CI reduces build and test time and increases visibility and awareness of build and test results. The subsequent delivery process benefits from CI due to the improvement of dependability and reliability. Jenkins is widely used as a CI tool in companies.

Continuous Delivery is a software development discipline where software is developed in a manner that allows it to be released to production at any time [13]. The discipline is achieved through optimization, automatization, and utilization of the build, deploy, test, and release process [14].

Both CI and CD underscore the critical role of employing automated testing. Automated testing is when software tools execute tests before it is released into production, ensuring that the software behaves as expected. General software testing is composed of five phases. The first phase, test planning, is mainly to plan all the test activities that are to be conducted in the whole testing process. The second phase is test development, where the test cases are developed to be used. Test execution is the next phase. It encompasses the execution of the test cases. The fourth phase is test reporting where the relevant bugs are reported. Test result analysis is the last stage in which the software developers analyze the defect. It can also be performed together with the client as it will help in the better understanding of what to ignore and what to fix, enhance, or modify [15].

Nowadays about 90% of all the current car innovations are based on electronics and software and the software used in a car has increased a lot reaching the size of 100 Million lines of code [1]. Therefore, automated testing is becoming increasingly important in the automotive industry to address the needs of complex software systems effectively. In the context of automotive systems, test automation consists of the following three steps: test case generation, test execution, and result analysis.

The generation step usually utilizes state charts, control flow, information flow, and MSCs (Message Sequence Charts). When the second step, test execution, is completed, the test results should be analyzed to unveil the error that caused a test to fail and discover errors that need to be fixed [7].

2.2 Requirement Traceability

Requirement traceability is defined as the ability to describe and follow the life of a requirement in both forward and backward directions (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [16].

Traceability helps trace interdependencies between different types of artifacts [17]. For example, it can tell which test cases verify a certain requirement and identify whether the requirement is issued by a person or document or a group of persons [5]. Hyun Cho discussed the growing complexity of systems and the evolving needs of customers [3], highlighting the importance of traceability in software development: traceability ensures the quality and timely delivery of systems under tightening development schedules and budgets; it also aids in making informed decisions, assessing implementation completeness, and identifying potential for reusing components, thereby facilitating more effective system development processes. Volvo Car Corporation is currently using a platform called CarWeaver for requirement management.

2.3 Tool Integration

In the automotive industry, many companies follow the international standard ISO 26262 to guarantee high-level functional safety. ISO 26262 emphasizes the importance of traceability throughout the entire development process to ensure that each safety requirement is implemented and verified.

In addition to ISO 26262, global automakers are required to apply the Automotive SPICE (Software Process Improvement and Capability Determination) framework to evaluate the development capabilities and qualities of suppliers for preventing accidents caused by errors in the electronic components [18]. It provides a set of requested practices in the development life-cycle, including system and software testing. According to the Automotive SPICE process reference model, there are five processes directly dealing with system and software testing: System Integration and Integration Test, System Qualification Test, Software Unit Verification, Software Integration and Integration Test, and Software Qualification Test [19]. ASPICE highlights the importance of bidirectional traceability and consistency between artifacts for the testing processes. Falcini et al. shortly described the base practices of the five testing processes according to [20]. Seven base practices of the software qualification test process are listed in Table 2.1. As can be seen in BP5 and BP6, bidirectional traceability needs to be built between requirements and test cases and there should be no contradiction between them.

Id.	Definition
BP1	Develop software qualification test strategy including regression test strategy consistent with the project plan and the release plan.
BP2	Develop the specification for software qualification test including test cases based on the verification criteria, according to the software test strategy to provide evidence for compliance of the integrated software with the software requirements.
BP3	Select test cases from the software test specification. The selection of test cases shall have sufficient coverage according to the software test strategy and the release plan.
BP4	Test the integrated software using the selected test cases. Record the software test results and logs.
BP5	Establish bidirectional traceability between software requirements and software qualification test cases.
BP6	Ensure consistency between software requirements and software qualification test cases.
BP7	Summarize the software qualification test results and communicate them to all affected parties.

Table 2.1: SWE.6 Software Qualification Test Process Base Practices

However, in real-life practice, achieving effective traceability management and ensuring consistency are often difficult. Some reasons can be software projects usually involve distributed teams, experience constant changes to software artifacts and trace links, and include multiple stakeholders with different backgrounds [21]. Amalfitano et al. [22] also observed insufficient support for traceability creation and management activities in the automotive industry. A significant factor is the lack of integration among the tools employed throughout the software development lifecycle. Besides, traceability relationships are still manually established and maintained sometimes. As suggested by Maro et al., the challenges for traceability, stemming from the use of diverse artifacts and tools, can potentially be resolved through the adoption of an integrated tool platform or tool integration [23].

Research into tool integration began in the 1990s [24]. It aims to link software development tools into a cohesive workflow to support the entire software lifecycle. By allowing data and information to flow smoothly between different development stages and components, toolchain integration becomes highly beneficial in complex areas like the automotive industry. Proper integration of these tools can enhance traceability and help with consistency, ensuring that the final product meets all requirements and quality standards that the project should follow.

Some research on toolchain integration within the automotive domain has been conducted based on particular projects [25] [26] [27]. The dSPACE company introduced SYNECT as a system integration tool for Model-Based Design systems [28].

It provides centralized data management for the integration of artifacts of testing and development. The tool can be easily applied to Hardware-in-the-Loop (HIL), Model-in-the-Loop (MIL), and Software-in-the-Loop (SIL) testing. However, its use is limited to companies that employ dSPACE products for simulation testing. Companies using standalone requirement management tools might not use SYNECT, or they may need significant effort and resources to integrate it. Here, a clear gap remains. The projects explored above often depend on specific tools and platforms, which constrains the generalization and applicability of the proposed solutions. Such diversity means that those solutions are custom-fit for certain situations, which is a challenge when trying to use them in different contexts.

A more commonly used software, Microsoft Excel, is adopted by some companies to establish traceability. Navas et al. reported manual efforts in a case study [29]. The case company in this study uses a requirement traceability matrix document to create and maintain traceability links of requirements and test cases. This document must be manually completed by the person in charge of testing. The manual process is tedious, and tracking artifacts becomes challenging when changes occur and the document is updated. Consistency validation was not covered in this study. Some companies have automated their use of Excel for maintaining traceability instead of building it manually, but issues remain. Excel only records the links or IDs of artifacts, without knowledge of the artifacts themselves. As a result, traceability is managed separately from the actual artifacts, increasing the risk of inconsistencies between the links and the artifacts themselves [30].

Klespitz et al. introduced a method called Augmented Lifecycle Space (ALS), building on Open Services for Lifecycle Collaboration (OSLC), to fill bidirectional traceability gaps and correct inconsistencies in either homogeneous or heterogeneous development environments [31]. In heterogeneous environments, information remains isolated and inaccessible to other tools. Conversely, in homogeneous environments, tools within the system can exchange data freely, and all artifacts are available to any tool that requires them. The core concept of ALS involves generating workflows automatically. The steps to get ALS proposed by Klespitz et al. include categorizing existing artifacts within the tool environment, exploring their relationships, and automatically generating any missing artifacts based on the model's traceability and consistency requirements.

To illustrate this approach further, Klespitz et al. provided a practical example where a heterogeneous system was established using IBM Rational DOORS for requirements management and Atlassian JIRA to manage test information, including test cases and results [32]. Integration between these tools was accomplished through a simple CSV file exchange. Initially, DOORS generates a CSV file containing references to relevant test cases. This file is then updated by JIRA, which appends test results and aligns them with the required test identifiers. The updated CSV file is sent back to DOORS for comprehensive analysis.

This method has shown its ability to improve traceability and consistency through practical demonstration, yet its broad applicability is limited. The success of the ALS heavily depends on the adoption of the OSLC standard across all utilized tools.

Challenges arise when some tools lack OSLC support or when there are inconsistencies in their implementation. Additionally, the ALS approach requires significant effort and expertise to integrate various tools with diverse functionalities, interfaces, and data formats, particularly in heterogeneous environments. The need to develop OSLC adapters for tools that do not support the standard natively adds to this complexity. Moreover, the ALS should ideally identify inconsistencies automatically, which is not always achievable.

There is a growing need for a more universal and automated solution that automotive companies can adopt to enhance traceability across integrated toolchains and improve consistency among artifacts. This solution should minimize dependency on external vendors and specific standards. Such a solution would not only improve the efficiency of traceability management but also promote a smoother adaptation process for companies looking to implement or improve their traceability systems.

2.4 Challenges in Achieving Effective Traceability in Automative Industry

A comprehensive study by Maro et al. identified 22 traceability challenges through two literature reviews and a case study conducted at an automotive supplier company [23]. These challenges are broadly categorized into seven areas: knowledge of traceability, tools, human factors, organization and process, uses of traceability, measurement of traceability, and exchange of traceability information. Notably, three challenges are closely related to the problems addressed in the thesis.

Inaccessibility of artifacts: The first challenge involves the difficulty in accessing artifacts that need to be connected by the traceability link, especially in projects with a large number of artifacts. Tools that support searching by ID or keywords can make it easy for users to find necessary artifacts, so the basic method to build traceability, which is manually copying the ID from one tool to another, becomes available.

Perceived as an overhead: The second challenge is the perception among developers that creating traceability is an additional, disruptive activity to their workflow. This perception arises partly because the individuals creating traceability links often do not benefit directly from them, leading to low motivation and potentially incorrect or missing links. The proposed solutions are to ensure that the traceability links created provide immediate benefit to the creators and also to automate the tasks whenever possible.

Lack of proper visualization and reporting tools: The third challenge focused on the lack of visualization and reporting tools for traceability links, which are presented in large tables or lists. This makes it hard to comprehend the links or detect the flaws in them. Insufficient visualization can arise from the manual creation of traceability links, such as copying IDs between artifacts. Some requirement management tools used in the industry don't support such kind of visualization. The absence of visualization also results in the traceability links never being used or

being under-utilized. To address these issues, it is essential to provide end users with proper visualization tools and powerful reporting tools capable of producing overviews and statistics for reviews. Effective visualization should enable users to see which requirements are already implemented and tested or which tests do not have corresponding requirements. Additionally, tool vendors are encouraged to develop features that allow custom reports to be generated from traceability information based on user needs.

2.5 Traceability Visualization

Visualization is a process of "representing data, information, and knowledge in a visual form to support the tasks of exploration ... and understanding" [33]. It occurs when interacting with visual elements, which represent data that have been collected, stored, fetched, filtered, transported, and transformed to and from some data storage [34].

In extensive and complex systems, there are more traceability links and the complexity of their usage increase. Therefore proper visualization helps understand traceability.

Li et al. discussed three main concepts of traceability visualization: the traceability information (what to visualize), the visualization technique (how to visualize), and the task context (when to visualize) [6]. The authors suggested visualizing the artifact types, artifact metadata, granularity, artifact attributes, link information as well as confidence. Techniques like matrices, graphs, lists, and hyperlinks are found to be useful and helpful. Several techniques of traceability visualization were proposed by Heim et al. and Duan et al. as well, aiming to help users understand the "cloud of links" and get to the underlying information smoothly [35] [36]. Traceability visualization can be employed in various tasks of software development, ranging from management, design, and implementation, to testing. In software testing, it validates if requirements have been implemented correctly [37]. It is usually linked to artifacts like test cases and requirements. Visualizations of software testing support hypothesis generation, and decision-making, and can map to information needs [38].

The visualization of test results, including traceability information, is important. It helps understand the test coverage and the relationship between tests and requirements, making identifying gaps in the testing strategy easier [39]. Good visualization supports better decision-making by providing clear insights into the software quality and areas that may need more focused testing or development efforts.

2.6 Hardware-in-the-loop Testing

Hardware-in-the-loop (HIL) simulation is a technique for performing system-level testing of embedded systems in a comprehensive, cost-effective, and repeatable manner [40]. It rigorously tests the functionalities, system integration, and communication of electronic control units (ECUs) within a simulated environment. A HIL

simulator is connected to a test object and an HIL PC to enable HIL testing.

In the case department, there are three ways to perform HIL testing:

- Manual execution of in software called dSPACE ControlDesk.
- Direct trigger of automated HIL testing via software called dSPACE AutomationDesk. Users select the project to be tested and import the necessary libraries to execute HIL test cases directly through the AutomationDesk software.
- Indirect trigger of automated HIL testing via Jenkins. There are two types of pipelines defined for this purpose. The first pipeline is triggered on demand when testers need to perform HIL testing on a project. The second pipeline operates as follows: a developer commits changes to the source code. Jenkins will query the Version Control System for any source code updates. Upon detecting an update, Jenkins triggers a build job including various steps in a pre-defined sequence. When the software has been compiled and packed in a format suitable for installation on the test object, a test request is dispatched to the HIL PC. A script on the HIL PC then downloads, unpacks the software, and launches a specific test suite, such as an acceptance test. The outcomes of the test phase are reported back to Jenkins. Finally, the end user can access the Jenkins Web server to review the build and test results.

2.7 Tools and Software Used at the Company

2.7.1 dSPACE AutomationDesk

The case company uses a tool named AutomationDesk from dSPACE, a leading supplier of HIL simulators at Volvo Car Corporation, for conducting automated HIL testing. AutomationDesk offers a user-friendly graphical interface, enabling test case developers to create and modify test routines without the need for programming expertise.

Essential concepts in AutomationDesk related to this thesis include project, sequence, library, data object, result, and report:

- A project in AutomationDesk serves as a container for all instantiated resources that implement a specific automation task, termed an automated test case in this thesis. It encompasses the implementation of HIL test cases, outcomes of executed tasks, and their corresponding reports. The Project Manager pane allows users to handle and display project elements in a hierarchical layout.
- A sequence represents the implementation of an automation task as a control flow, defined with automation blocks. It is editable via the Sequence Builder pane. The main components of a sequence are blocks. Users can create blocks independently, or select from template blocks available in AutomationDesk libraries, both built-in and customized. The graphical interface facilitates the

easy organization of blocks into a control flow chart to form a sequence and execute test cases.

- A library acts as a storage for templates that aid users in creating sequences, data objects, or automation blocks within their projects. Accessible through the Library Browser panel, libraries in AutomationDesk are categorized into built-in and custom libraries. Built-in libraries come as predefined software components, with source code hidden from users, whereas custom libraries allow users to create elements per their requirements. Valid elements that can be added to a custom library are templates for data objects, blocks, and sequences.
- Data objects are designed to hold values of various types such as string, float, int, and file, among others. They are instantiated specifically for a project, sequence, or an automation block. A data object can also be referenced to other data objects.
- A result is a set of data derived from executing a project, project folder, or sequence. Based on the result, users can generate a report document. While AutomationDesk provides default PDF and HTML formats for reports, users have the option to customize style sheets via XSL files for PDF report generation.

2.7.2 CarWeaver

To manage system designs and requirement documentation, the case department employs CarWeaver, a derivative of SystemWeaver which is an information management platform by Systemite. CarWeaver is enhanced with various plugins by Volvo Car Corporation to customize both the model and its functionalities to meet specific needs.

CarWeaver's primary function within the department is to describe test case, alongside establishing and managing a multi-level requirements framework. This setup supports a many-to-many traceability matrix, allowing a single test case to correspond to multiple requirements, and vice versa, ensuring comprehensive coverage and integration. However, CarWeaver's capabilities are focused on describing test case information, such as test case ID, test case purpose and so on, rather than implementing or executing these test cases, which is handled by other software. CarWeaver does not support the establishment of traceability between test case implementation, test case execution and requirement specification.

The thesis mainly utilizes the requirement ID and test case information in CarWeaver.

2.7.3 XML, XSLT, XSD

XSD (XML Schema Definition), XML (eXtensible Markup Language), and XSLT (eXtensible Stylesheet Language Transformations) are interconnected technologies

used to transform and represent XML data. They're used in test report generation, web services, and so on.

XSD, also known as XML Schema, defines the structure and content of XML documents. It outlines how to describe XML document elements, including data types, sequences, and so on. XSDs ensure XML documents adhere to defined constraints and structures. It also validates if an XML document meets the expected format.

XML is a markup language used for encoding documents in a format readable by both humans and machines. Its primary purpose is data storage and transportation, with a focus on simplicity, generality, and usability across the Internet. Users can create and define their own tags and document structure in XML.

XSLT is a language for transforming XML documents into other XML documents, HTML for web pages, or other document types such as PDF. It applies stylesheets to XML documents, transforming and rendering the original document into a new format using XPath.

In summary, XSD defines the data structure, XML represents the data itself, and XSLT determines how to display the XML data.

AutomationDesk generates reports based on execution results, which are provided as XML files. These XML files follow the rules set by XSD files that are provided by AutomationDesk. To customize the report's stylesheet, users must write XSLT files.

2.7.4 Power BI

Microsoft Power BI, developed by Microsoft, is an interactive tool designed for data visualization. It supports data import from various sources including databases, web pages, PDFs, and structured files like spreadsheets (CSV, XML, JSON, XLSX) and SharePoint. The case company uses Power BI to display diverse data types through an easy-to-use interface.

2.7.5 Jenkins

Jenkins is an open-source software that streamlines the automation of various software development tasks such as building, testing, and deploying. This capability supports the principles of continuous integration and continuous delivery by ensuring that new changes are seamlessly integrated. Within the scope of the the thesis, Jenkins is used for HIL testing, which can be initiated automatically after a successful build or manually, based on user requirements.

3

Research Methods

The research methodology adopted in this study is design science research (DSR). The thesis project follows the regulative DSR cycle described in [41] and the guidelines for applying DSR outlined in [42] [43].

3.1 Design Science Research

Design Science Research (DSR) is a methodological paradigm within Information Systems. It seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts [43]. These artifacts, from models and methods to processes and software systems, are developed to address specific organizational challenges or opportunities.

DSR is inherently a problem-solving process. The core of DSR lies in its iterative process of building and applying these artifacts to gain knowledge and understanding of both the problem domain and the solution space.

A design science contribution can be based on a new artifact. It can also be an improvement upon an established solution to a well-known problem or just a marginal modification of an existing artifact [44]. The main goal of this study is to build tool integration for requirement traceability in complex environments. It involves the development of a requirement-centric framework that has high generality. Such an objective aligns with the principles of design science research, making it the chosen methodology for our study.

In line with the general guidelines for conducting design science research, the study also adhered to the seven guidelines for running a DSR master thesis with industry provided by Knauss [42].

3.2 DSR Iterations

The practical DSR framework applied in this thesis is the regulative cycle described by Roel Wieringa [41]. The regulative cycle can be found in Figure 3.1. Design science research should be conducted in several iterations. Each iteration starts by investigating a practical problem. From the second iteration onwards, each iteration is built on the previous one, meaning the problem also comes from solving earlier

practical problems. The next step is designing solutions for the problem identified earlier. These solutions are then validated. One chosen design will be implemented. The outcomes of this implementation are then evaluated, which could be the start of a new turn through the regulative cycle.

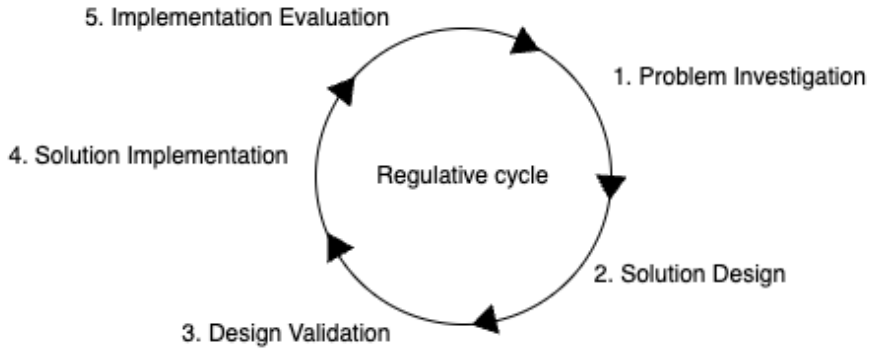


Figure 3.1: A regulative cycle

This thesis research was carried out over two iterations, each containing the complete range of design science research steps, from the initial problem investigation to the implementation evaluation. The following sections give an overview of both the definition and application of DSR activities. They also include a summary of how these activities were implemented in the context of this thesis.

3.2.1 Problem Investigation

Problem investigation is the first step in the regulative DSR cycle. It seeks to gather insights and comprehension about the problem without changing it. The problem should be clearly described and explained after this step. Additionally, it should outline the potential outcomes if the problem remains unaddressed.

The problem investigation can be categorized into four types depending on the underlying reasons and purposes [41].

Problem-Driven Investigation: This is essential in the early stages of DSR. The purpose of problem-driven investigation is to describe, explain, and diagnose the problems.

At the very beginning of the thesis study, the author examined the team's responsibility, current workflow, and way of working. This was done by reading through documents hosted on the company's documentation platform named Confluence, participating in the teams introductory presentation, checking the latest acceptance testing report, and communicating with team members.

Upon gathering basic background information about the company and team, the author proceeded to organize multiple meetings and discussions with a system engineer from the team. These interactions focused on exploring the traceability issues encountered by the case ART. As the system engineer has six years of experience in the

company, working closely with software integration and delivery, while others in the team have limited experience in requirement traceability, the initial problem investigation was strongly supported by the system engineer. During the problem-driven investigation, the author also looked into the existing HIL testing reports, current web pages of test result visualization, and the workflow to build the visualization.

A literature review was also performed on the challenges and potential solutions regarding traceability and consistency in both the general software industry and the automotive sector specifically. This review helped in a deeper understanding of the reason and situation of common traceability issues, thereby equipping the author with insights for designing solutions applicable beyond the case company.

Goal-Driven Investigation: This is motivated when there is no problem experienced but changes are expected to happen in agreement with some goals. Goal-driven investigation usually involves an analysis of goals to be achieved. This can be done through the description of stakeholder goals, definition and operation of them, as well as prioritizing the goals. The goal-driven investigation was not included in this study because the problems related to the study already exist within the case department.

Solution-Driven Investigation: This is based on the search for new technology that can solve the problems. The investigation can start by exploring new technology to understand its properties. This thesis study doesn't focus on any specific new technology.

Impact-Driven Investigation: This is to investigate the results of previous actions, instead of preparing for future solutions. Therefore it's also called evaluation research. This kind of investigation usually involves describing the earlier implemented solutions, analyzing their impacts, identifying stakeholder goals, and so on.

The second iteration cycle used the impact-driven investigation to identify issues in the artifact implemented during the previous cycle. The problem investigation activities were part of the evaluation stage of the first cycle as can be seen in Figure 3.2. The first iteration concluded with a workshop presenting the framework, followed by a survey. The overlap between the evaluation of the first cycle and the problem investigation of the second cycle occurred after the survey. It began with a focused discussion group to analyze the feedback from the workshop and continued with an initial code review session. The problem investigation in this cycle concentrated on the security of the designed framework and potential improvements to the implemented workflow.

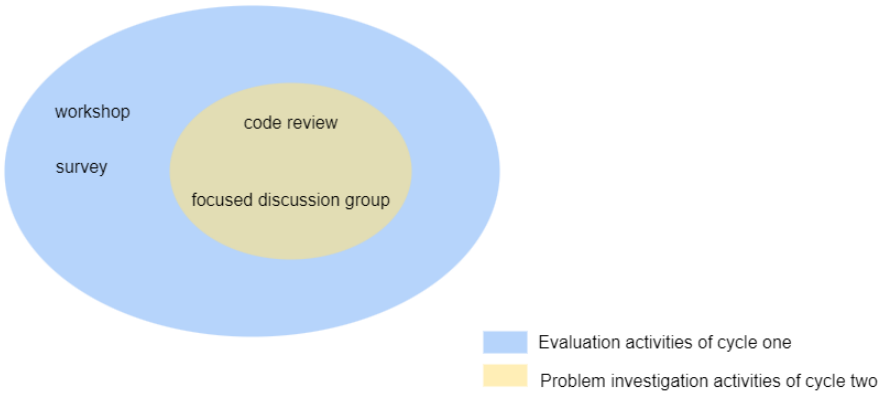


Figure 3.2: Evaluation activities of the first cycle and problem investigation activities of the second cycle. The workshop, survey, code review and focused discussion group together evaluated the artifacts developed in the first cycle. The code review and focused discussion group also identified problems in the second cycle.

3.2.2 Solution Design

Solution Design is the process of creating innovative solution candidates.

In the initial iteration cycle, the author reviewed the literature on existing solutions to traceability and consistency challenges. This review provided valuable insights into tool integration. Although the study identified multiple problems in the current traceability workflow, it did not create separate solutions for each issue. Inspired by existing toolchains, the author designed a more generic and broadly applicable framework to connect different software within the context of a safety-critical environment, focusing on requirement-centric approaches.

Once the idea of the framework was decided upon, the author explored the software tools involved in-depth to develop detailed designs. This included the requirement management tool CarWeaver and the automated HIL testing software dSPACE AutomationDesk. A critical part of this process was reviewing documentation on AutomationDesk to understand its features fully and gain insights into the detailed design of the framework. This review focused on the methods for incorporating specific information into test sequences as desired and exporting test results in a format that could be converted for compatibility with data from other software.

Discussions with the company's system engineer, as mentioned earlier, also significantly influenced the solution design during the first cycle.

The design of the solution for the second iteration cycle primarily stemmed from the focused group discussions and the initial code review session held after the evaluation workshop in the first cycle. This cycle emphasized improving the framework implemented previously. Therefore, devising potential solutions became more straightforward than in the initial cycle.

3.2.3 Design Validation

Design validation is a knowledge task used to assess if a proposed design will help stakeholders achieve their goals once it is implemented. Roel Wieringa proposed three validation questions to guide this evaluation, listed as follows [41].

- Internal validity: "Would this design, implemented in this problem context, satisfy the criteria identified in the problem investigation? This contains two subquestions: In problem domain D, would solution S have effects E? Does E satisfy stakeholder criteria C?"
- Trade-offs: "How would slightly different designs, implemented in this context, satisfy the criteria?"
- External validity: "Would this design, implemented in slightly different contexts, also satisfy the criteria?"

The thesis study followed the guidance to perform design validation. In iteration one, all three questions were answered for the proposed framework. In the second iteration, the focus was only on internal validity, since this iteration mainly made improvements to the previously developed framework according to the evaluation feedback.

3.2.4 Solution Implementation

One or multiple solutions that are designed and validated in the previous steps are implemented in this stage.

Implementation of iteration one focused on building the framework, while the one for the second iteration mainly addressed the feedback collected from the evaluation of the first iteration to improve the framework.

As the proposed framework fetches data from different software and processes them to get analysis results, the implementation covers data generation, data fetching, data processing, data analysis, and visualization.

The framework implementation started with the traceability establishment from AutomationDesk. The first part is a suite to generate a traceability table in the test report. It includes a customized library in AutomationDesk to add traceability information to the test result, an updated XSL script for report styling, and guidance for updating the test cases. The second part features a key Python script to parse the test result from XML format to JSON format that contains necessary traceability data. The script can be executed with various parameters, either through the Jenkins pipeline script or directly within AutomationDesk via a built-in GUI tool that is also developed by this study.

The implementation then proceeded with the interaction with CarWeaver, which included retrieving requirement data via API (Application Programming Interface) and processing the data to make it comparable with the test results in JSON format. This part was programmed with Python.

When the data generation, retrieval, and process are done, data analysis begins. It compares the JSON files, analyzes the requirement status, and stores the results in a CSV file which will later be visualized. When the status of all requirements of the scope under test is known, the status of the scope can be easily known as well. Python scripts were created during this stage.

The final step is to import the CSV result into Power BI to build a visualization of the project status.

In the second iteration, the implementation involved running the API utilization code on a different Jenkins node than the one communicating with AutomationDesk. Additional changes were made based on the code review and the focused group discussion.

Implementation details will be discussed in the Findings section.

3.2.5 Implementation Evaluation

The implementation evaluation stage evaluates if the implemented solutions can solve the problem identified correctly. This phase may overlap with the problem investigation of the next cycle if there is one.

In the first iteration, the author evaluated the implementation by conducting a workshop at the case company to demonstrate the solutions. Subsequently, a survey was distributed to gather detailed feedback on the functionality and usability of the solution. This was followed by a focused discussion group to explore the feedback received from the workshop further. Additionally, an initial code review was conducted to assess the quality of the code and identify any issues.

In the second iteration, an evaluation meeting was held with a system engineer to validate the updates and assess how the framework performs in various contexts.

# Participant	Role	Activities				
		Workshop	Survey	Code Review	Focused Discussion Group	Evaluation Meeting
Person-A	System Engineer	X	X		X	X
Person-B	Software Engineer	X		X	X	
Person-C	Software Engineer	X	X			
Person-D	Integration and Release Engineer	X	X			
Person-E	Integration and Release Engineer	X	X			
Person-F	DevOps Engineer				X	
Person-G	DevOps Engineer				X	
Person-H	Electrical Engineer	X	X			
Person-I	Electrical Engineer	X	X			
Person-J	Master's Student	X	X			
Person-K	Test Engineer	X	X			
Person-L	Test Engineer	X	X			
Person-M	Test Engineer	X				
Person-N	Test Engineer	X				
Person-O	Software Verification Engineer	X				
Person-P	Software Engineer	X				
Person-Q	Integration and Release Engineer	X				

Table 3.1: Participants in the evaluation activities of the two cycles.

Table 3.1 lists the participants in the evaluation activities across two cycles. Fifteen individuals took part in the workshop, of whom nine responded to the survey. The code review was conducted by the author and a software developer. The focused

discussion group included four stakeholders from the case company. Both the code review and the focused discussion group contributed to the problem investigation in the second cycle. At the end of the second cycle, an evaluation meeting was held with a system engineer.

4

Findings

4.1 Cycle I Findings

4.1.1 Problem Investigation

In the first iteration cycle, a problem-driven approach was adopted as outlined in the Research Methods chapter. The initial investigation was based largely on many discussions with an experienced system engineer at the case company, who specializes in software integration and release. These discussions were intentionally unstructured to foster deeper insights.

Document reading also strongly supported the problem identification. The case company uses Confluence as one of the workspaces for knowledge sharing and document management. This platform houses extensive information across various teams and departments and is regularly updated, making the investigation process more efficient by reducing reliance on time-consuming interviews, workshops, and meetings. Although the author reviewed many documents on Confluence, only those directly related to the thesis study will be described in the subsequent section.

A comprehensive literature review was conducted to understand the general background of the thesis study. In the first iteration, the review focused on challenges related to requirement traceability and consistency in the automotive industry. Several studies suggested tool integration as a solution to these challenges, and it was then explored.

The problems identified in this cycle will be elaborated as follows: First, the thesis motivation and the case team's operational methods will be discussed. Subsequently, a thorough analysis was conducted to answer Research Question 1 and Research Question 2, covering how testing is executed in the integration and release processes, the state of traceability before this study, issues with the test simulation tool AutomationDesk, and the current status of visualization.

The case team in this thesis study is responsible for integrating and releasing inverter software. The development of this software adheres to AUTOSAR (Automotive Open System ARchitecture), a standardized framework for automotive software. This architecture is structured into distinct layers, as illustrated in Figure 4.1. The foundational Hardware layer consists of physical elements, including electronic control units (ECUs). Above this, the AUTOSAR Runtime Environment (RTE)

serves as middleware, supporting the interaction between the application software and the hardware via standardized interfaces. The System Software, also known as basic software, is organized into four functional groups. At the uppermost level, the Application Layer houses the software components that perform specific vehicle functions [8].

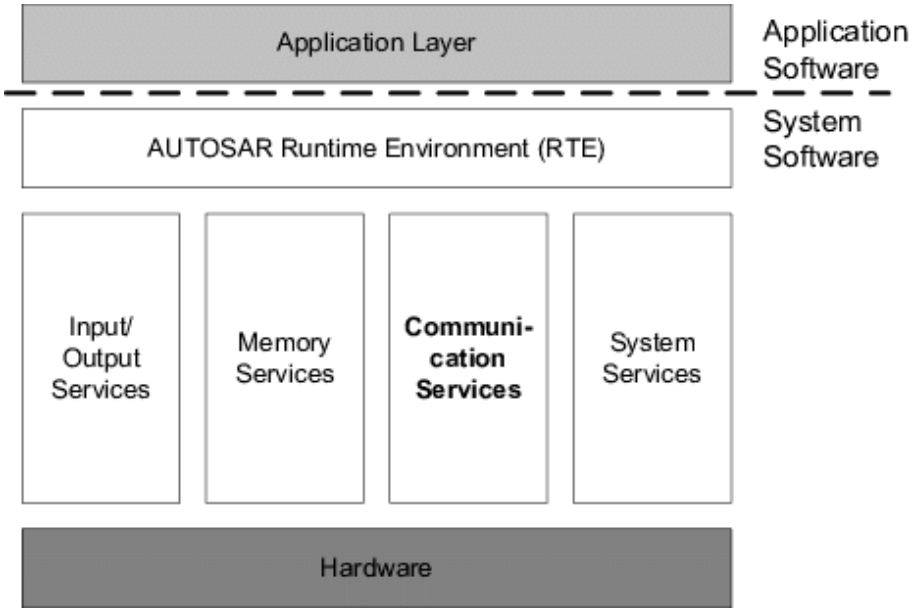


Figure 4.1: AUTOSAR-System Software Stack Overview

The integration and delivery activities within the case team are based on this architecture. Software components for the application layer are developed separately by different in-house teams within the company. The case team then integrates these components with the runtime environment to build the integration software. The third layer, system software, comes from external suppliers. Once the integration software is assembled, the case team executes important HIL test cases to ensure everything works as a preliminary check, known as a smoke test, before sending the software to the supplier. Upon receiving the integration software, the supplier develops it into fully functional software and sends it back to the case team. This final software undergoes acceptance testing through HIL, carried out by another internal acceptance testing team. The workflow is depicted in Figure 4.2. Although this description simplifies the actual delivery process, it provides enough detail to understand how HIL testing and traceability are applied currently.

As indicated in Figure 4.2 and mentioned above, two testing activities happen in the integration and delivery process.

1. The first activity occurs after the integration software is assembled and before it is sent to the external supplier. At this stage, smoke testing is carried out to quickly verify the functionality of the most important features. Only a select few of the defined test cases are performed, as these are deemed the most critical. In some projects, test cases for smoke testing are automated using dSPACE AutomationDesk, whereas in others, they are manually executed us-

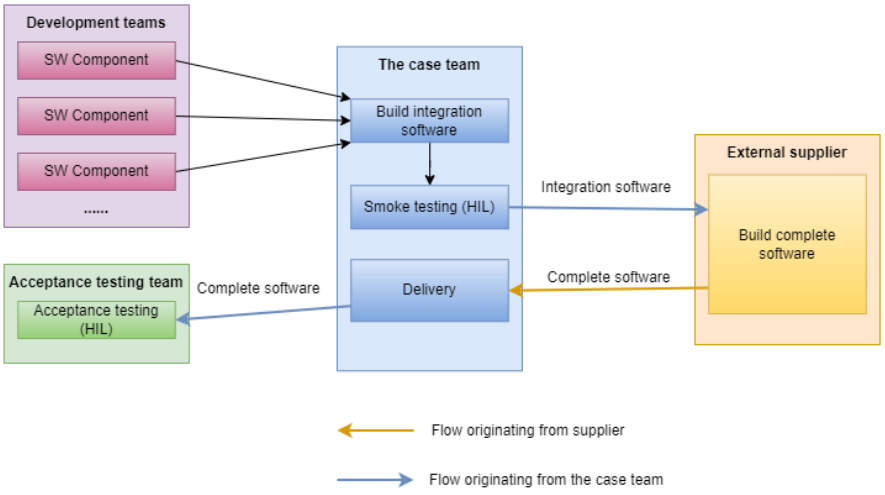


Figure 4.2: Integration and delivery process in the case team.

ing dSPACE ControlDesk. The automated tests can also be initiated through the Jenkins pipeline.

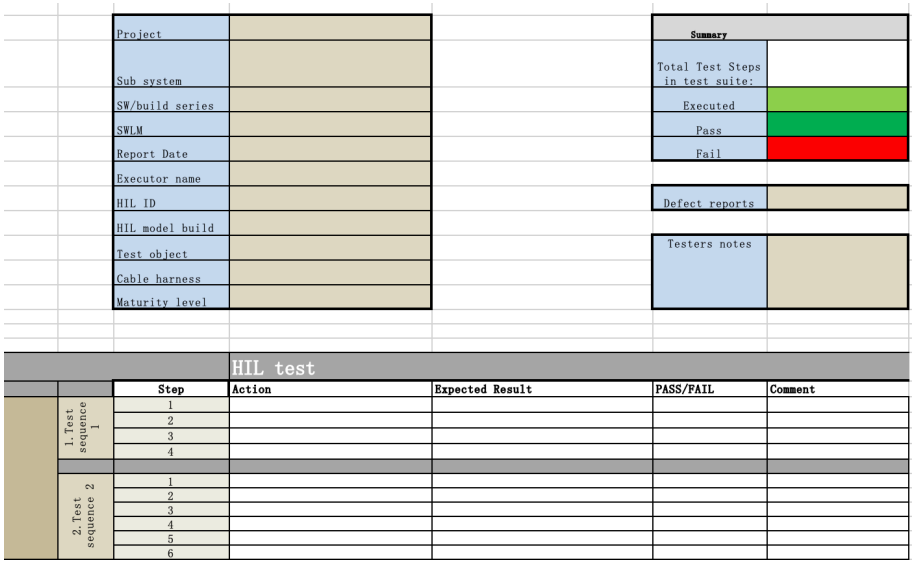
2. The second testing activity takes place within the Acceptance Testing team, where acceptance testing is conducted on the fully developed software received from the external supplier. Similar to the smoke testing phase, some test cases for acceptance testing are automated, while others require manual execution.

For both smoke and acceptance testing, regardless of how test cases are executed, the results must be manually recorded in an Excel sheet template by the tester. This template, commonly used in the case department across various testing stages, begins with a test overview section. This section includes metadata about the project under test, tester information, and test results. Following the overview is a detailed test specification section, where test cases are listed along with corresponding test steps and outcomes. The specifics of the test cases vary depending on the project under test and the type of testing being conducted. This template is illustrated in Figure 4.3.

A limitation of this template is that it focuses exclusively on test execution without including information on related requirements.

To establish traceability using this template, a separate Excel sheet is employed. This sheet maps requirements to test cases as specified in CarWeaver, and it is created based on the data exported from CarWeaver. Although this secondary Excel sheet contains multiple columns, the most important ones include ownership of test cases and requirements, test case name, test case ID, requirement ID, requirement handle, testing level, and test outcome. Figure 4.4 shows some of the template's key headers.

This template is stored on Confluence. Testers must download a new local copy each time they intend to use it. During use, the tester needs to filter out the relevant test cases to be executed and enter the verdicts manually. Upon completion, the sheet



Project	
Sub system	
SW/build series	
SWLM	
Report Date	
Executor name	
HIL ID	
HIL model build	
Test object	
Cable harness	
Maturity level	

Summary	
Total Test Steps in test suite:	
Executed	
Pass	
Fail	

Defect reports	
Testers notes	

HIL test

	Step	Action	Expected Result	PASS/FAIL	Comment
1. Test sequence 1	1				
	2				
	3				
	4				
2. Test sequence 2	1				
	2				
	3				
	4				
	5				
	6				

Figure 4.3: Template to record smoke/acceptance testing result.

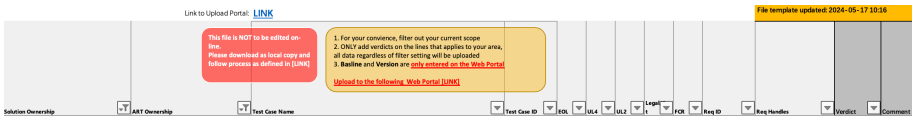


Figure 4.4: Part of headers of the template for traceability building.

must be uploaded to a custom web portal designed in-house. This portal processes the data, forwards it to a company-specific message bus, and visualizes the project status in Power BI. Figure 4.5 provides an example of the resulting visualization.

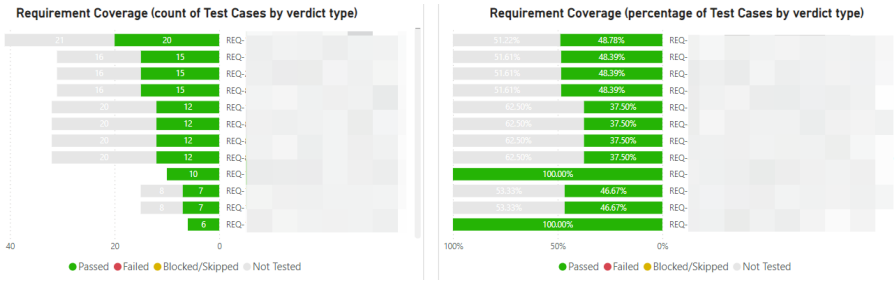


Figure 4.5: Visualization dashboard of the project status.

Since the traceability template is derived from data exported from CarWeaver, any updates made to specific information within CarWeaver require the template to be regenerated. Currently, this process of version control is handled manually, which poses potential issues such as human error, network disruptions, and other operational challenges. Additionally, the existing method for managing traceability involves several manual steps, making it time-consuming and complex.

The workflow described above for building traceability and visualization is only suited to manual HIL testing and does not support automated HIL testing. This limitation stems from a lack of appropriate data, such as the information typically found in an Excel template. However, the company is now aiming to increase the

use of automated test cases. More test cases will be transitioned to AutomationDesk and executed automatically, rather than manually. This shift in testing methodology requires an automated approach to establishing traceability, which can eliminate the need for manual input into Excel templates.

Given that similar workflows of building traceability for manual HIL testing are already in use across various departments within the company, and considering the broad impact of increasing automation, developing a new solution for traceability in automated HIL testing could benefit not just the case department but also the whole company.

More issues related to inconsistency emerged as the author investigated the practical application of AutomationDesk and CarWeaver.

As previously discussed in the Background section, in the context of HIL testing, CarWeaver is used to designing and defining test cases, while their implementation and execution are carried out in dSPACE AutomationDesk.

In dSPACE AutomationDesk, each test sequence contains a "Requirement" data object, which specifies the particular requirements it aims to test. Ideally, this data object should contain the requirement ID from CarWeaver. However, test cases are developed by various teams, each focusing on different components, and inconsistencies arise in documenting the requirement information.

For some test cases, the Requirement data object correctly matches the requirement values stored in CarWeaver. However, certain data, while initially aligned with CarWeaver's records, have become outdated; changes made in CarWeaver are not updated in the AutomationDesk test cases.

For other test cases, the Requirement data object values do not directly match those found in CarWeaver. Some have relatively clear requirement descriptions that are possible to interpret and locate in CarWeaver, whereas others are only described with less informative text that may be understood only by the developers who created them. This inconsistency in documentation standards makes it difficult for different teams to have a uniform understanding and access to requirement details.

Another challenge in tracking requirement information in the data object is that some values are still linked to the old requirement management tool. Since the company transitioned from this old platform to a new one, which is CarWeaver, updates to both test cases and requirements are necessary. Although CarWeaver monitors these changes across the two platforms, AutomationDesk does not recognize them. As a result, the requirement ID linked to a HIL test case in AutomationDesk might not reflect the latest version. Additionally, some test cases refer only to a subset of relevant requirements in the Requirement data object. This lack of consistency in maintaining up-to-date traceability information between AutomationDesk and CarWeaver presents risks for product releases and complicates the testing process, ultimately affecting the efficiency of software releases.

Additionally, Merely having a Requirement data object is insufficient to establish effective traceability. Given the many-to-many relationship between test cases and

requirements, it is not feasible to automatically determine which test case implemented in AutomationDesk corresponds to which test case definition in CarWeaver.

As discussed above, it is possible to manually trace CarWeaver information in AutomationDesk to some extent and vice versa if the Requirement data object is correctly configured. However, this workflow has a significant gap due to the absence of a direct connection between these two software systems. When users need to trace a particular item, they find it tedious and time-consuming because they have to manually navigate through CarWeaver or AutomationDesk to locate the corresponding information. This situation is far from ideal, as it affects the efficiency of tracking and managing project artifacts across these platforms. Moreover, the observed inconsistency between CarWeaver data and AutomationDesk requirement information cannot be addressed due to the inadequate implementation of traceability for automated HIL testing.

The challenge extends to the reporting mechanism in AutomationDesk. The core problem lies in the fact that these reports, serving as artifacts within Jenkins jobs in the CI pipeline as well, do not include vital requirement information. The lack of traceability information in the report makes it tedious to identify if the planned requirements for a release cycle are all validated or not. It also hinders the identification of requirements of error-prone test cases.

Owing to the issues outlined above, the case team, the acceptance testing team, and other departments employing a similar workflow have been unable to adhere to the ASPICE standard for qualification testing, which prioritizes traceability and consistency.

Other than the inconsistency issue, many test cases defined in CarWeaver are not implemented in AutomationDesk, meaning they're either just defined but not implemented, or performed manually instead of automatically. This issue was noticed because the author examined a small subset of test cases and their corresponding requirements in CarWeaver, then searched for them in AutomationDesk using either the test case names or the requirement IDs.

Apart from the lower implementation rate of automated test cases, other challenges in the current visualization flow for automated testing have been identified. Currently, traceability visualization has been limited to manual testing at the case company. The visualization of requirement traceability in automated testing is not available because there is no suitable data format for recording test execution results. It has been noted that while automated integration and verification test results from some ongoing projects are displayed in Power BI, this visualization is restricted to test verdicts per baseline. Specifically, it only includes the Baseline name, CarWeaver Test case Test System IDs, Test Case Name, and Test Result. That means there is no traceability between tests and requirements, making it impossible to effectively get the requirement coverage.

4.1.2 Solution Design

The main problems identified in this iteration are the absence of traceability in automated HIL testing and the mismatch between AutomationDesk test cases and CarWeaver requirements. Data plays an important role in establishing traceability and identifying these mismatches. Moreover, it is an essential resource for all systems involved in traceability and reporting.

To address the issues outlined earlier, several key questions must be considered: How is data stored across different software applications? How can this data be retrieved? And how is data integrated across various software systems? Accordingly, the proposed solution should be data-centric. Since these issues arise from interactions among multiple software programs, a standalone code solution that operates independently of external software is not suitable. Inspired by the literature review during the problem investigation phase, tool integration has proven to be a viable approach.

Currently, there is no tool or plugin available that directly connects CarWeaver and AutomationDesk. As a result, this thesis cannot rely on any existing support tools for integration. The integration methods from existing research, which depend on specific vendors, do not meet the requirements of this study. Furthermore, the use of Excel, previously discussed, will not be included as this study seeks to establish a more automated traceability process that eliminates the need for manually maintaining Excel sheets. The ALS approach described before is also unsuitable because the software in this study does not comply with the OSLC standard. Therefore, the integration must be accomplished in a more generic manner, which involves reading data from the software. Developing a framework that incorporates the software in use could effectively address these issues.

The solution is illustrated in Figure 4.6. It is a requirement-centric framework that can link AutomationDesk and CarWeaver, enabling the establishment of traceability and consistency checks. The core concept of this framework is to extract test results from AutomationDesk and requirements specifications from CarWeaver. The data is then processed to enable comparisons, ultimately generating outputs that reflect the status of both the requirements and the overall project.

Eight components developed during this thesis are highlighted in the framework.

COMP 1-1 Updated test sequences in AutomationDesk

One of the issues identified was the lack of test case ID information from CarWeaver in the AutomationDesk test cases. Furthermore, significant inconsistency exists between the Requirement data object and the actual requirement IDs documented in CarWeaver. To establish traceability, it is crucial to ensure that test cases provide accurate information. This requires updating the Requirement data object and introducing a Testcase data object that records the respective test case ID from CarWeaver.

The data objects described above can be viewed as attributes of the test sequence. However, the test results don't output these data objects by default, which prevents

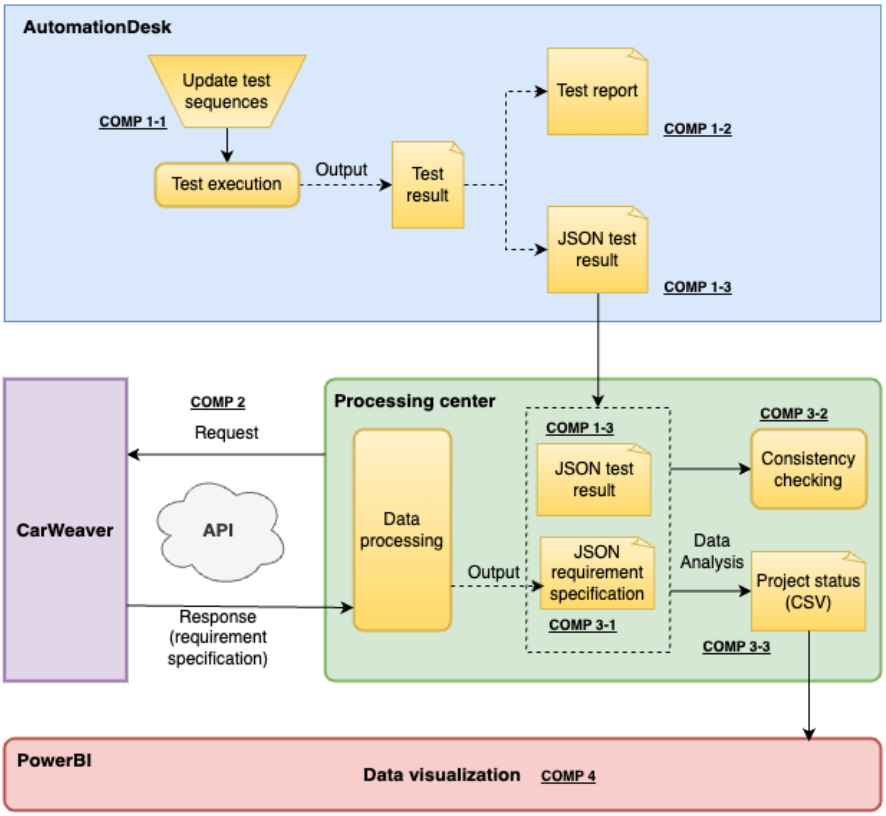


Figure 4.6: A framework to build traceability for automated HIL testing and check consistency between AutomationDesk test cases and CarWeaver requirements.

their inclusion in the test report and JSON test results. Consequently, the test sequence itself needs to be updated to add a new step that will enable the display of data objects in the test results.

COMP 1-2 Test report in AutomationDesk

AutomationDesk supports report generation, enabling users to either use the default report template or customize it according to their needs. The case company uses a customized template for generating their reports. Inspired by the existing HIL information tables in the report, the concept of adding another table specifically for requirement traceability was developed. The new traceability table in the report template allows testers to easily view the CarWeaver requirements and test case IDs of the executed tests.

The creation of the traceability table should be as automated as possible. Its content includes the test case name in AutomationDesk, the test case ID in CarWeaver, the requirement ID in CarWeaver, and the outcome of the test. The prototype can be found in Figure 4.7.

The data presented in the test report are derived from the results produced by AutomationDesk following the test execution.

COMP 1-3 JSON test result in AutomationDesk

Test Case Name	Test Case ID in AutomationDesk	Requirement IDs	Verdict
Sample name	TC-1234	REQ-5678, REQ-0199	Passed
...	...	...	...

Figure 4.7: Prototype of traceability table

Although AutomationDesk outputs test results in an XML file, this format includes excessive details that are not required for establishing traceability. In the framework, compatibility and comparability between data from different software are expected. The JSON format is particularly suitable because it is both human-readable and straightforward to process. Therefore, it is necessary to convert the native test results into a JSON file, which only contains information required for building traceability. Each entry in the JSON file will be structured as follows:

```
{
   "test-case": "A dummy test case name",
   "verdict": "Passed",
   "carweaver-test-case-id": "TC-123456",
   "requirement-id": [
      "REQ-987654",
      "REQ-456789"
   ]
}
```

COMP 2 CarWeaver API usage

The CarWeaver API, developed internally by the case company, offers a wide range of endpoints to access resources hosted within CarWeaver. The API responses are in JSON format and provide detailed information about each requested resource. The requirements within CarWeaver are organized hierarchically. It is possible to retrieve requirements from various scopes by using the appropriate endpoints.

COMP 3-1 JSON requirement specification

Although the CarWeaver API responds in JSON format, the amount of information and its structure are not directly suitable for building traceability. Additionally, accessing some information may require nested API calls. Hence, the requirement specifications retrieved through the API need to be reorganized and stored in a separate JSON file.

COMP 3-2 Consistency checking

Using the JSON files that store test results and requirement specifications, inconsistencies between two software systems can be conveniently reported by comparing the data with a script. By setting specific conditions within the code, these inconsistencies can be detected. Examples of potential inconsistencies include:

- The Requirement data object of a test sequence in AutomationDesk contains

only a portion of the requirements defined in CarWeaver.

- In AutomationDesk, the mapping relationship between the Testcase data object and the Requirement data object doesn't align with the definition in CarWeaver.
- Either the Testcase or Requirement data object doesn't exist in a specific scope in CarWeaver.
- Some test cases that should be executed according to the requirement specification in CarWeaver are not included in AutomationDesk.
- Duplicated test cases are executed in AutomationDesk.

COMP 3-3 Project status (CSV)

In CarWeaver, requirements for various scopes are clearly defined. Each requirement can be linked to multiple test cases. To thoroughly validate a requirement, it is essential to cover all associated test cases during the test run. Executing HIL test sequences that belong to the same requirement scope allows for the analysis of the requirement status. By understanding the status of all requirements within a specific scope, such as a project, it is possible to determine the overall status of that scope.

Requirement statuses are divided into four categories: fully validated and passed, fully validated but failed, partly validated, and not tested.

- Fully Validated and Passed: This status indicates that for a requirement defined in CarWeaver, all connected test cases have been covered in the test run and have passed successfully.
- Fully Validated but Failed: This status applies when all test cases connected to a requirement in CarWeaver are covered in the test run, but at least one test case has failed.
- Partly Validated: This status means that only some of the test cases connected to the requirement in CarWeaver have been covered in the test run.
- Not Tested: This status is assigned when none of the test cases connected to the requirement have been tested.

Since requirement information is managed in CarWeaver, and HIL testing is conducted using AutomationDesk, integrating these tools is a key step. To analyze the requirement status, it is important to prepare and harmonize data from both software systems. This integrated data can then be used to assess and compare the fulfillment of project requirements.

By analyzing the JSON data of test results and requirement specifications, a CSV file can be created to display the requirement statuses for a specific scope. This file also helps determine the overall requirement status of the scope.

COMP 4 Data visualization in Power BI

The CSV file containing the requirement statuses can be visualized using software

like Power BI. Various diagrams like a pie chart can be created to provide a user-friendly view of the data, such as showing the coverage of requirements.

4.1.3 Design Validation

The three validation questions proposed by Wieringa are answered in this iteration.

- **Internal validity**: The designed framework supports the creation of requirement-centric traceability and helps in the identification of inconsistencies. When the framework is used, the HIL testing report will include a traceability table. Python scripts will be used to process and analyze data from test results and requirements, eliminating the need for manual data entry and maintenance in Excel sheets. This ensures robust traceability and consistency checks for automated HIL testing, addressing the issues identified in the first iteration cycle.
- **Trade-offs**: The framework will still operate with minor variations. For instance, although JSON is chosen for structuring test results and CarWeaver requirements, other formats like XML and YAML are also suitable due to their capability to organize data effectively. This adjustment affects components COMP 1-3 and COMP 3-1, as modifications to the code are required to support file creation in these formats. For the storage of test results and requirements, using a centralized database that supports a search mechanism could also support the frameworks functionality. In such a scenario, data interactions would occur between the processing center and the database rather than separately from CarWeaver and AutomationDesk. The specific implementation would depend on the chosen database. Furthermore, visualization tools are not confined to Power BI; alternative dashboard applications can be employed, although the procedures to create graphs will differ based on the software used.
- **External validity**: The framework does not depend heavily on specific external vendors, allowing for wide applicability across various contexts. Even if CarWeaver is replaced with another requirements management platform, or if different software is used for testing instead of AutomationDesk, the framework remains functional as long as the new software supports data exportation. This thesis uses an API to retrieve data from CarWeaver, but alternative methods such as direct database access or web scraping are also viable.

  The core concept of the framework involves data preparation, transportation, processing, analysis, and visualization. Therefore, as long as valid data is available, the framework can be adopted. This versatility makes it highly adaptable for companies that employ different workflows and software for testing and requirements management.

4.1.4 Solution Implementation

The framework is implemented by assembling the defined components within it.

In the thesis study, the implementation started with updating test sequences in AutomationDesk and adding a new traceability table in the test report. A deep investigation was performed to understand how the report generation works in the existing workflow.

In AutomationDesk, a report can be generated automatically after a sequence has been executed, or from an existing result, meaning that a report is always based on a result. AutomationDesk uses a standardized XSD file as a schema for test results. The result data is stored in XML format post-execution and conforms to the XSD schema. Without this XML data, a report that includes the execution results cannot be created. An investigation of the XSD file used by AutomationDesk for execution results revealed that the key elements supporting customized report generation are called "blocks". Therefore, as long as the data is included in the test result, it can be retrieved, parsed, and placed in the final traceability table.

AutomationDesk offers two default formats for presenting reports: PDF and HTML. Users can also create their style sheets for the reports by writing XSL files per the XSD schema.

The mechanism to generate the test report is illustrated in Figure 4.8.

The case department has developed a customized style sheet, consisting of a series of XSL files, to enhance the presentation of the report. Compared to the standard formats, these customized reports provide a more user-friendly display of information, featuring a glanceable header section. This section includes additional data such as the system under test, a summary of the test case, and information about the test object. For ease of use and quick reference, it was decided to place the desired traceability table in the header section.

COMP 1-1 Updated test sequences in AutomationDesk

Data objects can be treated as attributes of the test sequences and are not automatically displayed in the test report or results. To have them included in the test results, a block that contains the data object information must be developed and explicitly added to the test sequence.

The thesis does not undertake the actual updating of existing test cases, as this requires relevant metadata, typically only known by the test case developers. The test case developers need to perform the updating activities by themselves.

As discussed in the solution design phase, the updates that should be made to the test sequences are outlined as follows

- **Update Requirement data object**

  Test case developers should find the respective requirement ID in CarWeaver. Afterward, by selecting the data object in AutomationDesk and entering the requirement ID, they can modify the value.

- **Add Testcase data object**

  Similar to the first update, test case developers should navigate in CarWeaver

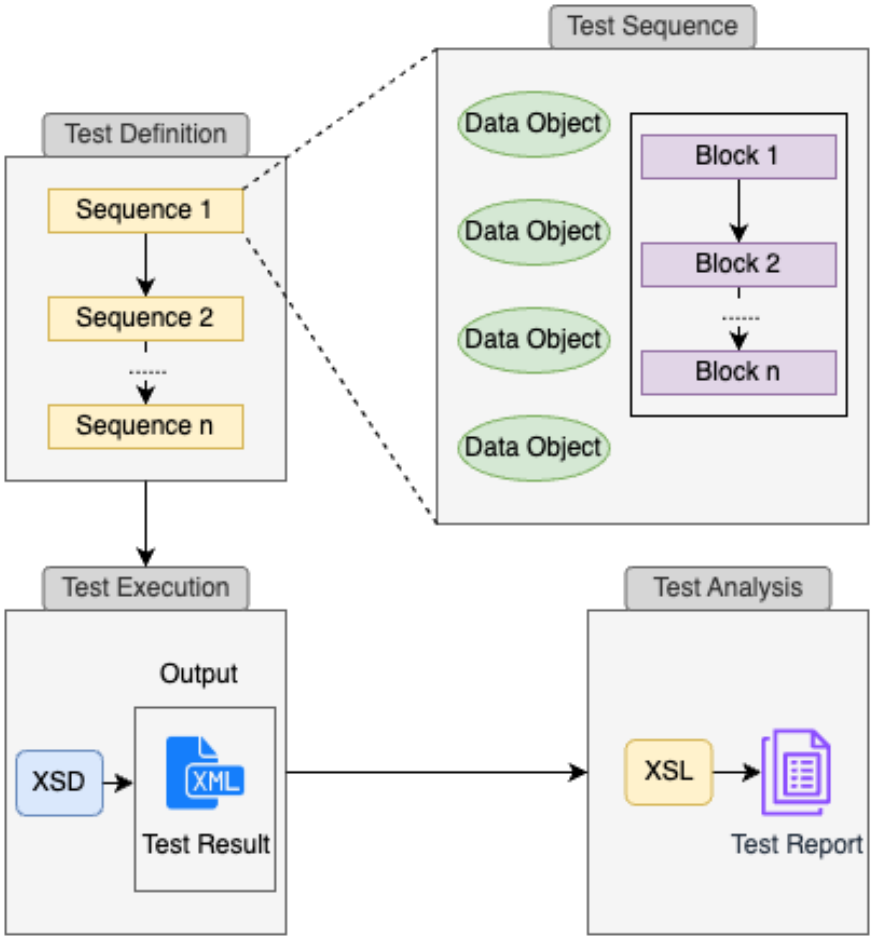


Figure 4.8: An explanation of how the test report is generated in dSPACE AutomationDesk.

to retrieve the relevant test case ID. Subsequently, in AutomationDesk, they need to create a new data object under the test case and input the acquired test case ID.

Its important to note the distinction between the test case ID in CarWeaver and the test case ID generated by AutomationDesk. In AutomationDesk, test cases are saved in blkx files, which are XML files containing definitions of saved or exported AutomationDesk elements such as sequences. Each test case stored in a blkx file is assigned a unique ID automatically by AutomationDesk. However, this ID is unrelated to the test case ID defined in CarWeaver and cannot be used for establishing traceability.

- **Design a new block for data object display and add it in the sequence**

  The author explored various block types available in AutomationDesk to determine the most automated and efficient solution. A summary of these findings is presented in Table 4.1.

  Since the requirement ID is a data object of the test sequence, it's logical

# Attempt	Block Type	Adopted
1	AddDataObjects	No
2	AddText	No
3	Exec	Yes

Table 4.1: Various blocks that were explored during implementation phase.

to drag a data object block called AddDataObjects from the built-in Report library to the sequence flow. This block acts as a container for a data object and can show its information. To connect it to the corresponding requirement, a new data object should be created within this block, and its reference field should be linked to the requirement data object in the sequence. In this way, the requirement information is integrated into the automation sequence and should also appear in the report. Following the execution of the updated sequence, the report successfully displayed the requirement ID. However, it was presented in a table format, which is not ideal for a single value and complicates the extraction of the requirement ID for the traceability table. An example of this table format, showing only the requirement information, is depicted in Figure 4.9.

Variable	Value
String	REQ-dummy requirement ID

Figure 4.9: The table automatically inserted into the test report after adding a data object block in the automation sequence. The block is connected to the requirement data object of the test sequence.

Inspired by the data object block, the author considered replacing the block with another type of block, which is the AddText block. It's also a built-in block type provided by AutomatinDesk itself. It's a container of several data objects. By default, it will print out in the test report the string specified by users if it's included in the sequence. After dragging the AddText block into the test case sequence, it's necessary to edit the value field to include requirement information.

The author modified the field to "CarWeaver Requirement: example requirement id" and executed the test case. As anticipated, the report correctly displayed the added text. However, in practical scenarios, testers must manually enter "CarWeaver Requirement: " followed by the value of the related Requirement data object. This manual entry process can lead to errors, as any typographical mistake or omission in "CarWeaver Requirement: " could cause parsing failures.

A more automated procedure should be developed to connect and enter data automatically, reducing the need for manual edits by testers and lowering the risk of errors.

The optimal solution for maximizing automation is to use the Exec block, which works like a script by embedding Python code. AutomationDesk in-

cludes a built-in Python environment and provides various Python libraries. These tools help execute code within the Exec block and integrate it with testing processes. In this iteration, the code retrieves data from the Requirement and Testcase data objects of a test sequence and logs this information. The details are then displayed in the XML test result.

This block can be inserted at any point in the test sequence, though placing it at the beginning is often most convenient.

COMP 1-2 Test report in AutomationDesk

A traceability table in the test report can offer a clear and concise overview that allows testers to easily see the relationships between test cases, requirements, and test verdicts. Minimizing the effort required to update test cases will decrease manual labor and lower the likelihood of errors for testers.

The software testing process generally follows a sequence of steps, beginning with test definition, followed by test execution, and concluding with test analysis. In the case company, when the predefined HIL test cases are executed, an XML file known as ReportPool.xml is created. This file is a comprehensive repository of the test results, which includes details such as runtime logs, metadata related to test cases, the verdicts of the tests, and so on. The ReportPool.xml is integral to the testing process as it serves both as the output from the test execution phase and the input for the test analysis phase. Based on the information contained in this XML file, a customized test report is generated, providing a tailored overview of the testing outcomes.

To incorporate a traceability table into the test report, reverse engineering techniques were employed to devise a method for its generation. The placement of this table in the test report, derived from the results contained in the ReportPool.xml file, necessitates that relevant data be included as part of this file. The data encapsulated within the ReportPool.xml originates from the execution phase and directly references the test sequences under test.

Therefore, including traceability information in the test results is only feasible if the test sequences are configured to support this functionality. This realization led to the proposal of introducing a new block within each test sequence that records the requirement ID and test case ID as part of the test results. Given that the existing test sequence definitions already incorporate a data object named Requirement, it is straightforward to introduce an additional data object named Testcase. This new object acts as an attribute and represents the test case ID defined in CarWeaver, which corresponds to the AutomationDesk test sequence.

For the XSL processing to accurately identify and retrieve the elements that store the requirement ID and test case ID within the test results, it is imperative that the logging details inputted into the ReportPool.xml are distinctive. The implemented solution involves specifically logging entries such as "CarWeaver Test Case ID: example-test-case-id" and "CarWeaver Requirement ID: example-requirement-id".

When the test result was prepared, the author proceeded to define the elements of the table and the styling attributes within the XSL file. The Requirement ID and test case ID are extracted from the Reportpool.xml file and then recorded in the report with the help of the XSL file. This setup can generate the traceability table in the report, which is a visualization of the link between test cases, requirements, and their respective test results.

COMP 1-3 JSON test result in AutomationDesk

Although test results are accessible in the ReportPool.xml file and are detailed further in a test report for better visibility, they lack data that is directly usable for status analysis. Therefore, it is necessary to develop a method to produce data ready for analysis, such as JSON. This would involve converting the test results from AutomationDesk into JSON format, which supports requirement status assessments. The JSON data should include essential elements like the test case name, test case ID in CarWeaver, requirement ID in CarWeaver, and the test verdict.

There are two methods to initiate HIL testing in AutomationDesk. The first method is to manually execute the test sequences directly within AutomationDesk. The second method uses a Jenkins pipeline to trigger testing on a HIL machine by calling the AutomationDesk API. In both scenarios, the execution results must be generated as JSON data, regardless of the testing trigger method.

When testers manually initiate testing directly in AutomationDesk, the exportation of JSON data can be implemented through a GUI tool. This tool reads the latest test results and generates a JSON file. The design choice of integrating a GUI tool within AutomationDesk, accessible via a simple click rather than through script execution, is intended to streamline the process. This approach simplifies the operation by eliminating the need for explicit script invocation.

The GUI tool was developed primarily using the Tkinter library in Python. When a user clicks the GUI icon within AutomationDesk, it triggers a Python script. This script defines a class to identify the active project in AutomationDesk and access the default folder containing all associated test results. Each test run's results are stored in a separate folder. To generate JSON data, the script searches these folders for the most recent ReportPool.xml file and selects it for processing.

The ReportPool.xml file is parsed to extract the test case name, test case ID, requirement ID, and verdict ID, which are then stored in a JSON file. The GUI tool does more than merely execute the script; it also displays messages related to the script's execution. If there is a failure in execution for any reason, the GUI provides messages that assist in troubleshooting. Additionally, if the JSON data is successfully exported, the GUI indicates the storage location of the JSON results and offers a button that allows users to instantly access and review these results.

HIL testing can be integrated into the Jenkins pipeline by leveraging the AutomationDesk API. This integration is possible because the HIL machine is connected to a computer equipped with the AutomationDesk software. Through this setup, the API is able to interact with the computer to initiate HIL testing. By default, the results of these tests are stored on the same computer where the HIL testing is

conducted.

The process of generating JSON data from the test results via Jenkins is similar to the interaction between the AutomationDesk GUI tool and the key Python script. This involves creating an instance of the class defined in the script and then calling its methods to execute functions.

In the case study company, a series of Jenkins pipelines are pre-configured to perform HIL testing for selected projects, software versions, testing levels and so on. These pipelines are set up with numerous parameters that define the runtime environment and determine which code repositories are accessed. To start HIL testing through Jenkins, users must initiate a Jenkins job by building it with these parameters. The parameters are understandable by the pipeline script, which then triggers a batch file. This batch file, in turn, calls a Python script to interact with AutomationDesk and subsequently saves the test results in JSON format. This Python script is the same one used by the AutomationDesk GUI tool, as mentioned earlier.

The JSON file that records the test outcomes is stored on a remote HIL computer that hosts the AutomationDesk software. For these files to be accessible when tests are initiated via Jenkins, they need to be exposed as Jenkins artifacts. Jenkins artifacts are files that are created as part of the execution process of a Jenkins build job. After the test results, including the ReportPool.xml and JSON file, are produced on the HIL computer, they are transferred to the Jenkins workspace. The Jenkins workspace is essentially the directory on the Jenkins agent where the specific job's build activities are conducted. This space serves as the hub for all build-related operations, such as checking out source code, compiling code, executing scripts, and creating artifacts. The necessary files are then included as Jenkins artifacts in the pipeline script. This integration ensures that upon the successful completion of a Jenkins job, all related artifacts can be viewed and accessed through the Jenkins website.

The GUI messages and their corresponding scenarios are detailed below.

1. The JSON data is successfully generated and saved to a file. As shown in Figure 4.10, the GUI displays messages indicating the paths to the latest test results and the newly created JSON file, which contains traceability information. Additionally, the pop-up includes a button that allows users to directly open the JSON file with a single click.

2. If no project is open or active in AutomationDesk, the GUI will display a message, as illustrated in Figure 4.11, indicating this status. Under these circumstances, the JSON data cannot be generated because the script is unable to locate the folder containing the original test results. To resolve this issue, either open the project under test or click on an already open project folder to activate it.

3. If the execution result lacks a "ReportPool.xml" in the result folder, the GUI will display a message indicating this absence, as shown in Figure 4.12. Without the original test result, the JSON data cannot be generated. Users must conduct a test to produce the corresponding JSON file.

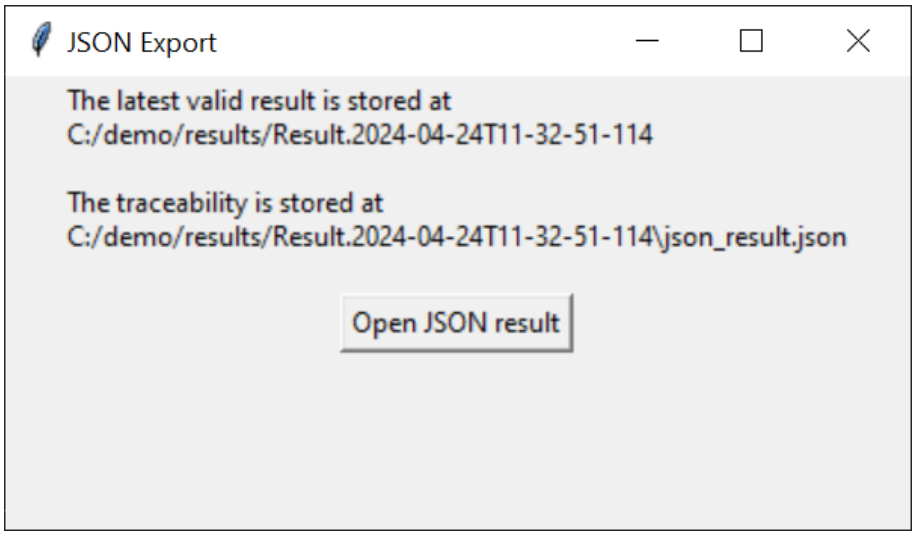


Figure 4.10: The GUI message shown when the JSON file is successfully generated.

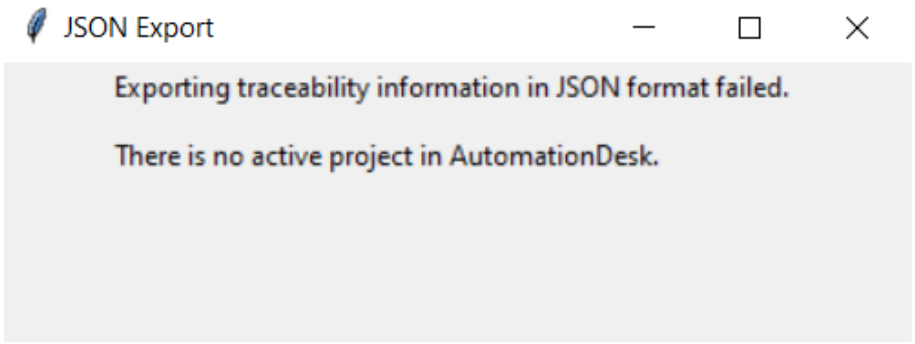


Figure 4.11: The GUI message shown when there is no active project.

COMP 2 CarWeaver API usage

As previously mentioned, the requirement status can be analyzed by comparing test results with the requirements specification, provided that both are stored in a compatible format, such as JSON. Although the requirements are managed using the software CarWeaver, it is necessary to retrieve the related requirements and store them in JSON format.

To automate data retrieval and comparison, particularly within the Jenkins-enabled automated testing workflow, it is essential to gather specific requirement data from CarWeaver. The author designed a new Jenkins build parameter that accepts the resource ID of an item in CarWeaver. This item is organized hierarchically in CarWeaver and represents a specific scope. Ideally, all requirements defined within this scope should be accessible.

This new build parameter is then converted into a format that can be understood as an endpoint for the CarWeaver Application Programming Interface (API).

An API allows different software applications to interact with each other. It serves as a bridge, enabling one application to access the services or data of another securely.

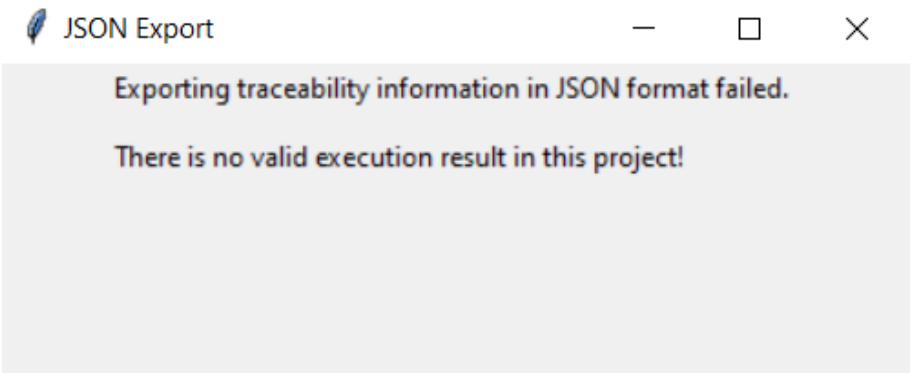


Figure 4.12: The GUI message shown when no valid execution results are available.

In API terminology, an 'endpoint' is a specific URL (Uniform Resource Locator) where an API can be accessed by a client application. It denotes a specific function or resource that is available within the API. APIs can adhere to several protocols and architectures, including REST (Representational State Transfer), SOAP (Simple Object Access Protocol), and GraphQL. For security, many APIs, particularly those utilizing OAuth, require both a client key and a client secret to ensure that interactions between client applications and API services are authorized. The client key identifies the application making the request, while the client secret verifies that the request originates from a legitimate source.

CarWeaver's RESTful APIs are developed by the case company. To access the API, users must obtain a client key and a client secret from the API management team. When a request is made to an endpoint with the proper authentication, the API sends back data in JSON format.

In this iteration, the implementation of the framework focused on the requirements in the "Specific Item Test" scope within CarWeaver. The data to be gathered included all test cases in this scope and their related requirements. However, after detailed exploration and investigation, it became clear that there was no existing endpoint capable of retrieving the required information. There were also no alternatives, like using endpoints from other scopes, to obtain the data. Therefore, in this iteration, a dummy endpoint and data were used as placeholders to ensure the framework operated properly.

In the first iteration, the code that parses the CarWeaver ID and utilizes the CarWeaver API was integrated into the script interacting with AutomationDesk and executed on the HIL computer. To securely store the API key, it was set as an environment variable on the HIL machine. This prevents the key from being directly exposed in the code and accessible to anyone who can view the codebase.

COMP 3-1 JSON requirement specification

In the thesis work, the data fetched via CarWeaver API is then organized and adjusted for project analysis. The revised data structure can be outlined as shown below.

```
{
    "specific_item_test_name": "Dummy component under test",
    "specific_item_test_id": "Dummy value",
    "traceability": [
        {
            "carweaver_test_case_name": "dummy test case 1",
            "carweaver_test_case_id": "TC-123456",
            "systemweaver_id": "Dummy value",
            "requirements": [
                {
                    "requirement_id": "REQ-456789",
                    "systemweaver_id": "Dummy value"
                },
                {
                    "requirement_id": "REQ-124125",
                    "systemweaver_id": "Dummy value"
                }
            ]
        }
    ]
}
```


COMP 3-2 Consistency checking

This component aims to check for inconsistencies between the test specifications in AutomationDesk and the requirements in CarWeaver by comparing two JSON data files that store the necessary information. The requirements data is used as the reference, and a Python script is employed for the comparison. Inconsistencies will be reported in JSON format. An example of the results of inconsistency checking can be found below.


```
{
    "Requirement data objects in AD conflict with CW": {
        "Test case": "TC4",
        "Requirements in AD": ["REQ-3", "REQ-5", "REQ-9"],
        "Requirements in CW": ["REQ-3", "REQ-5"]
    },
    "Test cases exist in AD but not CW": ["TC-not-in-CW"],
    "Missing test case in AD": ["TC6", "TC7", "TC8"],
    "Duplicated test cases in AD": ["TC1"]
}
```

COMP 3-3 Project status

The project's status is assessed using a Python script. This script constructs two dictionaries: one stores requirement information via the CarWeaver API, and the other gathers test results from HIL testing executions. The first dictionary maps requirements to their associated test cases, while the second associates test cases with their outcomes. Both dictionaries are populated from JSON files described in previous components.

The analysis compares data from CarWeaver with test results. The script cycles through each requirement and its linked test cases in the first dictionary, checking if each test case has been executed and whether any have failed. Based on these findings, it updates the requirement's status accordingly. If all associated test cases are tested and passed, the requirement is marked as "fully_validated_passed." If all are tested but any fail, it's marked as "fully_validated_failed." Requirements with some, but not all, test cases tested are labeled "partly_validated," and those with no tests conducted are designated as "not_tested."

COMP 4 Data visualization in Power BI

When the JSON data containing test results and requirements is prepared, it can be processed and analyzed using Python. This analysis determines the status of each requirement. The results are then saved in a CSV file, which can be visualized using Power BI. A sample of the CSV file after processing is shown in Table 4.2. The first column is the requirement ID in CarWeaver and the second column is the requirement status.

Requirement	Status
Req-test-1	Fully Validated and Passed
Req-test-2	Fully Validated but Failed
Req-dummy-3	Partly Validated
Req-dummy-4	Not Tested
Another-req-ID	Fully Validated but Failed
Req-6	Not Tested
Req-7	Partly Validated

Table 4.2: The analysis result of requirement status.

The steps to visualize the data in Power BI Desktop are listed below.

1. Download the requirement_status.csv file from Jenkins artifacts.The automated HIL testing is triggered in the Jenkins pipeline, and the project's status CSV file is stored as a Jenkins artifact.
2. Open Power BI Desktop.
3. Open the requirement_status.csv file downloaded in the first step. Transform the data as necessary to correctly handle the headers.
4. Create a visualization, such as a pie chart, to represent the data.

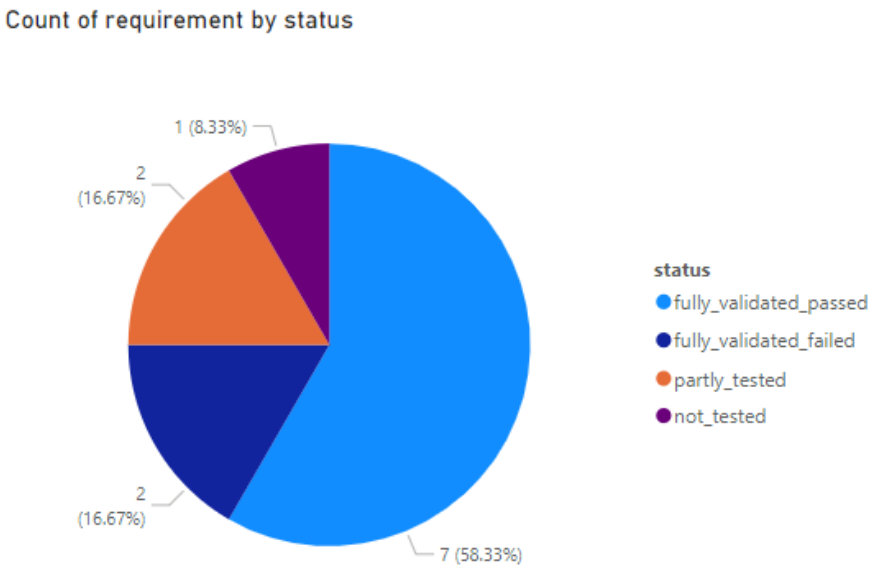


Figure 4.13: A pie chart in Power BI that visualizes the status of the requirements.

Figure 4.13 provides a Power BI visual representation of the project requirements. The pie chart shows that 58.33% of the requirements are fully validated and have passed, 16.67% are fully validated but failed, another 16.67% are partially tested, and 8.33% are not tested. Thus, the overall status of the project can be classified as partly tested.

4.1.5 Implementation Evaluation

The evaluation of the first iteration was performed through several activities, as shown in Figure 4.14.

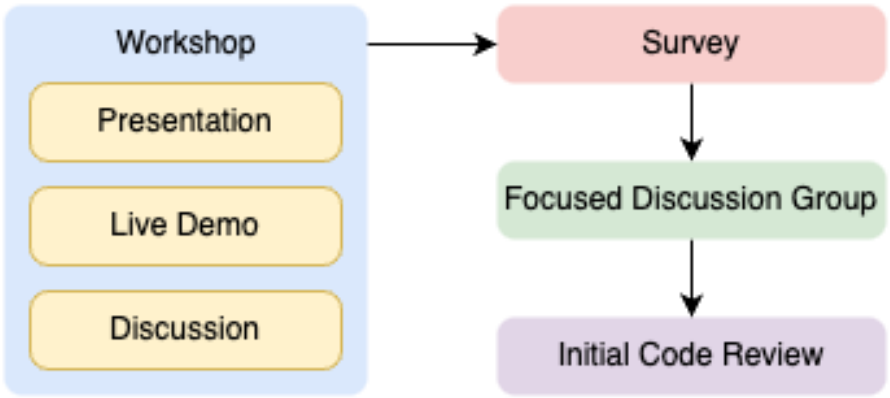


Figure 4.14: The evaluation activities performed in the first iteration.

To effectively evaluate the initial results of the first iteration, the author organized a workshop attended by 15 key stakeholders from various departments within the company. This ensured a wide range of perspectives. The workshop began with an

informative introduction to the project's main topic, followed by a presentation of newly developed components, including the strategic reasons for their design and expected impacts.

The session then transitioned into a live demo, during which the author demonstrated how these artifacts could be integrated into daily operations, focusing on their functionality and potential benefits. During this interactive session, a software developer from the HIL automation team actively participated and offered several constructive suggestions. These were particularly valuable as they reflected experience with software safety and HIL operations via AutomationDesk.

Recognizing the importance of this feedback, the author arranged a focused discussion group with the software developer after the workshop to explore the feasibility of these suggestions and any potential enhancements. This smaller group setting also included three additional stakeholders who provided essential insights into how requirement management is applied across various release teams. The discussions highlighted the challenges and inconsistencies in using the CarWeaver tool within the company, identifying areas for further improvement.

After the workshop, a detailed survey was sent to the participants to gather more detailed feedback, focusing on the functionality and usability of the artifacts. This helped gauge the effectiveness of artifacts from the users' perspective and pinpoint opportunities for improvement.

An initial code review session was conducted. While the review focused on code details that may vary significantly when the framework is used in different companies, some comments still provided valuable insights for improving overall solution design.

Activity 1 Workshop

Feedback was gathered during the workshop. Some of this feedback was followed by brief discussions to quickly assess the feasibility of the suggestions.

One developer noted that COMP 1-2, the traceability table, has a good layout. However, they suggested that the process for updating test sequences (COMP 1-1) could be more automated.

> Feedback 1: I really like the layout of the test report. The requirement traceability table looks very nice and would be highly appreciated. We could add a Testcase field in the template for test cases, so testers don't need to do it manually.

In the case company, a template has been developed to create new test cases like a scaffold. It defines the required data objects, such as Requirement, Responsible (the tester responsible for this test case), and the basic skeleton of the test cases. The feedback suggests adding a Testcase data object to the template, so testers don't need to manually create and rename a new data object to Testcase. This change can make the process more automated.

Another piece of feedback was about the Jenkins parameter value. The automated HIL testing can be triggered in the Jenkins pipeline, where the ID of an item in

CarWeaver can also be given as a build parameter. The feedback indicated that the ID itself is not as readable as the item name.

> Feedback 2: I think we should have a readable link to CarWeaver. Internal CarWeaver IDs are very difficult to interpret. There should be a link to CarWeaver in a human-readable format so that people can double-check and verify its accuracy.

Another participant argued that this is not very straightforward to implement, as items in CarWeaver might experience version changes while keeping the same name. In this situation, using just the item name can be problematic for identifying the correct resource.

> We tried using the name itself, but we observed that while the name remains the same, the version of the instance can change. This means we may not have the latest information from CarWeaver. Currently, we are using the ID. We'll try to figure out a way to use a generic name that provides information to all users while also fetching the latest status from CarWeaver.

Activity 2 Survey The survey aimed to evaluate the usability and functionality of the framework and determine whether it can address the department's inability to check consistency. The components of the framework were assessed using a 6-point Likert scale, which differs slightly from the typical 5-point scale by including an "I don't know" option. This addition was necessary because the framework integrates various software tools, and some stakeholders in the workshop are only familiar with one or two of them. For instance, one participant works exclusively with software delivery, while another focuses solely on building automation workflows in AutomationDesk.

The survey received nine responses from the workshop participants. Six questions were asked to evaluate the functionality of the framework using a Likert scale. The results are shown in Figure 4.15. The feedback indicates a largely positive reception of the framework's features, especially in its ability to provide detailed and useful traceability and visualization outputs. The GUI tool for generating JSON data, which implements COMP 1-3 in the framework, was generally found useful. However, a subset of users found the visualizations derived from CSV files (COMP 3-3 and COMP 4 in the framework) to be less helpful. This suggests a need for more tailored or adjustable visualization options. The neutral and negative responses regarding the GUI tool and JSON output indicate a need for more intuitive or detailed presentations to explain their benefits.

The usability of the framework was assessed with six questions, focusing on the ease and efficiency of using its key components and overall user satisfaction. A bar chart of the evaluation results can be found in Figure 4.16. Most respondents appreciate the ease of accessing traceability information and the helpfulness of the new GUI in AutomationDesk, indicating these aspects are well-received.Many participants expressed a clear intention to continue using the GUI tool for future projects, and the convenience of building visualizations in Power BI was appreciated, suggesting effec-

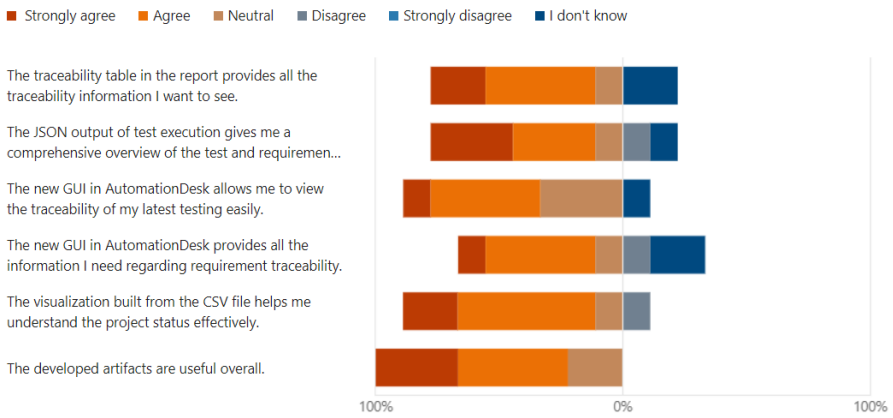


Figure 4.15: Results of the functionality evaluation of the framework.

tive integration and functionality. While the feedback on checking JSON and CSV artifacts in Jenkins was less positive, with some neutral responses, this highlights an opportunity for further clarification and improvement to fully leverage these features. Overall, the responses lean towards a positive opinion of the framework's usability.

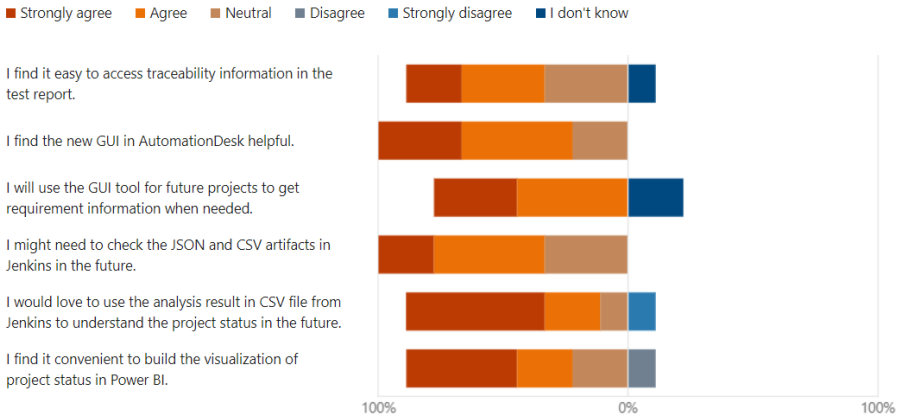


Figure 4.16: Results of the usability evaluation of the framework.

The study was performed in the integration and release team and focused on the software qualification testing level. However the framework can be applied to the broader context. Therefore, the workshop included a larger group, encompassing other teams and departments, to evaluate the applicability and generalization of the framework within the company. The pie chart in Figure 4.17 shows the results, illustrating respondents' opinions on the potential for extending the framework to other departments and testing levels. The overall feedback is positive, which indicates that the framework can bring benefits beyond the case team.

The framework was also evaluated on its ability to address the missing strategies of software qualification testing in the case department. These missing components were described in the problem investigation of the first iteration and can be categorized by the base practices towards software qualification testing according to ASPICE. The main issues include the lack of test specifications, methods to ensure

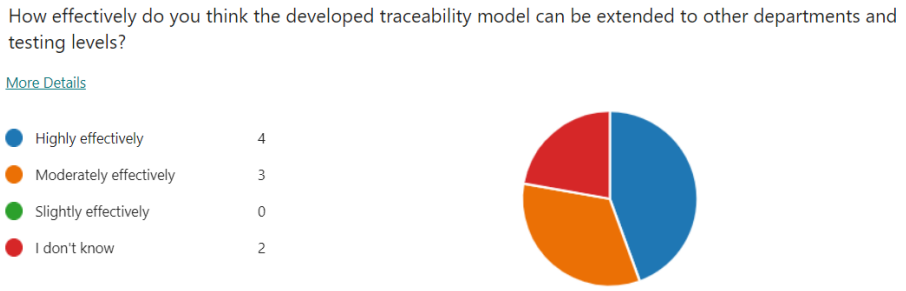


Figure 4.17: Results of the applicability evaluation of the framework.

consistency between requirements and test specifications, and methods to summarize and communicate test results. While the framework does not address the creation of test specifications, it helps with establishing traceability and consistency checking. The question to evaluate this aspect was straightforward, and the results are shown in Figure 4.18. Most respondents expressed strong confidence that the framework can address these gaps.

Figure 4.18: Results of the evaluation on if the framework can address the missing strategies in software qualification testing in the case department.

Activity 3 Focused Discussion Group

In the designed framework, the processing center directly interacts with various software, including CarWeaver and AutomationDesk. However, during the focused discussion group, the concept of a message bus was suggested as an alternative approach for managing data transmission.

The message bus is a generic software architecture pattern that allows different systems to share state information without needing to interface directly. It decouples the system components, enabling them to communicate via messages instead of direct calls, thereby reducing dependencies. This architecture also makes the scaling of applications easier.

In the case company, a message bus has been implemented. The key components of the message bus are Activity, Task, and Product. For example, in HIL testing, one test run may include several test cases (test sequences). When HIL testing is initiated, it is referred to as an Activity. The execution of each test case is termed a Task. Any artifact generated during the process, such as a test report, is called

a Product. The entire execution of HIL testing is represented using these three concepts and then uploaded to the message bus. Although the message bus covers a wide range of systems and software, this study focuses on the test execution activities within AutomationDesk.

One participant in the discussion group suggested using the message bus for project status analysis.

> If you have already developed a way to transfer requirement information via MessageBus, it is by far the easiest and best solution since we don't need to install any library. We just need to write a new adapter to the existing code base that interacts with the message bus. We can just read out the requirement ID, and test case ID and send it on the bus.

However, at the time the study was conducted, the in-house MessageBus was not ready for use to build the proposed framework. The processing center requires requirement data and test results to compare and analyze project status. Currently, the MessageBus neither stores the requirement information itself nor provides a method to fetch requirements from CarWeaver for a specific scope. Therefore, it can't be used at this moment.

If this issue is resolved in the future, requirement data can be retrieved from the MessageBus instead of directly from CarWeaver. To access information sent on the MessageBus, an Elastic Search database is available. This database stores all data sent via the MessageBus and can be queried by external consumers. However, in this iteration, implementing this feedback is not feasible.

The discussion of the MessageBus also proved the applicability and efficiency of the framework. Since it doesn't require advanced architectural design, it can be adopted easily.

During the focused discussion group, two developers from another software release team raised concerns about the versioning of CarWeaver items. In CarWeaver, an item such as a requirement may undergo several version updates. For example, once a requirement is released, it cannot be modified again. The developers questioned whether the CarWeaver API could correctly handle these version differences. After discussing, it was clear this was not an issue, as the endpoint used is specified with the unique ID of a CarWeaver item, and the latest version is used by default. This discussion confirmed that the framework functions well in a dynamically changing environment.

Another topic discussed was a suggestion by one participant to include the CarWeaver ID of the "Specific Item Test" as a new data object of the folder being tested in AutomationDesk, rather than providing it in the Jenkins pipeline. However, this suggestion was disapproved. The folder being tested contains multiple test cases and is designed to test a specific requirement scope in CarWeaver, which may change over time. It is the responsibility of the testers, who execute automated HIL testing rather than develop HIL test cases, to ensure that the test cases and requirements are up to date. This argument demonstrated that, although a central processing unit is used in the framework, the design remains decoupled, with responsibilities

clearly distributed.

Activity 4 Initial Code Review

The initial code review was conducted with a developer from the team responsible for maintaining the codebase on which the framework implementation is based.

Several Python scripts were developed to build the framework. Although these scripts adhered to PEP8 guidelines, which provide best practices for writing Python code, and included detailed docstrings, there is room for improvement to enhance developer understanding. Specifically, adding an argument parser to key scripts would allow for easier parameter checking in the terminal.

During the development of the framework, several files used solely for testing data storage were placed in the same folder as the key script. This practice is not ideal as it mixes test files with production code, making them harder to manage. It is recommended to separate test data into its own folder to maintain a clean and organized structure. Additionally, placing hardcoded strings into global constants is advised, as these values may need to be changed in the future. These coding improvements promote a cleaner implementation of the framework, which is beneficial for ongoing maintenance.

Another issue identified during the code review was that the execution of test result transformation and the use of CarWeaver should not occur in the key script where HIL testing is prepared and triggered. To address this, distributed storage should be considered to isolate code for different purposes and decouple the implementation. The problem investigation in the second iteration will explore this further.

The code review during this evaluation stage also highlighted security issues related to using the CarWeaver API and emphasized the importance of maintaining a uniform coding style across the codebase. This review examined the implementation structure and details of the framework, making it more robust and easier to maintain. Better solutions were suggested during this session, leading to overall improvements in the framework design.

4.2 Cycle II Findings

The previous iteration implemented most of the components in the framework, so the second iteration was lightweight and focused more on refining and improving the framework.

4.2.1 Problem Investigation

The investigation phase of the second iteration significantly overlapped with the evaluation of the first iteration. Both the focused group discussion and the initial code review from the previous cycle examined the implementation details of the framework and identified possible improvements in usability. In this iteration, the author assessed this feedback to evaluate the feasibility and potential impact of these

improvements on the framework. The investigation activities in this iteration are classified as Impact-Driven Investigation.

A primary issue identified was the use of dummy data for requirements and the dummy CarWeaver API endpoint. The author had intentionally used these placeholders in the first iteration and was aware of their limitations. However, relying on artificial data restricts the ability to fully evaluate the framework's efficiency, as certain issues may remain undetected without authentic requirement data.

During the code review session, another issue was identified in collaboration with the reviewer: the HIL computer is not suitable for generating JSON data and interfacing with CarWeaver. In the Jenkins pipeline, which triggers automated HIL testing, the stages related to HIL are executed on a specialized computer. This machine, configured exclusively for HIL testing, has the dSPACE toolchainincluding AutomationDesk and ControlDesk installed. Resources for HIL testing are costly, and several codebases are configured on this machine to meet specific company requirements.

Given the complexity and safety-critical nature of the settings on such computers, performing unrelated tasks on them can be problematic due to the potential for unintended disruptions. In the case company, Volvo Car Corporation, multiple HIL computers are used. Each is connected to different test objects like Electronic Control Units (ECUs) to cater to varied testing needs and hardware specifications. In the automotive industry, where software development and release are frequent and critical activities, a reliable HIL testing environment is essential. Any disruption in the HIL environment could significantly impact testing tasks and consequently slow down the entire software development and release process.

In the first iteration, the framework was fully implemented on the HIL computer following the execution of a Jenkins job. The necessary code was integrated into a script that configures the HIL environment according to Jenkins parameters. This script also initiates testing, logs activities, and concludes the test sequence. Additionally, it manages Git operations. Interactions with CarWeaver also occur on the HIL computer.

To run the API utilization code effectively, several Python libraries must be installed globally. This installation could lead to conflicts with other libraries already in use for HIL testing, potentially causing dependency issues. Typically, virtual environments would resolve this by providing isolated workspaces for different projects, a common practice in software development. However, the use of virtual environments is prohibited on the HIL computer due to restrictions with the Python interpreter used in the dSPACE system. Additionally, the client secret required from the API management team is set up as a system environment variable on HIL computers manually. Since there are many HIL computers, this secret must be individually configured on each one to support all types of HIL testing. This method is not only labor-intensive but also error-prone. Moreover, manually handling this process across numerous computers increases the risk of the client secret being exposed. Therefore, the API utilization code should not be run directly on the HIL computers to prevent potential security issues.

4.2.2 Solution Design

The first issue is the absence of a real endpoint. This can be resolved by contacting the API development team. Implementing a new endpoint is beyond the scope of this study, as there is no access to the CarWeaver database. Therefore, retrieving data is impossible without support from the API development team.

Regarding the second issue, the utilization of the CarWeaver API and the analysis of project status should be conducted on a different Jenkins node, rather than on the HIL computers. A Jenkins node is part of a distributed build environment in Jenkins, where actual job executions occur. Typically, companies use multiple Jenkins nodes for various purposes. Components of this framework should be implemented in a non-safety-critical environment.

4.2.3 Design Validation

Using real data through an API allows the framework to test if the expected data schema can be successfully built. It also helps identify any issues during API utilization, such as authentication problems, data access issues, and rate limiting.

Running the API utilization code on a separate node can prevent disruptions in the HIL environment. Since the client key and secret are stored as Jenkins credentials and can be used in the pipeline script, sensitive data is protected.

4.2.4 Solution Implementation

The author discussed the first problem with a CarWeaver API developer to request a new endpoint. This endpoint retrieves all test case IDs associated with an item named "Specific Item Test" in CarWeaver. Currently, there is an endpoint that accepts a test case ID and returns all related requirements that can be partially validated by the test case. The script managing API utilization runs a loop to gather all requirements for the test cases under "Specific Item Test." A third endpoint retrieves all test cases linked to a specific requirement. This process is currently cumbersome but could be simplified in the future with the development of more efficient endpoints.

To address the second problem, the Jenkins pipeline script used to trigger automated HIL testing should be modified to use a different node for CarWeaver API utilization and project status analysis. This node should not be used for HIL testing. The client key, configured as a Jenkins credential, can be safely used in the script with proper permission settings.

The project analysis is based on requirement data from CarWeaver and test results from AutomationDesk. The test results are generated and stored directly on the HIL computer. While CarWeaver API calls can be made independently of any other codebase, fetching the test results requires additional effort as they reside on another computer. However, this is feasible. The HIL testing results, including the key file Reportpool.xml, will be uploaded to JFrog Artifactory. They will be stored

in a folder named using the Jenkins pipeline name and the build ID. Therefore, the Reportpool.xml file can be fetched using the Artifactory API.

4.2.5 Implementation Evaluation

In this iteration, an evaluation meeting was held with the system engineer, who is also the industrial supervisor for this masters thesis project.

During the meeting, the author first demonstrated the diagram of the designed framework. Previously, the evaluation presented the components separately to show how they functioned within the case company. However, in this final evaluation meeting, the whole framework was displayed. Since the functionality and usability of the framework within the case company had already been well evaluated, this iteration focused more on implementation updates and the framework's capability in external contexts.

The update on moving the CarWeaver utilization code execution to a non-critical Jenkins node was well received. The idea of using Jenkins credentials for storing the client key and client secret was considered secure.

The system engineer, with extensive knowledge across different departments in the company, discussed the potential application of the framework to other departments. HIL testing and requirement management are uniformly conducted using AutomationDesk and CarWeaver. Although different departments might record test results differently, they use the same Excel template for traceability, which is uploaded to a web portal to build visualization, as described in the problem investigation of the first iteration. The framework provided a more standardized method within the company for running automated HIL testing and collecting test results.

Although the framework has not been evaluated at other companies, the system engineer agreed on its applicability: as long as the software used in other companies can export the required data, the framework can be implemented.

The performance of using the CarWeaver API was also evaluated. The current endpoint design does not support directly fetching the desired requirement information, so the data processing stage includes a loop to make RESTful requests. When dealing with a large scope of requirements, it takes some time to complete the multiple API calls. In future work, a cache design could be considered to store and update requirement information. For example, JFrog Artifactory could be used to communicate with CarWeaver to fetch and store requirement information of various scopes. This way, project analysis in the framework would not need to use the CarWeaver API every time; it could retrieve data from Artifactory since requirements do not change frequently. As the test case automation is not yet complete, the design details cannot be finalized. However, this feedback is valuable for future consideration.

Overall, the evaluation in this stage proved that the designed framework can be effectively applied in different contexts.

5

Discussion

5.1 Revisiting the Research Questions

Research Question 1 What is the problem with the current status of requirement traceability in the software build and automated test workflow?

The current workflow for building traceability applies only to manual HIL testing conducted using dSPACE ControlDesk. After executing the tests, testers manually record the outcomes in an Excel template, which doesn't have associated requirement information. To build requirement traceability, testers must fill out a separate Excel template that incorporates requirements from CarWeaver with the test results. This template is then uploaded to a web portal that processes the data to create visualizations. The process necessitates downloading the most current version of the template each time it is used, which is a cumbersome task. Additionally, the HIL testing reports do not include traceability information.

The manual HIL testing will gradually be replaced by automated testing using dSPACE AutomationDesk, even though the automation process is still in its early stages. Test cases will be automated, but there is no effective system to build traceability for them automatically. Continuing to use old manual methods for tracking is not efficient.

There is a significant inconsistency between the requirement management tool CarWeaver and many of the developed automated HIL test cases. Each HIL test case should be linked to a 'Requirement' data object, which must match the corresponding requirement ID in CarWeaver. The case company adheres to the ASPICE standard, which also stresses the importance of consistency. However, even the current manual workflow lacks a method to validate this consistency.

The complete absence of a workflow to establish traceability and ensure consistency results in a lack of requirement visualization for automated HIL testing.

Research Question 2 What improvements in test reporting and visualization are sought by stakeholders?

Stakeholders require direct access to key traceability information in the report, including the test case name, test case ID in CarWeaver, corresponding requirement ID in CarWeaver, and the test verdict. The clearest and most immediate way to display this information is through an additional table. Furthermore, it is recom-

mended to integrate this visualization into an existing dashboard, using Power BI as the software of choice. This visualization should reveal the status of project requirements under test, such as the number of requirements that are fully validated or not tested.

Research Question 2.1 How should traceability information be integrated into the HIL testing report?

The HIL testing report generated from dSPACE AutomationDesk is formatted as a PDF in the case company. It is divided into three main sections: The report header, which offers general information such as the test environment and the count of passed, failed, or errored test cases; the test overview, which presents a color-coded summary of test results, simply indicating the verdict of each test case; and the execution details, which provide comprehensive output from the test execution, including results from various test steps and verbose logging.

Traceability information is presented in the header section of the report, formatted as a table with four columns: the test case name, test case ID in CarWeaver, corresponding requirement ID in CarWeaver, and the test verdict.

To generate the table, each test case must be updated with a new data object named "Testcase". A developed block designed to append data object information to the logs should be added to all test sequences. Therefore, the test result file, which is in XML format, will include both requirement and test case information. The test report, derived from this test result file, can be customized to include the traceability table.

Research Question 2.2 In what way would the stakeholders prefer traceability data to get visualized?

The stakeholders are interested in seeing how traceability data can depict the project's status, especially in the context of software integration and release. It is important to ensure that all requirements defined for the project or a specific scope are successfully validated. Therefore, the visualization dashboard should include a diagram, such as a pie chart, that displays the status of these requirements. The chart should categorize requirements into four statuses: Fully Validated and Passed, Fully Validated but Failed, Partly Validated, and Not Tested.

Research Question 3 How can improved traceability in test reports and visualizations enhance the efficiency of software integration and release processes?

The addition of a traceability table to the test report allows users to quickly identify the most problematic requirements. When a defect is detected, this table can immediately show which requirement is affected and which test cases failed. This speeds up the debugging and defect-fixing process, as developers can directly trace back to the problematic requirement and the associated code.

The visualization lets users directly observe the project status on a dashboard, which supports quick identification of issues and enhances confidence in the software release.

The improvements are designed for automated HIL testing. Although it is not currently in use because not all test cases are fully automated, this setup is expected to significantly reduce manual effort and streamline the integration and release processes in the future.

Moreover, the framework built around traceability not only supports visualization but also consistency checking. This helps ensure the quality and reliability of the final software product during the integration and release process. Additionally, the data output from the framework may have further applications depending on the team's needs.

5.2 Contribution to Knowledge

Maro et al. identified 22 challenges related to software traceability in the automotive domain, solving 16 and leaving 6 unresolved [23]. The unresolved challenges include a lack of visualization tools, traceability establishment perceived as an overhead, assessment of traceability, return on investment, manual work, and lack of interchange standards.

The framework developed in this master's thesis addresses the first four of these challenges. It also tackles two partially resolved issues identified by Maro et al. by providing insights and suggestions to address them fully: diverse artifacts and tools and traceability links unused.

Diverse Artifacts and Tools: Tool integration is suggested by Maro et al. as one of the two solutions when different tools are used for different development activities in the whole software development life cycle. Using multiple tools can lead to discrepancies, especially when data is duplicated across systems and updates are only partially applied. This thesis introduces a practical example of tool integration within the automotive industry, highlighting its ability to identify and report inconsistencies.

Traceability Establishment Perceived as an Overhead, and Lack of Proper Visualization and Reporting Tools: Developers often feel disinclined to establish traceability links, seeing little direct benefit. Interviewees from Maro et al.s case study suggested that visualizing connections might encourage them to prioritize traceability. The designed framework includes a traceability table in the test reports that links requirements directly to test cases, enhancing visibility immediately after tests are run. Additionally, project status reports can be formatted for visualization in tools like Power BI, offering developers convenient access to traceability data. The survey and workshop conducted as part of this study confirm the practical benefits of this approach.

Traceability Links Unused, and Assessment of Traceability: This framework also addresses the issue of traceability links being ignored by centralizing data retrieval from various software tools, facilitating project status analysis and visualization. Moreover, it supports consistency checks between test specifications and requirements, helping to evaluate traceability link quality.

Return on Investment: Establishing and maintaining traceability incurs significant costs. Currently, no methods exist to directly measure the benefits of traceability. This master's thesis suggests that traceability can improve the efficiency of pre-delivery testing processes, like smoke testing and acceptance testing. Automating traceability and providing project status data could make the software integration and delivery process more efficient by reducing manual work. Therefore, the solution to this challenge is to explore how traceability implementation can improve the existing workflow, particularly in areas such as human resources, time efficiency, and delivery reliability.

5.3 Contribution to State-of-the-Art

This study contributes to the state of the art in software engineering, particularly in the areas of tool integration and traceability management. While the research was carried out in an automotive company, the developed framework is versatile and can be applied across various industries. A more general representation of the framework designed in this study is illustrated in Figure 5.1.

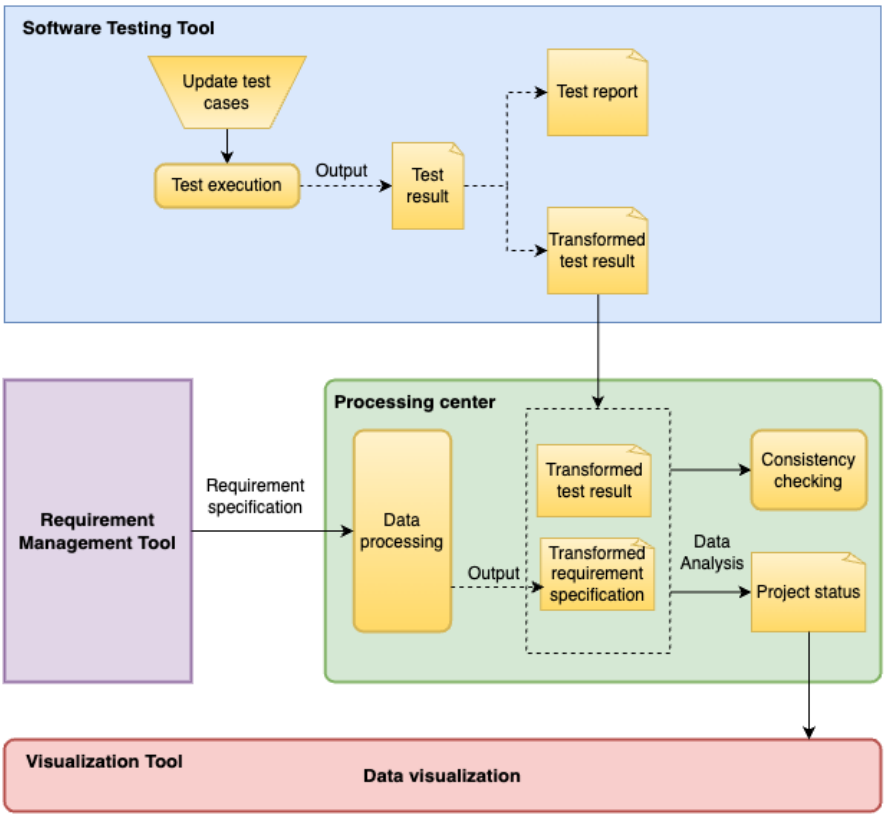


Figure 5.1: A more general representation of the tool integration framework to build requirement traceability and support consistency checking.

Many studies highlight the insufficient support to create requirement traceability and difficulties in ensuring consistency between artifacts [21] [22]. A common approach

to address these issues is through the use of tool integration [23], a topic of research since the 1990s.

This study introduces a requirement-centric framework to build tool integration for requirement traceability establishment and consistency checking. Since it's not dependent on specific vendors or tools, it has broad applicability and flexibility, making it suitable for different companies dealing with the integration of various software systems. Given the significant variation in the toolchains used by different companies, this framework offers an efficient solution that does not require substantial changes to existing processes. This approach addresses gaps in current integration solutions in the automotive industry [25] [26] [27] [28], requiring minimal tool adjustments and avoiding the need for specific vendor software purchases.

At the heart of this framework is the principle of gathering data from multiple software platforms needing integration, then centralizing their processing and analysis. The Augmented Lifecycle Space (ALS) concept by Klespitz et al. also focuses on data exchange [31]. However, it requires the tools to support Open Services for Lifecycle Collaboration (OSLC). This requirement restricts the practical use of ALS, as it is challenging for companies to develop OSLC adapters for tools that do not inherently support this standard. The framework introduced in this study overcomes this limitation by prioritizing data over specific standards, allowing for flexible data transfer methods such as APIs, direct exports, and message buses.

Moreover, this framework moves away from the traditional use of Microsoft Excel for managing traceability, which, while popular, comes with significant limitations. Traceability managed through Excel sheets relies on the links or IDs of artifacts [30]. If any discrepancies exist between these links and the actual artifacts, the traceability becomes inaccurate. Besides, Excel cannot automatically detect such inconsistencies. The proposed framework addresses and resolves this issue effectively. The output of this framework includes various datasets that extend its functionality beyond simple traceability. It enables comparisons between different datasets, like test outcomes and requirement specifications, to identify inconsistencies between corresponding artifacts. It also supports analyzing the status of requirements; for instance, if all test cases associated with a requirement are successfully executed, the requirement is considered fully validated and passed. Similarly, if all requirements within the same scope meet this criterion, the entire scope is considered fully validated.

5.4 Validity Threats

This section discusses validity threats based on the concepts of four validity threats provided by Runeson and Höst[45].

5.4.1 Construct Validity

In the evaluation phase of the first iteration, the author helped some survey participants by clarifying their misunderstandings about specific questions. Although this engagement was beneficial, it might have introduced bias by influencing the

participants' responses, which could limit the discovery of other relevant issues. Additionally, using a Likert scale constrained participants from sharing their direct and detailed opinions on the artifact being evaluated. To address these issues, the study included open-ended questions, allowing for more comprehensive feedback.

In the survey, a question was included to assess the adaptability of the designed framework: "How effectively do you think the developed traceability model can be applied to different departments and testing levels?" The Likert scale was utilized for this question instead of requiring participants to explain their ratings. Since no participants provided additional comments in the open-ended sections, interpreting the survey results became challenging. This situation presents a potential risk to the construct validity.

5.4.2 Internal Validity

The workshop aimed to gather a diverse group of stakeholders from different departments within the company and an external supplier closely related to the thesis topic. Unfortunately, scheduling issues resulted in a limited number of participants. This may have affected the evaluation of the framework, as it did not cover all possible scenarios relevant to the broader automotive industry. This limitation was partly addressed by setting up a focused discussion group that included key stakeholders, particularly those who missed the workshop.

The involvement of a system engineer as the industrial supervisor, who has a deep understanding of both the study's theme and the company's operations, might have also affected the internal validity. This arises from the possibility that the system engineer's proficiency with the framework could be mistaken for the framework's effectiveness itself. To counter this, the workshop introduced the framework to stakeholders unfamiliar with any previous implementations.

Ongoing discussions within the company about traceability indicate that data transformation or software updates might necessitate changes to the framework components. Although the use of the CarWeaver API might eventually be phased out if a comprehensive message bus for requirements is established, the framework's various data retrieval methods help maintain internal validity.

Studies that apply design science research frameworks typically consist of three cycles, with the final cycle focusing on evaluating the developed artifacts. However, this study only includes two cycles, with the second being significantly shorter than the first. By the end of the first cycle, most of the implementation was already complete, and the evaluation mainly aimed at further improvements. The unique nature of the artifact developed in this study, a tool integration framework, presents challenges in demonstrating its results and impacts to stakeholders before the framework is fully operational. This limitation may affect the internal validity since the design of the tool integration framework was not thoroughly discussed or evaluated before the coding phase. A more effective approach might have involved using prototypes during the first cycle to present the framework during the evaluation activities. A better balance between the two cycles can be achieved if insights gained from the

evaluation of the first cycle are integrated into the design of the framework and then implemented in the second cycle. This allows the design to evolve based on initial findings, potentially leading to a more robust framework.

5.4.3 External Validity

Multiple teams are adopting similar workflows for manual HIL testing. The designed framework can apply to all of them, provided automated test cases are prepared and suitable API endpoints are available. These endpoints should fetch the required information for the scope under test. Although the implemented code targets specific endpoints for an item called Specific Item Test in CarWeaver, the framework is still functional for other scopes if the necessary data is accessible.

This framework was evaluated only at the case company. Although it is designed to maximize generalizability, the absence of evaluations at other companies could potentially limit its external validity. For companies using the same tools as Volvo Car Corporation, specifically CarWeaver and dSPACE AutomationDesk, the framework can be directly applied. However, its applicability may be restricted to companies that use other tools lacking data export capabilities. To address this issue, combining the framework with manual data preparation processes could be considered.

Due to time constraints, the evaluation of the second and final cycle of this study was less extensive than the first. It included only one evaluation meeting with a single stakeholder. This limitation may pose risks to the external validity of the findings, as the improvements from the second cycle were not showcased to stakeholders from different departments. Ideally, evaluations in this cycle should distribute the implemented framework across various departments. This would allow testers, developers, and integration engineers to practically apply it and offer feedback.

5.4.4 Reliability

Unlike other master thesis projects, this study was conducted by a single author. Instead of using structured data analysis methods like thematic analysis for interviews, the information gathered from observations, documents, literature reviews, meetings, workshops, and focused discussion groups was directly interpreted. This method could impact the reliability of the findings, as interpretations can vary among different individuals.

The survey designed to evaluate the functionality and usability of the artifact was also created solely by the author, which might introduce a risk of having a narrow perspective and reducing creativity and innovation. This issue was addressed by consulting a system engineer prior to distributing the survey, aiming to gather additional insights. To pinpoint unclear questions, the author enlisted two release engineers with a limited understanding of the thesis context to review the survey. Their feedback on any confusion or misinterpretations helped refine the questions, making them clearer and more neutral.

6

Conclusion and future work

This research presents a novel framework that offers an approach to establish traceability and ensure consistency across various software systems. It features widespread adaptability and flexibility and adapts to different company toolchains without the need for new software installations or major changes to current workflows. The framework's primary strategy involves gathering and integrating data from multiple software systems, centralizing its processing and analysis. It relies minimally on specific tools and does not require the purchase of new software from any particular vendor. This method moves away from the traditional reliance on Excel sheets, commonly used in many organizations. Unlike the Augmented Lifecycle Space, which requires compliance with standards such as OSLC, this framework places few restrictions on the tools used. It offers more flexibility in integrating different software systems without specific standard requirements. The study was conducted within the Integration and Release team at Volvo Car Corporation. The framework developed was originally designed to address the traceability challenges around automated HIL testing encountered by the department but it can be applied to a broader context.

For future work, the proposed framework could be extended into a Software-as-a-Service (SaaS) product. SaaS is a model of cloud computing where the provider manages application software and all related resources, typically accessible through a web application [46]. Although the current framework reduces the implementation effort for various companies, it could be further automated to become a complete product. This would enable users to use a website to establish traceability and check consistency.

Similar to the framework, the SaaS product would focus on data operations.

For retrieving test results, the website could include a feature that lets users select and view test results stored in external environments like JFrog Artifactory or databases. Users would be able to provide a schema for test result data to fit the different formats and structures used by various companies. This would require developing a parser to adapt and reformat the data.

For requirements information, the website could let users set up data fetching configurations. This feature would provide flexibility in API settings, such as authentication and endpoint selection. Like the test results, a schema definition and parser would be necessary.

If the SaaS product stores the fetched test results and requirements in a database, visualization could be automated. Currently, the framework involves manual tasks such as downloading CSV files from Jenkins artifacts and creating diagrams in PowerBI Desktop. Automating these processes with an API from the visualization tools would reduce manual tasks and could include version control features.

Additionally, the case company, Volvo Car Corporation, could explore specific advancements. As discussed in the findings, using a message bus to replace direct communications with CarWeaver could improve the process. Future enhancements could involve fetching requirements information directly from the message bus, potentially reducing reliance on the CarWeaver API.

Bibliography

[1] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: Results from a survey," Jul. 2014. DOI: 10.1145/2631890.2631891.

[2] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: Association for Computing Machinery, 2006, pp. 33–42, ISBN: 1595933751. DOI: 10.1145/1134285.1134292. [Online]. Available: https://doi.org/10.1145/1134285.1134292.

[3] H. Cho, "Traceability-driven system development and its application to automotive system development," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1, 2014, pp. 143–146. DOI: 10.1109/APSEC.2014.30.

[4] P. Kafka, "The automotive standard iso 26262, the innovative driver for enhanced safety assessment & technology for motor cars," *Procedia Engineering*, vol. 45, pp. 2–10, 2012.

[5] B. Ramesh, "Factors influencing requirements traceability practice," *Commun. ACM*, vol. 41, Dec. 1998. DOI: 10.1145/290133.290147.

[6] Y. Li and W. Maalej, "Which traceability visualization is suitable in this context? a comparative study," vol. 7195, Mar. 2012, pp. 194–210, ISBN: 978-3-642-28713-8. DOI: 10.1007/978-3-642-28714-5_17.

[7] D.-h. Kum, J. Son, S.-b. Lee, and I. Wilson, "Automated testing for automotive embedded systems," in *2006 SICE-ICASE International Joint Conference*, 2006, pp. 4414–4418. DOI: 10.1109/SICE.2006.314687.

[8] T. Galla, D. Schreiner, W. Forster, C. Kutschera, K. Göschka, and M. Horauer, "Refactoring an automotive embedded software stack using the component-based paradigm.," Jan. 2007, pp. 200–208.

[9] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *J. Syst. Softw.*, vol. 123, pp. 176–189, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:206538464.

[10] M. Fowler, *Continuous integration*, https://martinfowler.com/articles/continuousIntegration.html, Accessed: 2024-3-18, 2006.

[11] M. Leppänen, S. Makinen, M. Pagels, *et al.*, "The highways and country roads to continuous deployment," *Software, IEEE*, vol. 32, pp. 64–72, Mar. 2015. DOI: 10.1109/MS.2015.50.

[12] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and prac-

tices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. DOI: 10.1109/ACCESS.2017.2685629.

[13] M. Fowler, *Continuous delivery*, https://martinfowler.com/bliki/ContinuousDelivery.html, Accessed: 2024-3-18, 2013.

[14] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[15] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, "Software testing techniques: A literature review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.

[16] O. Gotel and A. Finkelstein, "Extended requirements traceability: Results of an industrial case study," Feb. 1997, pp. 169–178, ISBN: 0-8186-7740-6. DOI: 10.1109/ISRE.1997.566866.

[17] P. Carlshamre, K. Sandahl, B. Regnell, and J. Dag, "An industrial survey of requirements interdependencies in software product release plannin.," Feb. 2001, pp. 84–91, ISBN: 0-7695-1125-2. DOI: 10.1109/ISRE.2001.948547.

[18] D. Kim, J. Kim, and M. Lee, "Improvement of hils using advanced exploratory and optimization techniques for system qualification test," *International Journal of Automotive Technology*, vol. 24, pp. 901–911, May 2023. DOI: 10.1007/s12239-023-0074-x.

[19] F. Falcini and G. Lami, "System and software testing in automotive: An empirical study on process improvement areas," Apr. 2021, pp. 253–262. DOI: 10.1109/ICST49551.2021.00035.

[20] V. Q. W. G. 1. /. A. SIG, "Automotive spice process assessment / reference model," VDA QMC, Technical Report, version 3.1, 2017. [Online]. Available: https://vda-qmc.de/wp-content/uploads/2023/02/Automotive_SPICE_PAM_31_EN.pdf.

[21] R. Wohlrab, J.-P. Steghöfer, E. Knauss, S. Maro, and A. Anjorin, "Collaborative traceability management: Challenges and opportunities," in *2016 IEEE 24th international requirements engineering conference (RE)*, IEEE, 2016, pp. 216–225.

[22] D. Amalfitano, V. De Simone, R. R. Maietta, S. Scala, and A. R. Fasolino, "Using tool integration for improving traceability management testing processes: An automotive industrial experience," *Journal of Software: Evolution and Process*, vol. 31, no. 6, e2171, 2019.

[23] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, Mar. 2018. DOI: 10.1016/j.jss.2018.03.060.

[24] I. Thomas and B. Nejmeh, "Definitions of tool integration in software engineering environments," *Software, IEEE*, vol. 9, pp. 29–35, Apr. 1992. DOI: 10.1109/52.120599.

[25] M. Biehl, J. El-khoury, F. Loiret, and M. Törngren, "On the modeling and generation of service-oriented tool chains," *Software and Systems Modeling*, vol. 275, Dec. 2012.

[26] E. Armengaud, M. Biehl, Q. Bourrouilh, *et al.*, "Integrated tool-chain for improving traceability during the development of automotive systems," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:17303902.

[27] C. Wolff, L. Krawczyk, R. Höttger, *et al.*, "Amalthea tailoring tools to projects in automotive software development," Sep. 2015. DOI: 10.1109/IDAACS.2015.7341359.

[28] A. Himmler, J. Allen, and V. Moudgal, "Flexible avionics testing - from virtual ecu testing to hil testing," vol. 7, Sep. 2013. DOI: 10.4271/2013-01-2242.

[29] N. Cara, Navas, J.-P. Steghöfer, and R. B. Svensson, "Automotive spice compliance in an agile software development process a case study on optimization of the work products," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:240285726.

[30] S. Maro, "Improving software traceability tools and processes," 2020.

[31] J. Klespitz, M. Bíró, and L. Kovács, "Augmented lifecycle space for traceability and consistency enhancement," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016, pp. 003 973–003 977. DOI: 10.1109/SMC.2016.7844854.

[32] J. Klespitz, M. Biro, and L. Kovács, "Demonstration of augmented lifecycle space in heterogeneous environment," Feb. 2018, pp. 000 167–000 172. DOI: 10.1109/SAMI.2018.8324007.

[33] M. Ward, G. Grinstein, and D. Keim, *Interactive Data Visualization: Foundations, Techniques, and Applications, Second Edition* (360 Degree Business). CRC Press, 2015, ISBN: 978-1-4822-5738-0. [Online]. Available: https://books.google.se/books?id=XHZ3CAAAQBAJ.

[34] P. Strandberg, W. Afzal, and D. Sundmark, "Software test results exploration and visualization with continuous integration and nightly testing," *International Journal on Software Tools for Technology Transfer*, vol. 24, Apr. 2022. DOI: 10.1007/s10009-022-00647-1.

[35] P. Heim, S. Lohmann, K. Lauenroth, and J. Ziegler, "Graph-based visualization of requirements relationships," Oct. 2008, pp. 51–55. DOI: 10.1109/REV.2008.2.

[36] C. Duan and J. Cleland-Huang, "Visualization and analysis in automated trace retrieval," in *2006 First International Workshop on Requirements Engineering Visualization (REV'06 - RE'06 Workshop)*, 2006, pp. 5–5. DOI: 10.1109/REV.2006.6.

[37] A. Egyed and P. Grünbacher, "Supporting software understanding with automated requirements traceability," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 05, pp. 783–810, 2005.

[38] A. Ahmad, O. Leifler, and K. Sandahl, "Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations," *IET Software*, vol. 16, Jun. 2021. DOI: 10.1049/sfw2.12030.

[39] W. E. Lewis, *Software Testing and Continuous Quality Improvement, Third Edition*, 2nd. USA: Auerbach Publications, 2008, ISBN: 1420080733.

[40] J. A. Ledin, "Hardware-in-the-loop simulation," *Embedded Systems Programming*, vol. 12, pp. 42–62, 1999.

[41] R. Wieringa, "Design science as nested problem solving," in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, ser. DESRIST '09, Philadelphia, Pennsylvania: Association for Computing Machinery, 2009, ISBN: 9781605584089. DOI: 10.1145/1555619.1555630. [Online]. Available: https://doi.org/10.1145/1555619.1555630.

[42] E. Knauss, "Constructive masters thesis work in industry: Guidelines for applying design science research," May 2021, pp. 110–121. DOI: 10.1109/ICSE-SEET52601.2021.00021.

[43] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004, ISSN: 02767783. [Online]. Available: http://www.jstor.org/stable/25148625.

[44] P. Johannesson and E. Perjons, *An Introduction to Design Science*. Jul. 2014, pp. 1–197, ISBN: 978-3-319-10631-1. DOI: 10.1007/978-3-319-10632-8.

[45] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, pp. 131–164, 2009.

[46] T. Golding, *Building Multi-Tenant SaaS Architectures*. O'Reilly Media, 2024.

A

Appendix

A.1 Evalution Survey in the First Iteration

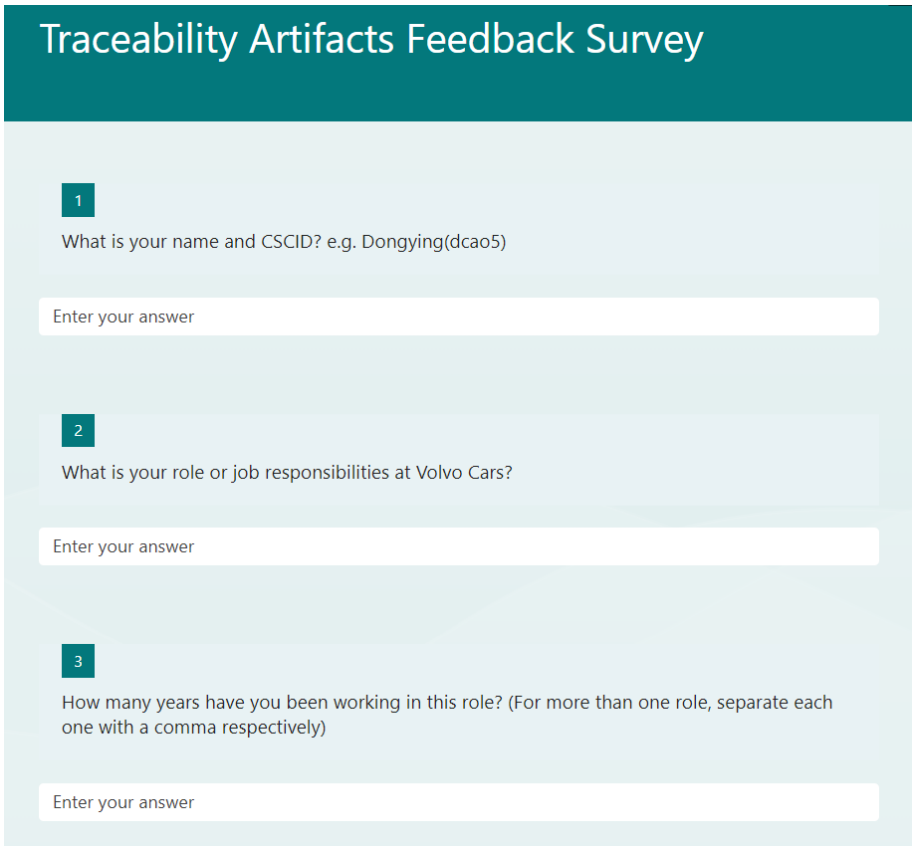


Figure A.1: Survey questions 1-3.

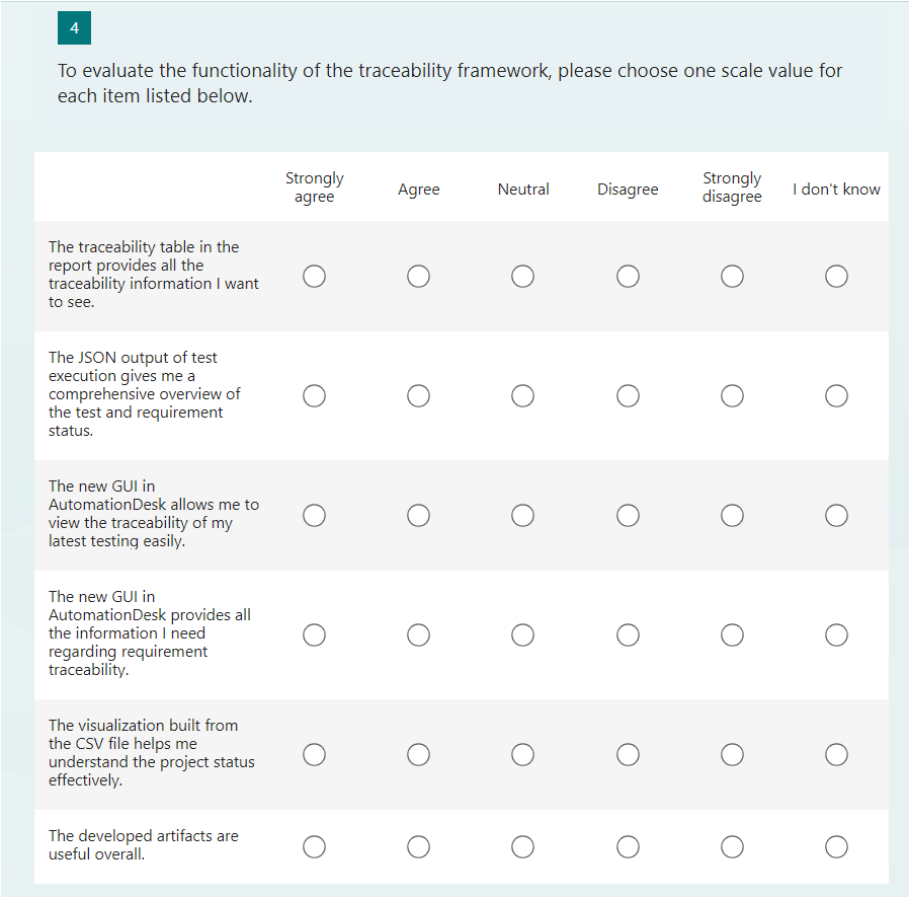


Figure A.2: Survey questions 4.

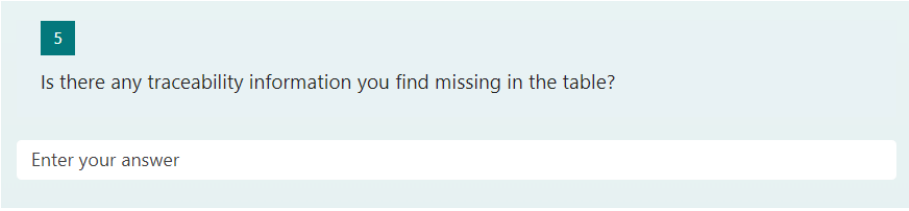


Figure A.3: Survey questions 5.

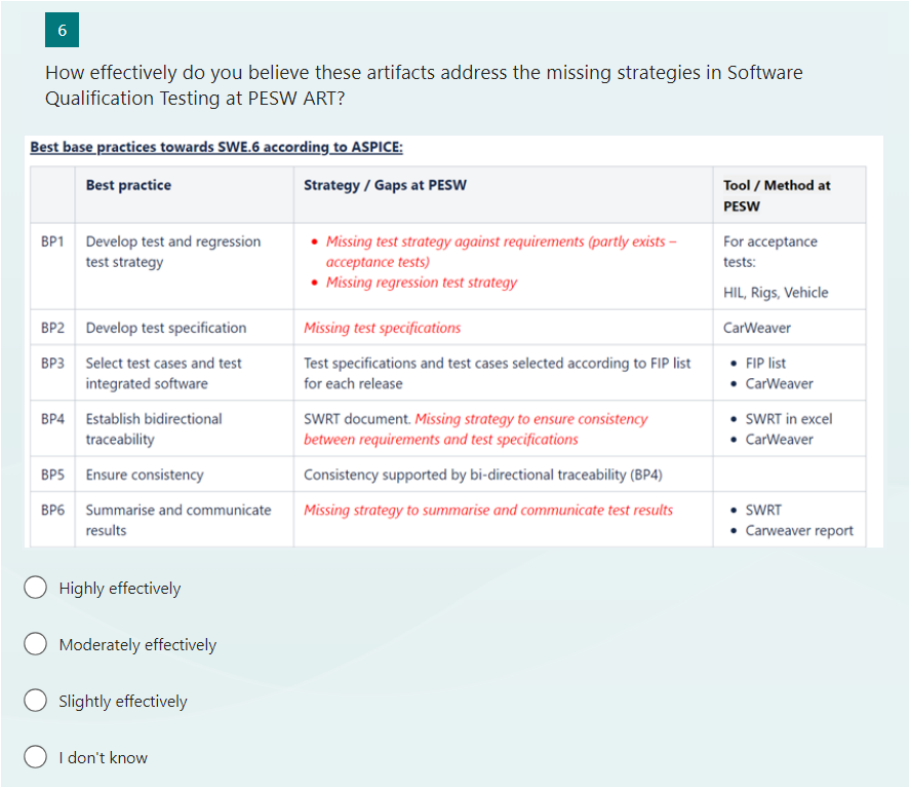


Figure A.4: Survey questions 6.

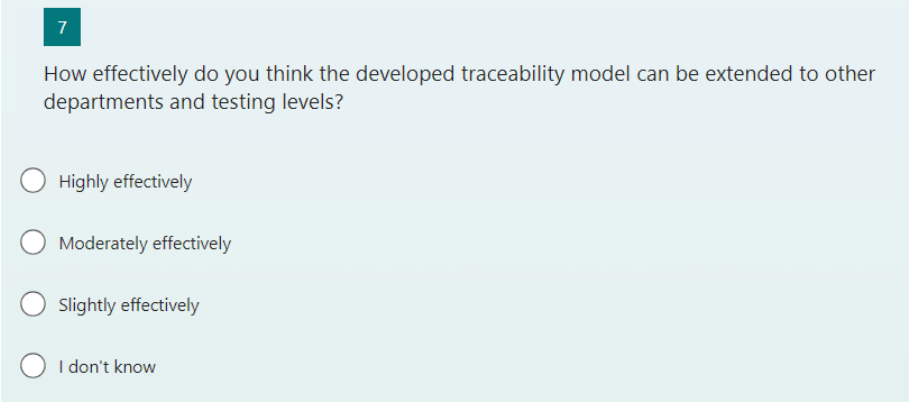


Figure A.5: Survey questions 7.

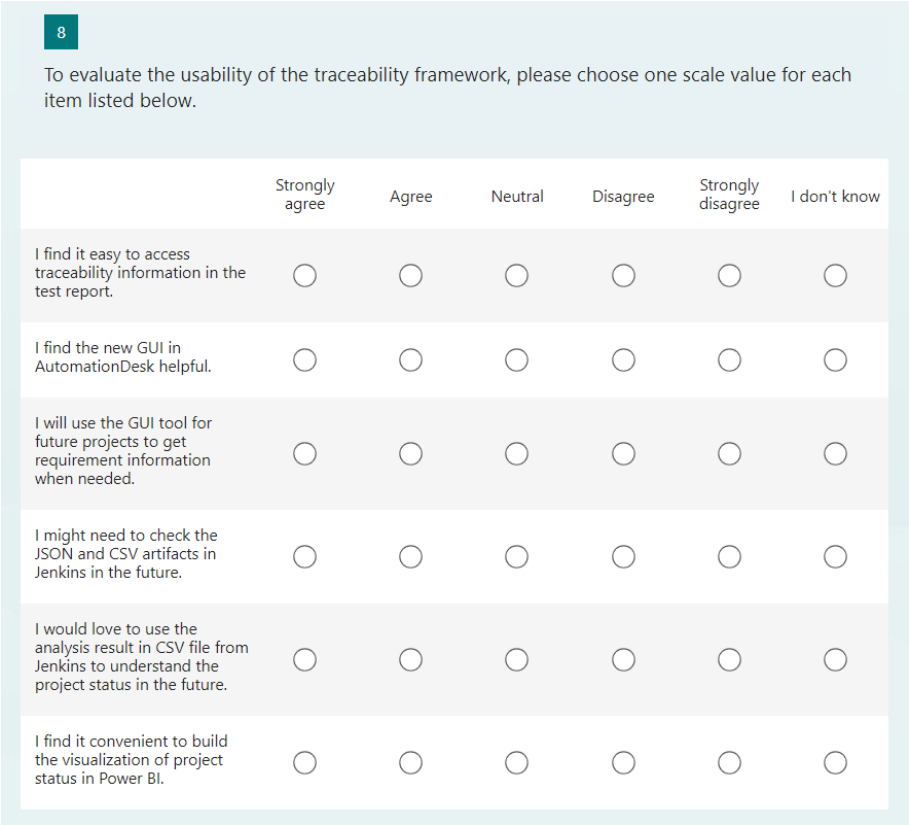


Figure A.6: Survey questions 8.

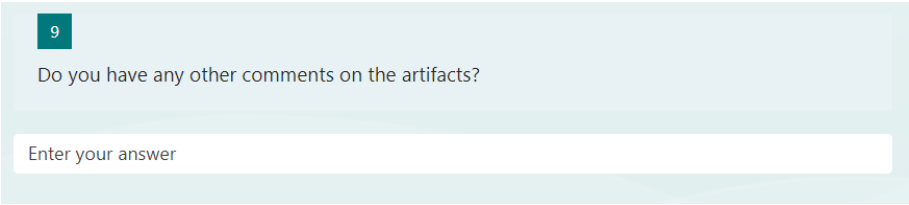


Figure A.7: Survey questions 9.