Mariona Valero Canadell

Tools to assist in the design process of automotive software projects

FINAL DEGREE PROJECT

Directed by Jordi Pascual Fontanilles

Computer Engineering

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2023

Resum

Aquesta tesi aprofundeix en les problematiques trobades durant la fase de disseny d'un projecte de software per a ECUs, específicament aquells que sorgeixen de la traducció manual de les decisions de disseny en diversos arxius. Aquestes tasques manuals no només són repetitives i consumeixen molt de temps, sinó que també són propenses a errors humans, cosa que pot provocar inconsistències en el disseny. Per abordar aquests problemes, aquest projecte es centra en la creació d'una eina d'automatització destinada a agilitzar el procés de disseny i reduir els errors humans.

L'eina desenvolupada avalua la coherència dels fitxers de disseny i l'automatització de la generació de la documentació de disseny. Estableix dos pipelines. Una que alinea els fitxers de disseny per mitigar el risc d'inconsistències que poden generar problemes complexos durant la implementació.

Un èxit notable d'aquest projecte és la prova i validació del procés de verificació estàtica en diversos casos de prova. Els resultats demostren consistentment la confiabilitat i efectivitat de l'script d'automatització per millorar l'eficiència i la precisió del procés de disseny.

A més de la pipeline estàtica, la dinàmica també ha demostrat la seva eficàcia en extreure dades amb precisió dins de les limitacions del projecte de prova. No obstant això, requereix un refinament més gran. La pipeline dinàmica depèn que els desenvolupadors modifiquin el codi font per a la configuració i no avalua la coherència amb el disseny d'alt nivell, cosa que en limita la funcionalitat per a l'empresa.

En resum, aquest projecte aconsegueix amb èxit els seus objectius de simplificar el procés de disseny de projectes de programari. L'eina d'automatització desenvolupada té el potencial de reduir significativament el temps i l'esforç invertit en tasques manuals relacionades amb el disseny, minimitzant així el risc d'inconsistències en el disseny i, en última instància, millorant l'eficiència i la qualitat de la implementació del disseny programari.

Resumen

Esta tesis profundiza en los desafíos encontrados durante la fase de diseño de un proyecto de software, específicamente aquellos que surgen de la traducción manual de las decisiones de diseño en varios archivos. Estas tareas manuales no sólo son repetitivas y consumen mucho tiempo, sino que también son propensas a errores humanos, lo que puede provocar inconsistencias en el diseño. Para abordar estos problemas, este proyecto se centra en la creación de una herramienta de automatización destinada a agilizar el proceso de diseño y reducir los errores humanos.

La herramienta desarrollada evalúa la coherencia de los archivos de diseño y la automatización de la generación de la documentación de diseño. Establece dos pipelines. Una que alinea los archivos de diseño para mitigar el riesgo de inconsistencias que pueden generar problemas complejos durante la implementación.

Un logro notable de este proyecto es la prueba y validación exitosas del proceso de verificación estática en varios casos de prueba. Los resultados demuestran consistentemente la confiabilidad y efectividad del script de automatización para mejorar la eficiencia y precisión del proceso de diseño.

Además de la pipeline estática, la dinámica también ha demostrado su eficacia al extraer datos con precisión dentro de las limitaciones del proyecto de prueba. Sin embargo, requiere un mayor refinamiento. La pipeline dinámica depende de que los desarrolladores modifiquen el código fuente para la configuración y no evalúa la coherencia con el diseño de alto nivel, lo que limita su funcionalidad para la empresa.

En resumen, este proyecto logra con éxito sus objetivos de simplificar el proceso de diseño de proyectos de software. La herramienta de automatización desarrollada tiene el potencial de reducir significativamente el tiempo y el esfuerzo invertido en tareas manuales relacionadas con el diseño, minimizando así el riesgo de inconsistencias en el diseño y, en última instancia, mejorando la eficiencia y la calidad de la implementación del software.

Abstract

This thesis dives into the challenges encountered during the software project design phase, specifically those arising from the manual translation of design decisions into various files. These manual tasks are not only repetitive and time-consuming but also prone to human error, potentially leading to design inconsistencies. To address these issues, this project focuses on the creation of an automation tool aimed at streamlining the design process and reducing human errors.

The developed tool assesses the design files coherence and the automation of the generation of the design documentation. It establishes two pipelines. One that aligns design files to mitigate the risk of inconsistencies that can lead to complex issues during implementation.

A notable achievement of this project is the successful testing and validation of the static check pipeline across various test cases. The results consistently demonstrate the reliability and effectiveness of the automation script in enhancing the design process's efficiency and accuracy.

In addition to the static pipeline, the dynamic pipeline has also proven its effectiveness by accurately extracting within the test project's limitations. However, it requires further refinement. The dynamic pipeline relies on developers to modify the source code for setup and doesn't assess coherence with High-Level design, limiting its functionality for the company.

In summary, this project successfully achieves its objectives of simplifying the software project design process. The developed automation tool has the potential to significantly reduce the time and effort invested in manual design-related tasks, therefore minimizing the risk of design inconsistencies and ultimately enhancing the efficiency and quality of software implementation.

Acknowledgements

Before getting into the project, I would like to express my sincere gratitude to the following individuals and organizations for their contributions and support through the development of this thesis:

First, I extend my appreciation to my professional tutor, Miguel Periago, from *Lear Corporation*. Miguel has given me the tools, knowledge and resources needed to implement and develop this project. I would also like to extend my deepest thanks to my academic tutor, Jordi Pascual, from *Universitat Rovira I Virgili* for his help and guidance during the writing of this document.

Furthermore, I am grateful to the entire team at Lear Corporation for providing me with the opportunity to work as an intern. Their trust and belief in my abilities have been truly motivating, allowing me to gain invaluable practical experience and apply the theoretical knowledge acquired during my studies.

Table of Contents

Figure Index

1 Introduction

This thesis was developed during an internship at Lear Corporation, this is a multinational company that produces seating and integrated systems for vehicles.

There are different departments in the company, this thesis is based on the software department projects, which usually revolve around creating the software for the Electronic Central Units of the cars. The outcome produced in this thesis will be used by the company to speed up the design and some of the testing of software projects of the department.

1.1 Environment and motivation

Any software project has different phases that it goes through, most of them being repeated more than once in cycles to increment the functionalities of the project. Developers are interested in focusing time in designing a solution but most importantly, implementing it.

Lear works on software projects that revolve around implementing software for ECUs, these follow different phases, and this project is focused on the design phase.

When design decisions have been made, translating them onto various files is laborious, manual work that can take much time since there are many that need modification, and this could cause some human error that create incoherence. For example, at Lear during the design phase of a project, they define software components on an Excel file with a specific format, which they also do for sequences of functions in a separate file. Additionally, they also use an application that allows them to represent both concepts together and in a more graphic way. When developers decide that a design change is necessary for the project, there are many adaptations to be made on the files mentioned, which causes an additional time expense. Also, if changes are not documented correctly, this could cause problems in the implementation that could be hard to trace back to the incoherence in the design.

For this reason, this is focused on creating an automation to the repetitive work that could greatly reduce the time invested in design. Besides, some aspects of the design can be automatically tested on the running implementation, such as sequence specifications like order of function calls and the period between those calls, which could also decrease the time of testing.

With the tool this project intends to implement, the manual work of comparing, generating, and updating files will be automated. This will reduce possible human error and it will reduce the time invested on repetitive tasks on the design phase of projects.

To exemplify, a possible situation for a project in the company could be that after a few iterations on a project, the design of the software for an ECU has been changed to make it more efficient. The designer changes one file that contains Software Component data, but one of the developers is basing the implementation on another file that also has software components, but it hasn’t been changed. For this reason, the tool will be useful as it will automatically say when files are not coherent.

1.2 Project Objectives

To ensure that we have a good sense of where the lead of this project is and to be able to assess the results during and after work, it is needed to define the objectives of this project:

1. Produce a pipeline that checks the design and the implementation automatically.
2. Check the coherence of the design with the implementation.
3. Automate the generation of design documentation.
4. Check the coherence of the different files of the design documentation.
5. Produce a pipeline that checks the implementation's execution.

Having these in mind, the main goal of the project is to create a tool that simplifies and shortens the design process of the company's projects.

1.3 Document Structure

This document is split in four sections, on which the project will be thoroughly explained. Beginning with the introduction which has established the motivation and environment of this project and its objectives. Then, we proceed to the background section, which will cover the concepts and technologies needed to understand and develop this project. After, the following section will cover the implementation, this includes the requirements, design, development, and testing. Finally, the conclusions will synthesize the information of this document.

2 Background

The software projects that this thesis focuses on, have many similarities, which allows developing a common tool that works for multiple projects. This section explains the theoretical concepts of the common properties among the different projects that will be tested.

2.1 ECU Architecture

An Electronic Control Unit (ECU) is a piece of hardware in cars that controls various systems and functionalities throughout the vehicle. Since nowadays, the automotive world has evolved in such a manner that demands more sophisticated functionalities from vehicles, the load of work placed in ECUs has increased substantially. This created a necessity of more powerful ECUs that communicate efficiently between themselves, since the number of ECUs inside vehicles has also increased.

The new functionalities of the ECUs have made the automotive sector move from classical architectures where all the functionalities of the vehicle were connected to a single ECU that controlled everything, to a distributed architecture, where each zone, group or domain of functionalities is controlled by a different ECU.

To create a good communication between ECUs, there are various buses and protocols. The most common are CAN, LIN, FlexRay and Ethernet. Each of these protocols works differently and is used for different purposes. CAN is the general protocol for most of the ECU's communication, LIN is for one way communication used, for example, to turn on the wipers. FlexRay is the most secure communication, since it has two buses that can work independently in case one of them fails. Therefore, these are used for the most important communications involving vehicle safety, like ABS or vehicle stability modules.

There are other important factors in ECU architecture. For one, there is the abstraction of hardware, which allows interaction with the components without needing knowledge of their specific functions. Additionally, software layers play a crucial role in simplifying the development and testing of the software for these components.

Finally, since the software is more advanced and might need updates over time, now there is the possibility of Over-the-Air updates that are useful once the vehicle is sold, being able to make the software more secure and updated without the need to physically be in the shop.

2.2 DevOps

The academic definition of DevOps is "A set of practices intended to reduce the time between committing a change to a system and the change being placed into a normal production, while ensuring quality". In other words, it is a set of practices whose primary objective is to create secure code faster and more efficiently. The word DevOps is a portmanteau of the words Development and Operations [3], it was created specifically for software development.

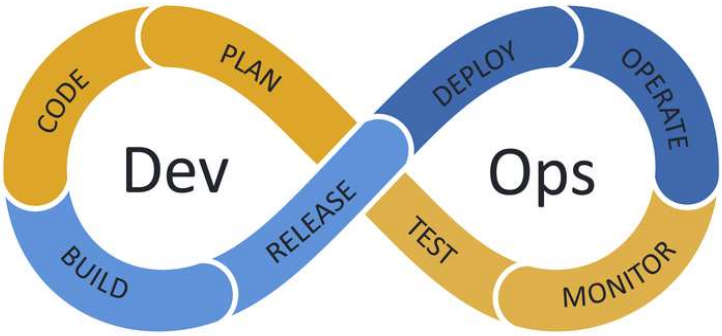

Figure 1: DevOps cycle

The most fundamental objective is to make very small but frequent increments [4]. To achieve this, the practices used are:

- **Continuous integration**: This practice consists of repeatedly merging new code onto the project to be tested and deployed.
- **Continuous delivery**: This practice is deeply correlated with the previous one, it entails that with every new integration there will be an automatic build and set of tests executed to ensure that the new addition works correctly before the release.
- **Microservices**: This is a design approach that combines microservices with different functionalities to form an application. This ensures scalability and makes it easier to detect errors. The services communicate through an interface, usually an HTTP-based API [4].
- **Infrastructure as Code**: When dealing with infrastructures, this practice simplifies the work by treating them as if they were code, using software development techniques. This allows the interface to be configured with code and to automate the build, which makes the integration of changes to the interface standardizable. This practice also allows enforcing standards dynamically, making it easy to flag uncompliant resources.
- **Monitoring and Logging**: This practice is crucial to ensure a well-functioning project, since with each increment there can be new improvements, but also errors or bugs. Monitoring and logging consist of capturing data and categorizing it properly, according to the increment it belongs to and if it is an error or performance data. This allows for a faster identification of errors, their source and of highly performative improvements to maintain.
  This practice is more important on cases on which the service has to be available 24/7 and with update frequency increases.

- **Communication and Collaboration**: Increasing communication and collaboration between the teams that work on a project is a key cultural aspect of DevOps [4]. Splitting responsibilities and making collaborations between teams allows the whole organization to work towards the common goals. DevOps also provides the tools to facilitate this practice, through chat, tracking systems and the generation of wikis [4].

DevOps has many advantages, since it is based on very small but frequent additions to the projects. It allows working faster and more efficiently, and it is also more secure. If any additions to the project don't work, they can be backtracked to a smaller piece of code. Moreover, the problems can be avoided by going back to an older version of the project, making the work more reliable by assuring that there will always be a working version. Finally, DevOps also makes collaborating simpler, reducing inefficiencies, and saving time [3].

2.3 High-Level Design definition

A software project goes through different phases throughout its development, allowing an efficient and fast delivery of a solution. There are many methodologies that provide views on how projects should be developed, but, the phases on each of them are common.

The common phases for software development are to define requirements, the design, proceed to implementation and then verification of the implementation. Finally, if needed, there will be maintenance [7].

All these phases are important and crucial to create a good software product, but in this thesis the focus is on the design, more specifically, the high-level design. On one hand, the design is a process on which the software solution is specified based on the requirements, creating a base for the implementation. On the other hand, the high-level design is the first step of the design process, where the focus is outlining the overall structure and the main components of the project.

The most important aspects of the high-level design for this process are the following:

- **Software Components (SwC):** A software package or module that encapsulates a set of functions [11]. This way, each software component has its own job, which can be implemented and tested separately. Important data specified during the design process of these components are their name and properties.
- **Sequences:** In this case, sequences refer to the structure of the function calls that the software is going to implement. These sequences need to specify which functions will be called, in which order, if they are called by other functions, if they are periodic calls, and if so, their period.

Another important aspect of the design is the way the data produced is represented. This can take many forms and each of them has their purpose. The relevant data representations for this thesis are:

- **UML diagrams:** UML is very useful for detail, visual representations of the components and sequences. Software components are represented in class diagrams that define their hierarchy and function. Sequences are represented through sequence diagrams that visually represent the flow of the functions.
- **Tables and databases:** Tables and databases are used to represent detailed and accessible data for both software components and sequences. These aren't as visual as the UML diagrams, but they are easier to modify through the project's phases and allow easier access to specific properties or specifications.

2.4 AUTOSAR

The company's software projects target the automotive industry, and therefore, they use the standard architecture specifically designed for this kind of projects, AUTOSAR. AUTOSAR, an acronym for Automotive Open Systems Architecture, was created by the AUTOSAR Consortium, a group of companies that were interested in creating a standard to develop Electronic Control Unit (ECU) software, the Consortium operates under the motto "Cooperate on standards, compete on implementation".

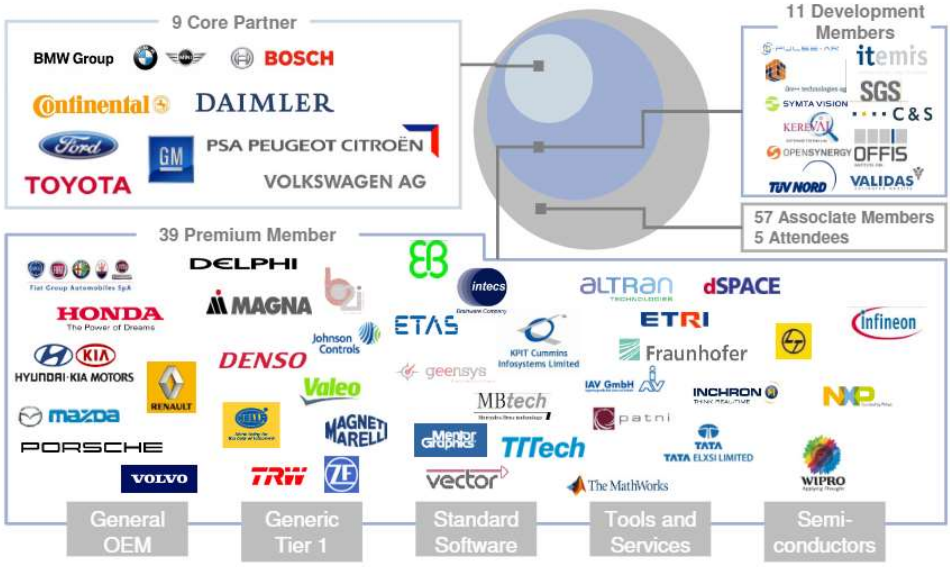

Figure 2: AUTOSAR Consortium

It can be defined as a standardization of automotive software created to produce more reusable and scalable ECUs, car components and software. The main goal of this standard is to make collaboration among companies of the automotive sector easier, enabling shared work on car models efficient. By using the AUTOSAR model, companies can achieve greater scalability in their software development process.

The most perceivable advantages the company can notice by using AUTOSAR in their projects are:

- **Modularity**: AUTOSAR promotes the development of modular software components, allowing for easier integration and reuse across different projects.
- **Portability**: The software can be easily transferred between different model ECUs and cars, greatly reducing costs.
- **Interoperability**: AUTOSAR makes it possible to have seamless integration of components from different suppliers on only one model.

AUTOSAR has been adopted by most companies of the sector and continues to adapt as the industry changes. It is anticipated that AUTOSAR extends to the new technologies emerging, like autonomous driving and electrification, making it possible to address the new challenges that these technologies will create.

2.4.1 AUTOSAR Architecture

AUTOSAR has specific standards for the different sector's necessities. There's the Classic Platform which is the AUTOSAR's solution for embedded systems with hard real-time and safety constraints; Adaptive Platform was created for high performance computing ECUs for uses such as autonomous driving; Foundation holds properties from both Classic and Adaptive platforms released in a separate standard; Acceptance Tests for Classic Platform are systems tests at both bus and application level to validate software components and communication bus performance and lastly, Application Interfaces are standardized set of application interfaces form different vehicle domains [1].

This thesis focuses on the Classic Platform, which distinguishes on the highest level of abstraction between three software layers of a microcontroller: application layer, Runtime Environment (RTE) and basic software (BSW) [2]. This layered architecture, depicted in Figure 2, allows for software and hardware independence, enhancing interoperability and modular development which, as mentioned previously in this document, are some of the main goals of AUTOSAR.

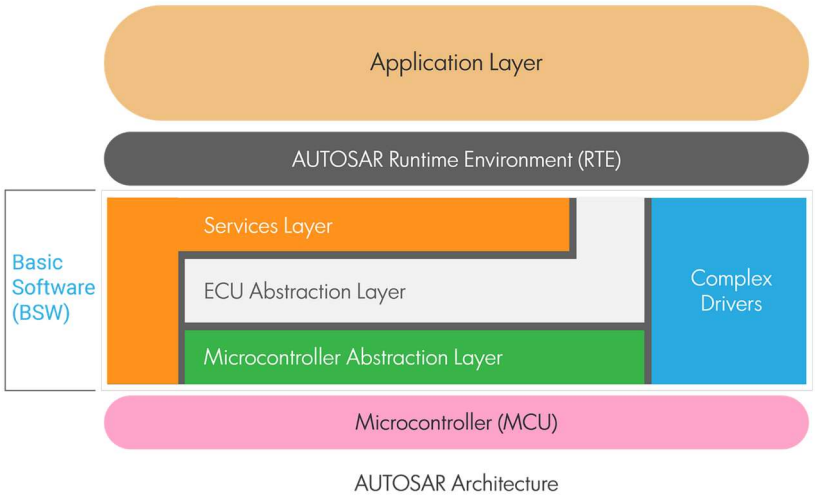

Figure 3: AUTOSAR Architecture

The application layer is based on separate components and is mostly hardware independent, this makes it easy to create software components without a specific hardware in mind. On the other hand, the basic software is hardware-oriented, and it can also be further divided into other sections: services, ECU abstraction and microcontroller abstraction.

Lastly, the Runtime Environment is the interface between the application layer and the Basic Software. On the highest layer of the RTE, the one closest to the application layer, the development of applications is independent to the hardware, on the other hand, the lower layer of the RTE is closer to the BSW which means that the applications developed will be more hardware specific.

2.4.2 AUTOSAR Methodology

Because of the need to improve the collaborations between companies, apart from the architecture mentioned in the previous subsection, AUTOSAR also provides a methodology to develop the automotive software. Furthermore, this methodology also helps cut down on development costs and improves the interoperability of the toolchains.

This methodology is based on standardizing the representation of information. It creates a basis to represent all the data consumed or produced, such as the software components and their properties, bus systems, functions, and their sequences, etc. The descriptions are based on .arxml files, and they are held in templates which define the formal exchange format (AUTOSAR Schema). With these descriptions, there are various tools that support the configuration and generation of RTE and basic software automatically.

2.5 Azure DevOps

Another tool used in the company's projects is Azure DevOps, a software development environment based on DevOps. It offers many functionalities and services useful for software projects all the way from planning, through development, to testing [6]. The details of the most relevant functionalities for this thesis are:

- **Azure Repos:** Since Azure DevOps is a set of services and tools for software development, it needs functionality to control the implementation versions and access the source files. This software allows for two types of version control: Git and Team Foundation Version Control (TFVC) [6]. The files can be organized into folders, branches, and repositories.
- **Azure Git Repos:** In the company's projects, specifically, the type of version control used is Git based. This allows the developer to have a copy of the source repository on their machine and a version history accessible to navigate through. Each change to the code can be committed on the developer's machine and then uploaded through a push command for others to use or review. This type of repository allows for all the Git command lines to manage the versions of the project. An important command for this thesis is "*checkout*" which allows accessing different repositories and branches.
- **Azure Pipelines:** Azure allows for process automation through pipelines. Many processes can be automated such as build, test, and release [6]. For example, you can create processes that automatically run every time a commit or push is made to the project you can also recreate production environments and test functionalities or automate tests that execute after every build [6].
- **Azure Artifacts:** This service allows the developers to share and access packages in one place. If there is an output produced by a pipeline, it can also be shared through this functionality for easy access. Azure Artifacts supports multiple package types [6].

2.6 Enterprise Architect, SDY and VP

To represent the High-Level Design, Lear uses various tools. In this section, the types of files that the tool will analyze will be defined and described. Starting with the tool used for UML diagram representation, Enterprise Architect.

Enterprise Architect is an application that graphically represents a design through UML diagrams. It also organizes software components into directories and defines each of their properties. Additionally, it offers functionalities for projects using the AUTOSAR modules, and it allows the representation of sequence diagrams.

Additionally, this application allows automation through libraries that can be imported into code using programming languages such as Java and Python. This feature will be useful for the purpose of this thesis. Through this library, the object model of the design can be imported as a data structure and operated with different functions. These operations can be seen later on.

Other data representations are through tables and databases. In this case, the company uses Excel spreadsheets with predefined formats. In this thesis, two of these formats will appear. Firstly, the file representation of the Software Components, which is called the Verification Plan or VP for short. This file has a structure on which the components' names and properties are defined on the file's columns, allowing easy access to each of them. They can be organized through the values of the columns to find common properties for the components, such as if the component must have Unit testing implemented or who is the developer in charge of it.

For the sequences, the spreadsheet format is called SDY. It holds information about the period of the functions, their order and the ECU that calls them. It also has a format like the VP that is standardized for the whole company. This will allow easy automation of the tests, but different (older or newer) versions of this format can cause problems if used, since columns and fields may differ. Moreover, scripts are tailored for one format and may not accommodate other versions.

3 Implementation

After the previous sections of this report, the basic concepts needed to understand the projects that are going to be analyzed by the tool have been made clear. These projects have their specifications defined in Excel files with various formats and on Enterprise Architect projects. All of these get posted to the git functionality of Azure DevOps. The rest of this document will focus on the implementation of the tool.

The main goal of this work is to analyze the data on the company's projects and produce a set of Azure DevOps pipelines. They will run scripts to check the data of the projects, extracting conclusions in a readable form so that developers can modify their projects to be coherent and up to their needs.

In this section, the project needs will be defined, also the specific requirements and how these have been implemented in the program. Then, the design of the project will be specified and followed by how this has been implemented. Finally, there will be a subsection with the validation and tests that the final project has gone through to make sure the functionalities work as expected.

3.1 Requirements

To ensure that the outcome of this work is useful for the company and serves the purpose it was designed for, a set of requirements that specify the implementation's expected behavior is defined. In the following subsections, each requirement will be explained in detail.

3.1.1 Generalized usage

The checks implemented must be interchangeable between projects, minimizing the need to adapt them. In other words, many different projects should be able to run the scripts with little to no adaptation.

This might not be entirely possible since every project differs from the rest in some way, but the implementation should take this into account and generalize the functions so that the data from the project can be reached regardless of where it is stored.

To give a practical example, two Excel files holding information about the software components of two different projects, both have 11 fields to test with the Enterprise Architect file corresponding to their project, but on each of those files the fields are held in different positions or have slightly different names. In the function of the script, this has to be accounted for, and the information has to be extracted from the proper column in each project, despite the possible difference in the Excel file formats.

3.1.2 Simple use

The pipelines generated must be easily integrated into the projects with little modification needed, and the checks should run automatically with each new increment on the project. In other words, the process has to be as automatic as possible, which will increase time and cost savings.

3.1.3 Multiple use cases

When working on a project, there may be different needs. For example, at the beginning phases of the project, a few files might not have been created yet. In another phase, many changes might have occurred in software components or state diagrams, and these might not be correctly represented in all files. For this reason, the pipelines will have to work in different ways for different scenarios.

If the software components were only defined in the Enterprise Architect project at the beginning of the project, the Excel file can be generated from that data. If these two files are present in later stages, they can be compared to check for coherence.

3.1.4 Adaptable functionalities

Projects change and evolve; in the future, the necessities and objectives of the company's projects might differ from the present ones. To make this project useful for an extended period of time, the functionalities have to be adaptable and reusable in the future.

For this reason, the implementation has to be well structured into functions that can be used later in different orders or adaptations. These also must be well commented so that they're easy to understand by future users.

3.1.5 Accessibility

The main goal of this project is to simplify the design process, which is why the output generated by the checks must be easily accessible and organized. For the whole process of automating to be worth it, the conclusions must be easy to extract and understand.

To accomplish this, the output files must be clear, organized neatly, and well labeled so that the users can find what they are searching for as quickly as possible.

3.2 Design

In this section, the project will take shape based on the specified requirements. There will also be an introduction to the tools used for the development of the functionalities. First, the specific tasks needed in each pipeline are presented. Next, how the pipeline will be integrated into Azure DevOps is explained.

3.2.1 Static check

The static check involves inspecting the concepts of a project that are not dynamic, that is, the ones that are fixed. Here, we discuss the tasks needed to complete for these static checks and the structure of the pipeline that will run them.

The pipeline will be implemented through Azure DevOps. As mentioned in Section 2, Azure DevOps has many functionalities, like Azure Git Repos, which is where the company uploads the projects on which they work. This is crucial for the pipeline use since it will make the files reachable through the Azure pipeline functionality that will automate the checks.

Figure 4 depicts the different steps that will compose the pipeline. Access to both the project to be tested and the scripts on which the functionalities of the testing have been implemented is required. Hence, there must be two checkouts to both repositories, marked with the numbers one and two in Figure 6. A checkout is a command that allows to access the specified repository and branch. Next, the pipeline will proceed to step three, run a batch file that will have to prepare the environment by installing the necessary packages and libraries. Following, in step four, the tests will run through the scripts. Finally, the generated outputs will have to be stored in a server accessible by the developers of the project.

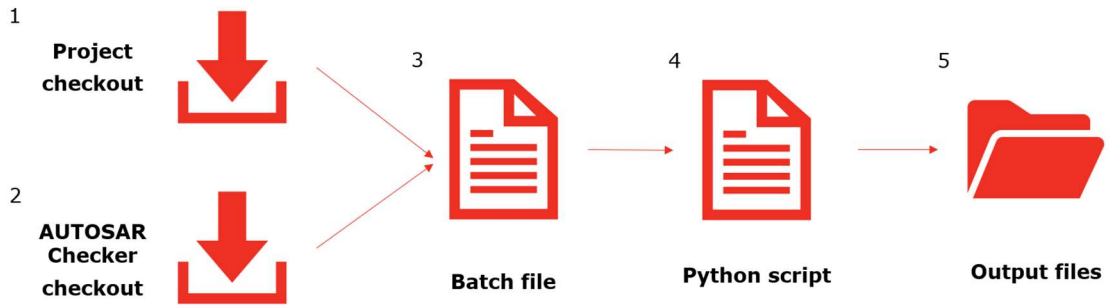

Figure 4: Static check pipeline structure

With the pipeline structure defined, we can proceed to specify the structure of the steps composing it. The batch file needs to follow a few ordered steps:

1. If the needed python version is not installed, install it. The needed version will be defined when the script is implemented, based on the libraries needed to function.
2. Set up python environment, by installing the needed libraries which will be specified in the standard for Python, which is using a requirements.txt file.
3. Execute script containing the checks, introducing all needed files as parameters.

Now that the batch file has been defined, the tests can be designed. These need to check very specific parts of the project, thus, its steps must be carefully defined. First, the needed data must be extracted from any Excel or Enterprise Architect file into a .txt, to facilitate the access to the information in a quick and readable way. In addition, if the same concepts are defined in multiple files, they must be compared to find differences. These have to be extracted into the output. On top of that, if any of the files that are recognizable by the tool aren't introduced as parameters, they will be generated if they represent Software Components. Since there will always be at least one file that has data about the software components, because that is a requirement to run the script.

To put it more simply, the steps for the script are:

- Data extraction
- Coherence check
- File generation

3.2.2 Dynamic check

In this subsection, the meaning of a dynamic check will be explained, and the tasks needed to run for its pipeline will be stipulated.

The dynamic checks of the project consist of testing some values that will be active during runtime. This involves any running functions and the data that surrounds their process. In other words, the concepts to be checked on the dynamic pipeline are properties like the period between calls on a function, the order functions are executed or the times they are executed throughout the whole program.

To summarize, it will allow simulating the ECUs and run the projects that need to be tested without the need of a physical ECU on which the software can be run. The pipeline will upload the results on a txt file in Azure, so developers can access them.

3.2.3 Tools

In this section, the tools chosen to develop this project are explained and justified, keeping in mind that tools like Azure DevOps, Enterprise Architect and AUTOSAR are company requirements. They are deemed mandatory for this project since the company uses them, and they are a crucial part of the projects that this tests intent to work with.

3.2.3.1 Python

The first choice to be made is, which programming language will be the most optimal to develop the functionalities needed. To make this decision, there must be a clear understanding of the project's needs. Starting with the simplest which is to generate .txt files, moving on to easy environment setup in another machine, and complex Excel formats manipulation. Most of these needs are met by many programming languages, so these aren't going to be as important in the decision-making process. In contrast, a very important necessity is to access the object that gets created when using Enterprise Architect and be able to modify it.

The official Enterprise Architect website only provides information on how to connect to the object model through two programming languages: C# and Java. Also, Python allows the use of a library that imports objects from applications, so that is the third possible option.

With a short list of possibilities, the conclusion was that Python would be a better option because of the accessibility to the Enterprise Architect application. Both Java and C# had to function with the application already running, which is not necessary for Python. The

fact that the tool to be implemented is based on automation on a remote server, means that Enterprise Architect will not be running when the pipeline is executed, for this reason, Python is the better choice.

Python is the final choice for this project, and it has access to Enterprise Architect through a library called *comtypes*. This library is a Python COM package which allows access to COM interfaces [12]. A COM interface stands for Component Object Model interface and is used to define the way that a component or object can be accessed and manipulated by another software or other clients.

3.3 Implemented functionalities

In this section, the final implantation of the tools produced in this thesis project will be explained. Starting with the implementation of the steps in the static check pipeline, which includes the batch file, the scripts containing all the tests and the output produced. Following, the steps of the dynamic check pipeline defined in the previous section will also be explained.

3.3.1 Static check pipeline

As mentioned in the design section, the static check pipeline starts with two checkouts. The checkouts are configured through the Azure Pipeline graphic interface. Once completed, the pipeline will have access to the tools' scripts and to the files of the project that are going to be tested. Following that step, the batch file is added to the pipeline.

The batch file includes the environment set up mentioned previously. It installs the Python version needed to function with the required libraries, which are also installed through the batch file. After this, the scripts can be run. The script has a set of parameters that can be introduced to include the files that need testing in the tool.

The accepted arguments on the Python script are:

- The Enterprise Architect project file path, with the required extension .eap. This parameter is introduced with the -p or –project option.
- A path to the Verification Plan, marked with a -v or –vp.
- Path to directory containing .yaml files specifying Software components, marked with -y or –yaml.
- A path to the SDY Excel, marked with -d or –seq.
- A path to the directory containing the source code of the project being tested, marked with -i or –implementation.
- A path on which to save the output produced, marked with -s or –save.
- A path to an Excel file containing the script arguments for different projects in the columns, as an alternative argument input method. This option was created for testing the script functionalities with multiple projects at a time. Marked with and -e or –excel.

These are the possible arguments, but there is no need to input all these files. The code will adapt and depending on the arguments introduced, the script will run different sets of tasks. Nonetheless, there are some conditions that need to be fulfilled.

For the script to run correctly there are two options The first option is that there must be at least one file containing Software Component information, like the Enterprise Architect project, the Verification Plan or the yaml directory. Otherwise, there must be an Excel file containing arguments of at least one project, as mentioned above. If this Excel file, meant for testing, is not the only argument introduced, the script won't run since it will be considered a wrong input. The other arguments, like the SDY, the implementation or the

saving path are optional. If the saving path is not introduced, the output files will automatically get generated on the same directory of the project being tested.

The different parameter combinations make the script act in different ways. of the explanation is provided using Figure 8.

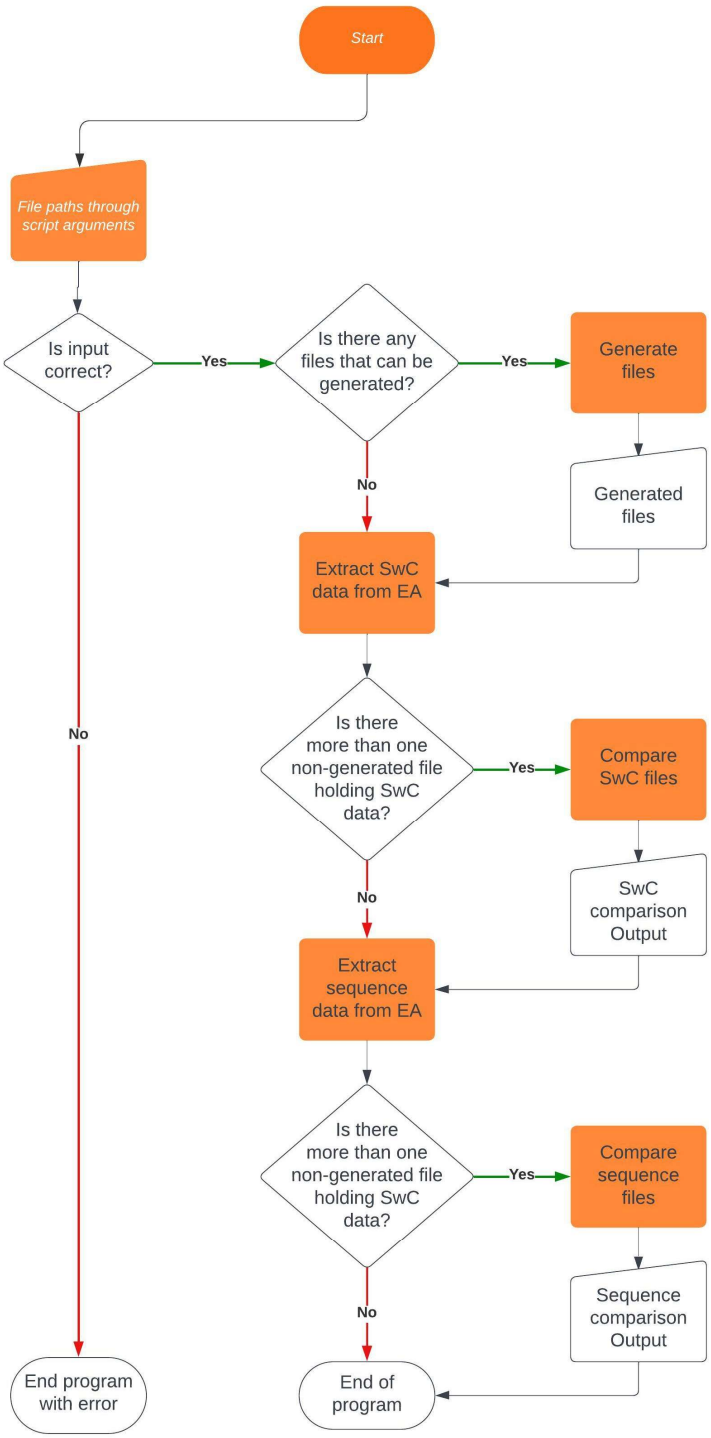

Figure 5: Static checks workflow

Figure 8 illustrates the workflow of the script, starting from the top of the figure, the script starts with the inputted paths to the necessary files. As explained above, not all the combinations of parameters are correct, so the script will make sure it can proceed without error before moving on to the tests. If the input is correct, the script will be able to move on to the checks.

The first task to do will be to check the inputted files, if the Enterprise Project, the Verification Plan or the Yaml directory are not in the input, they get generated through the others, since, as explained above in the arguments of the script, at least one of these files will always be present.

If there are no missing files or if the file generation has finished, the script will proceed to check the Software Components. First, it will always extract the data into .txt files. Then, without taking into account the ones just generated, if there's more than one source of data about Software Components, which could be the implementation, the Enterprise Architect project, the Verification Plan or the Yaml directory, they will get compared. The comparison will generate .txt files with simple sentences explaining the incoherence of the data. An example of one output file after comparing the Implementation with the Enterprise Architect would be "The software component X is not present on the Enterprise Architect" or "The software component X is not represented.".

After this, the script will move on to check the sequences. If the Enterprise Architect was not generated or the SDY file is present, there will be data extraction of the sequences into txt files. If both the conditions mentioned are true, afterward, the files will get compared. In the case that the Enterprise Architect project that was generated, it would only have information about SwC which would make the comparison useless.

Once all these checks have been run, the script will have ended its course and the pipeline will proceed to upload the output files to the server accessible to the developers.

At the beginning of the implementation process of this tool, all the checks and functionalities were on a single file. After a certain point, the big quantity of functions in one file would have made the code unreadable and disorganized, so a class structure was created to divide the functions into clear groups and create a more efficient process.

The class structure can be seen in Figure 9, and it involves an abstract class called checker. It creates a base for the other checkers, which will inherit some of the functions, allowing for code reusability. In the figure, the representation of the most important attributes and functions is present, and as it can be seen, there are four classes inheriting from checker.

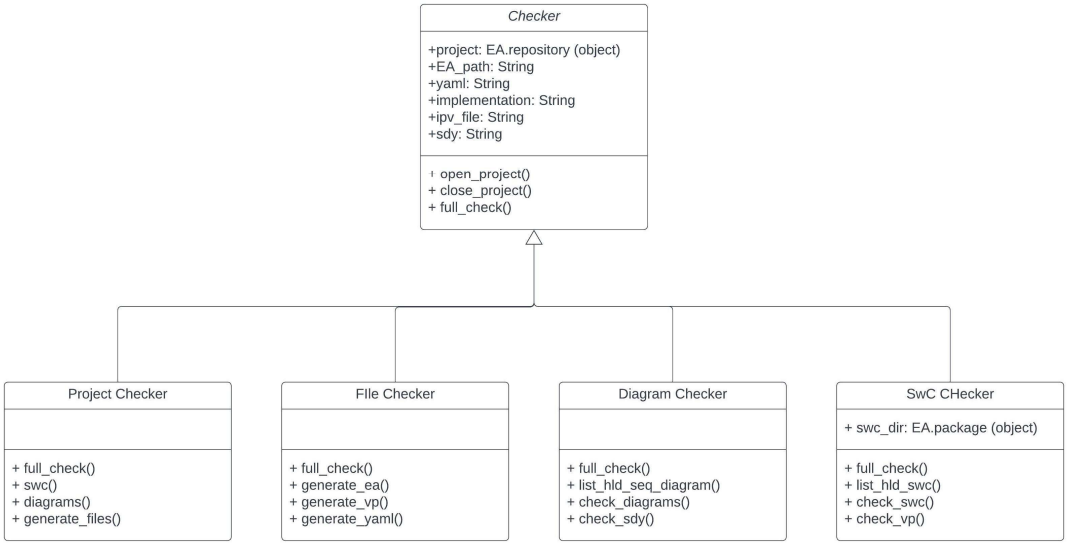

Figure 6: Static checks' script implementation classes

The Project checker will be the class that oversees defining which checks need to be made, depending on the input. The file checker will be the class that generates all the missing files. The diagram checker will extract sequence diagrams information and compare files if needed. Finally, the software component checker, will also extract SwC data and compare files if necessary.

All the classes have the attributes of the abstract checker class. The project attribute is an object of type EA.Repository, obtained with the comtypes library using the *comtypes.CreateObject('EA.Repository'*) function. Once the object has been created, the Enterprise Architect information can be imported through the path in the variable *EA_path*, or it can also be generated. The other variables hold the paths to the other files that the script might need. If the paths are not obtained through the parameters of the script, they will be generated when generating the files.

The *full_check()* function is not implemented in the checker class. Each of the classes has its own implementation of it, which will call the functions of the class in a certain order and ensuring that the checks are needed for each case.

3.3.2 Dynamic check pipeline

For the dynamic check pipeline, there were a few implementation problems that came up and created the need to change the design of this part of the tool. In the original design, the pipeline had tests that checked the coherence between the function sequences while the source code of the tested project was executed, and the sequences that were on the High-Level design.

To clarify, the initial design for this pipeline was to implement a set of tests that would compare the execution of a project to its High-Level design. The results would be extracted on a file and uploaded to a server.

As a result of the difficulties presented in this implementation, the final solution has a different workflow than originally intended. The first step will be to modify the functions of the project by adding a few lines of code that count the time a function has been running and the times it has been executed. Also using an array, the order of the functions is stored to be extracted later. In other words, there is no set of tests implemented in a separate tool, there is only data extraction gotten from the changes made to the source code.

The output of the execution is an Excel file with a few columns that hold different information. The first column is named “Order”, and it holds a number that specifies in which position of the sequence the function of that row has been called. Following, there are the names of the functions, the period on which they get called and how many times they have been called.

3.4 Tests and Validation

In this section, the tests made to ensure that the tool works as expected are explained, as well as the results obtained.

Starting with the static check pipeline, it has many aspects that need to be addressed. Firstly, the script on which the functionalities are implemented must be tested on the way it handles the arguments introduced. If the arguments are incorrect, it must stop the execution of the script. Otherwise, it must act accordingly to the arguments that have been introduced, meaning that it must adapt the tasks that it runs.

To test this, a case table was created to ensure all the possible argument combinations were accounted for. Once that was done, all the cases were executed, and the output was judged manually based on the expected outcome. In Table 1, some of the combinations of arguments and the expected outcome are represented.

Testing Excel file	Enterprise Architect introduced	Yaml Directory introduced	Verification Plan introduced	Expected Outcome
Yes	No	No	No	Works correctly, generates output according to parameters in Excel
No	No	No	No	Doesn't work, expects some parameter
Yes	No	Yes	No	Doesn't work, explains correct argument combinations
No	No	No	Yes	Works, generates Enterprise Architect and Yaml Directory from Verification Plan. Extracts software component data in txt file.

After testing if the script ran the correct tasks, the functionalities must be checked to ensure they work correctly. To do this, coherent and incoherent files were introduced in the tool. By manually defining the expected output of the files and checking the output of the tool, the functioning was confirmed to work correctly. This was also done in all the combinations of comparison. For example, an Enterprise Architect and Verification Plan that were coherent to each other but incoherent to the Yaml directory showed only incoherence in the Yaml directory output file.

The script was tested with many projects and combinations of files, and overall all the tests passed correctly. Meaning that we can assume the correct functioning of the script.

To test the function of the pipeline in Azure, I was only allowed access to one repository, which makes it harder to test. Since there is only one case to run, it was impossible to ensure that the pipeline would work on different cases. Nonetheless, the pipeline was correctly set up in the repository, it worked every time someone made a change in the project through git commands, and it automatically deployed the tests that produced an output reachable for the developers through a server.

For the dynamic pipeline, I was also only allowed one project. To test the functioning of the extraction of data, the tool was executed, and the output produced was inspected. The output was correct, as it had the expected values. For the dynamic check, the pipeline was also tested the same way as the static one.

With only one project to check the pipeline, the setup was made, and the pipeline did execute correctly with every commit and push. The output was also reachable for developers through a server.

4 Conclusions

In this thesis, we have dived into the challenges that arise during the design phase of software projects, particularly those related to the manual translation of design decisions into various files.

The repetitive and time-consuming nature of these manual tasks not only consumes valuable developer time but also introduces the potential for human error, which can result in design incoherence. To address these issues, this project was focused on creating an automation tool aimed at streamlining the design process and reducing human error.

The tool that has been developed assesses various aspects of the design process, from checking design and implementation coherence to automating the generation of design documentation. It provides a pipeline that systematically ensures the alignment of design decisions with their implementation, thus minimizing the risk of inconsistencies that could lead to complex and time-consuming issues during the implementation phase.

One of the significant achievements of this project is the successful testing and validation of the static check pipeline through all the test cases. The tests consistently gave correct results, affirming the reliability and effectiveness of the automation script in enhancing the design process's efficiency and accuracy.

Moreover, the integration of the pipeline into Azure, while tested with limited access to repositories, proved itself to be reliably executed whenever changes were made through git commands, automating the testing process, and providing developers with accessible output for evaluation.

In addition to its static pipeline, the dynamic pipeline proved its effectiveness by accurately extracting data and conducting coherence checks within the limitations of the test project. The successful setup and execution of the pipeline, coupled with the provision of output through a server. Nonetheless, the dynamic pipeline has much room for improvement.

The dynamic pipeline implemented was not completely fulfilling the objectives of the thesis and would need more work to do so. This pipeline still relies on developers to set it up by modifying the source code of the project to be tested. It also doesn’t check the coherence with the High-Level design, since it only extracts data. Therefore, this part of the project could improve to be more functional for the company. It could be done by making a tool that compares the output of the data extraction to the High-Level design values.

In conclusion, this project has achieved its objectives of simplifying and shortening the design process for software projects. The developed automation tool has the potential to significantly reduce the time and effort invested in manual design-related tasks, minimize the risk of design incoherence, and ultimately contribute to the efficiency and quality of software implementation.

5 References

[1] Webpage: https://www.autosar.org/standards. [AUTOSAR] 04/23
[2] Webpage: https://www.autosar.org/standards/classic-platform [AUTOSAR Classic Platform] 04/23
[3] Webpage: https://www.synopsys.com/glossary/what-is-devops.html [DevOps] 05/23
[4] Webpage: https://aws.amazon.com/devops/what-is-devops [DevOps] 05/23
[5] Webpage: https://piembsystech.com/automotive-ecu/ [ECUs] 08/23
[6] Webpage: https://learn.microsoft.com/en-us/azure/devops/user-guide/services?view=azure-devops [Azure DevOps] 08/23
[7] Webpage: https://en.wikipedia.org/wiki/Software_development_process [Softwre development phases] 08/23
[8] Webpage: https://en.wikipedia.org/wiki/Software_design [Design process] 08/23
[9] Webpage: https://en.wikipedia.org/wiki/High-level_design [High-Level Design] 08/23
[10] Webpage: https://sparxsystems.com/ [Enterprise Architect] 08/23
[11] Webpage: https://en.wikipedia.org/wiki/Component-based_software_engineering [Software Components] 08/23
[12] Webpage: https://pythonhosted.org/comtypes/ [comtypes] 08/23
[13] Webpage: https://www.vector.com/int/en/company/about-vector/ [Vector] 08/23
[14] Webpage: https://www.vector.com/int/en/products/products-a-z/software/canoe/?gclid=Cj0KCQjwusunBhCYARIsAFBsUP8YdC_eDkTN7_ElFep0YcaAHaIy9f2MaVcGFFZF_bIe2PCz1x8MDOUaAvMLEALw_wcB# [CANoe] 08/23