Judit Ferran Llaberia

Design and development of an Adaptive AUTOSAR application to perform over-the-air flashing updates

Bachelor's Thesis

Academic tutor: Dr. José Luis Ramírez Falo

Professional tutor: Javier Cuenca Granados

Bachelor's Degree in Industrial Electronics and Automation Engineering

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2023

This bachelor's thesis contains confidential information owned by Lear Corporation, with registered address at Carrer Fusters, 54, 43800 Valls (Tarragona).

The current document is the reduced version, confidential information has been eliminated.

Index

List of figures

List of tables

1. Introduction

This project was developed during my internship at Lear Corporation. Its objective has been to design and develop an Adaptive AUTOSAR application to perform over-the-air (OTA) flashing updates on vehicle electronic control units (ECUs), as well as implementing a software package containing the software to install. In more general terms, the development of this project led to explore the possibilities of the Adaptive AUTOSAR platform, focusing on one of its main characteristics: the possibility to develop the software dynamically, enabling the updatability of applications once they are already deployed in the vehicle ECUs.

1.1. Objectives

- Analysis of the AUTOSAR Standard, specifically the Update and Communication Management functional cluster.
- Development of a demonstrative application of the Update and Configuration Manager functional cluster, establishing the communication using Inter-Process Communication (IPC) and Scalable Service-Oriented Middleware over IP (SOME/IP) protocols.
- Creation of a Software Package and the pertinent cybersecurity mechanisms.
- Send, receive, and process the Software Package.

1.2. Scope

The automotive industry is rapidly evolving with the integration of advanced electronic systems in vehicles. To meet the increasing demands for safety, reliability, and software complexity, standardized frameworks have emerged. AUTOSAR is the global established standard for software and methodology on electronic systems, used by all different companies involved in the development and production process in the vehicle industry. This standardized approach not only simplifies the development process but also allows for efficient collaboration among different parties involved in the automotive ecosystem, including Original Equipment Manufacturers (OEMs), suppliers, and software developers. In an industrial context, production cost, time constraints, and efficiency play crucial roles, particularly in the development and production process of vehicles. A global framework provides software modularity and standardization, which have a significant impact on achieving these goals.

In response to the latest advancements in the automotive industry, AUTOSAR introduced a new platform in the standard: AUTOSAR Adaptive. Whereas the first platform of the standard, renamed AUTOSAR Classic after the release of Adaptive, focuses on embedded systems, Adaptive is designed to address the unique requirements of highly complex and interconnected electronic systems found in modern vehicles. Automated driving, electrification, connectivity are emergent areas in the automotive industry, and the Adaptive architecture is specifically designed to fulfill the demands of these advanced technologies.

This document aims to provide the reader with an insightful understanding of AUTOSAR, specifically its Adaptive Platform, encompassing its goals, characteristics, and architectural framework. While not intended to be an exhaustive study of the standard, it comprehensively presents the framework and its importance.

AUTOSAR, provides requirements and specifications, but does not implement the system itself. Instead, it establishes a common platform, fostering collaboration among various companies involved in the development and production processes of the vehicle industry. Producers can acquire pre-implemented software components and solutions from specialized providers within the AUTOSAR ecosystem, allowing companies to focus on their specific areas of expertise. This division promotes efficiency and specialization within the industry.

For the development of this project, a Software Integration Package (SIP) has been used. This SIP, acquired form a software provider, is a set of tools, basic software and libraries that allows for the development of AUTOSAR Applications. The software contained in this SIP is based in the AUTOSAR Adaptive R20-11 release, so the project has been developed according to those specifications. The kind of SIP used is intended for development and test purposes, not a commercial use, and because of that may contain elements whose functionality is not fully implemented or tested.

One of the differentiating characteristics of the AUTOSAR Adaptive Platform is its ability to facilitate continuous updates of ECU software throughout the vehicle's lifecycle. Over-the-air updates (OTA), enable software updates to be performed without the need for a physical wired connection. To initiate an OTA update, the vehicle must establish a connection to a backend system, through wireless technologies like Wi-Fi or 5G, and download a Software Package. Once in the vehicle, this package needs to be processed and sent to the ECU to be updated. Finally, the new software is installed, replacing the previous version.

This project focuses on the parts of the update process where AUTOSAR architecture is most relevant: how the software update is managed once the package is locally available in the vehicle. AUTOSAR establishes how this Package is transmitted between ECUs, processed, and activated, as well as its format. An application to perform this process will be implemented, capable of establishing communication between ECUs or components. The application will be modeled, build, and eventually coded. The software package, whose format shall follow the standard specifications, will be created. Moreover, is also relevant the protection of the Software Package against malicious attacks, as this protection mechanisms are intricately linked to the package format. For the software package cybersecurity mechanisms will be implemented, consisting in a signature and key pair.

2. AUTOSAR Adaptive

2.1 Introduction

AUTOSAR is a partnership of automotive industry companies whose objective is to develop and establish an open architecture framework for the electronic control units of the vehicles. AUTOSAR is used by OEM manufacturers, Tier 1 suppliers, semiconductors manufacturers, software suppliers and tools suppliers.

The AUTOSAR partnership was initiated in 2003 and its Classic Platform architecture was drafted. During the initial phase of AUTOSAR different partners joined the organization, and the structure and specifications of the AUTOSAR Classic Platform were defined. The first release of the platform took place in 2005. The constant evolution of the automotive industry led to the development of the Adaptive Platform, first released in 2017. Nowadays AUTOSAR is used all around the world, twenty-one out of the twenty-two top-selling international OEM are AUTOSAR partners.

AUTOSAR provides a layered architecture that separates software functionality from hardware dependencies. It defines a set of standardized interfaces, data structures, and protocols, allowing different software components to communicate with each other. This modular approach enables the development of complex automotive systems by integrating various software components from different suppliers.

By adopting AUTOSAR, automotive manufacturers and suppliers can collaborate more effectively, reduce development time and costs, enhance software quality, and enable the integration of third-party software components. It promotes the scalability and flexibility of automotive systems, allowing for easier updates, upgrades, and customization throughout the lifecycle of the vehicle.

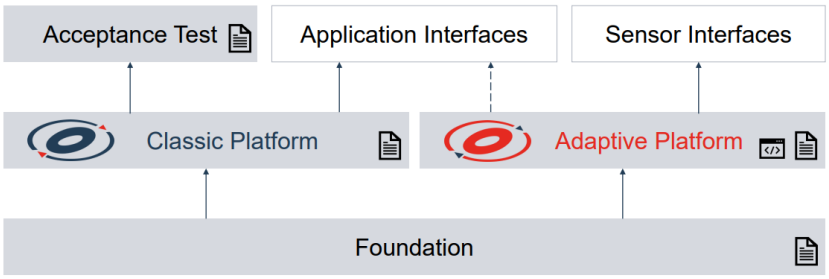

Figure 1: AUTOSAR Platforms

Currently AUTOSAR consists of three platforms: Classic, Adaptive and Foundation. The Classic and Adaptive Platforms, along with the AUTOSAR Foundation, work together to facilitate the development of standardized and scalable automotive software systems across different hardware platforms, serving different application domains within the automotive industry.

The **Classic Platform** is the original implementation of AUTOSAR and is primarily targeted at microcontroller-based ECUs and deeply embedded systems, with hard real-time constraints. This ECUs directly interact with sensors and actuators while operating under strict real-time constraints.

AUTOSAR Classic follows a layered architecture that separates the application software from the underlying hardware. It consists of three main layers: the Application layer, the Runtime Environment, and the Basic Software (BSW). The BSW further comprises three additional layers: the microcontroller abstraction layer, the ECU Abstraction Layer, and the services layer. This layered architecture allows the software to be hardware independent.

Since the first release of AUTOSAR new technologies emerged on the automotive industry. The **Adaptive Platform** (AP) is specifically developed to address the emerging requirements introduced by recent and future functionalities in vehicles, such as automated driving, Advanced Driver Assistance Systems (ADAS) or vehicle connectivity. These functionalities introduce new demands that the Classic Platform is unable to fulfill. The Adaptive Platform offers high-performance computing and high data volume communication mechanisms to meet these requirements. Adaptive, as the Classic Platform follows a layered architecture, as it will be further explained in section 2.5. Figure 2 shows both Classic and Adaptive architectures.

Despite sharing a layered architecture, AUTOSAR Classic and AUTOSAR Adaptive platforms are significantly different in several aspects. In AUTOSAR Classic the whole software stack, as well as the configuration, is compiled and linked as a single piece, whereas Adaptive offers a more flexible approach. Services run as processes, which can be installed separately, and configuration is load in form of manifest files. Regarding communications, AUTOSAR Classic utilizes proprietary communication mechanisms for inter-ECU communication, but Adaptive introduces service-oriented communication, such as Service-Oriented Middleware over IP (SOME/IP), enabling more flexible and scalable communication between ECUs.

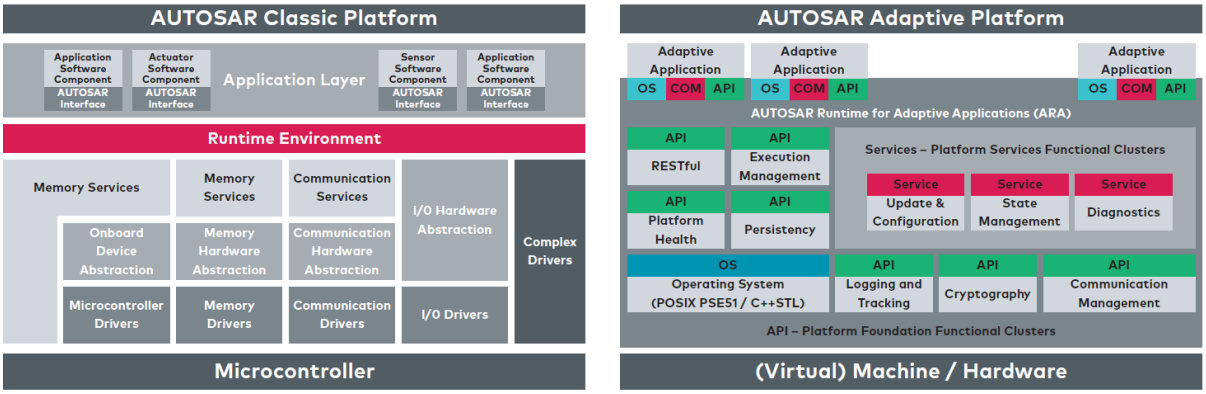

Figure 2: AUTOSAR Classic vs. Adaptive Architectures

AUTOSAR Foundation ensures interoperability between Classic and Adaptive platforms. Foundation establishes common requirements and specifications, such as protocols, shared between the AUTOSAR platforms.

It is important to note that the development of a new architecture does not imply the replacement of the existing Classic Platform. Instead, both architectures coexist within the same vehicle, serving different proposes based on their unique characteristics. Even in the case of a fully autonomous vehicle, where the need of the Adaptive Platform is evident, there will still be the need for Electronic Control Units that support high-safety and deeply embedded systems, remaining the Classical AUTOSAR Platform the architecture of choice for such applications.

2.2 Requirements and goals

The need to develop a new platform arises from the inability of the Classical Platform to fulfill the new requirements of the upcoming systems. Then, it is necessary to take into consideration what those new requirements are when designing the new platform. Seven basic requirements are defined, stablishing the base for the design of the new architecture:

Support of state-of-the-art technology

The Adaptive Platform shall support applications that requires a significant amount of systems resources, such as memory and CPU power, on advanced state-of-the art hardware. Because of that, the platform is designed to support high performance computing platforms and virtualized environments. The platform enables the simultaneous operation of multiple applications, each with their own concurrent internal control flows.

Software update and configuration

The Adaptive platform significantly differs from the Classical platform in its support for flexible data and software updates, including its configuration. Specifically, the Adaptive platform enables the uploading of update packages, allowing for changes to communications and applications while the system is running. Furthermore, AUTOSAR provides a unified approach for describing software systems deployed on both Adaptive and Classic platforms, facilitating the deployment and reallocation of applications, and providing a way to describe interfaces of the entire system.

Security

Security refers to measures taken to protect systems, data, and resources from unauthorized access. Measures are implemented to guarantee confidentiality and integrity of information.

The AUTOSAR Adaptive Platform is designed to support the development of secure systems, ensuring secure access to ECU data and services, as well as providing secure onboard communication.

Safety

Safety is concerned with the prevention of harm or injury, to individuals, the environment or property. In the automotive industry, safety implies the development of reliable systems and implementation of measures to minimize the risk of accidents, ensuring the proper functioning of critical systems.

The AUTOSAR Adaptive Platform supports the development of safety-related systems that prioritize reliability and high availability. The platform's specifications are structured to be analyzable and support methods for demonstrating the fulfillment of safety-related properties accordingly to the requirements and specifications of the architecture.

Reuse and Interoperability

To achieve this, Adaptive shall provide a comprehensive component model description for application software and support for various programing languages, enhancing flexibility and enabling developers to work with the more adequate programming language. Overall, the focus on reuse and interoperability in the AUTOSAR Adaptive Platform promotes a modular and versatile approach, allowing for efficient development, integration, and compatibility.

The platform shall support interoperability with software that is not part of the AUTOSAR platform as well as portability between different implementations of the AUTOSAR platforms.

Source code portability allows applications to be easily transferred and executed across different platform implementations.

Communication

Adaptive shall support the standardized automotive communication protocols, for efficient communication within the Electronic Control Units of the vehicle and support different network topologies, whereas it may be centralized or distributed network architectures, leveraging the standardized communication protocols and network topologies and supporting backward compatibility.

Diagnostics

The new platform shall continue to offer diagnostic features that can be used during runtime with both production and service-related proposes.

2.3 Technology drivers

The fulfillment of the new demanding requirements for the Adaptive Platform is possible thanks to the development and implementation of two key technologies to the vehicle, both related to the increased data volume that needs to be handled: Ethernet and high-preforming processors.

The new features have increased data speed demand on vehicle network, to fulfill this demand ethernet has been introduced to the vehicle. Automotive Ethernet offers higher bandwidth over the traditional technologies used in vehicle communications, such as CAN-BUS. Ethernet, implemented as IEEE 100BASE-T1 or 1000BASE-T1, can reach 100 Mbit/s or 1 Gbit/s, respectively, while CAN operates at speeds of up to 1 Mbit/s. Ethernet higher speed enables to transfer longer messages in a more efficient way.

This increase in data requires more powerful processors, as it is not only necessary to transmit it but also to process it. The way to improve the performance of ECU is the use of multicores and parallel execution.

Although the Classic Platform supports Ethernet it is not optimized for it making it difficult to fully benefit from the higher speed of Ethernet communications.

2.4 Characteristics

From the chosen programing language to its development methodology, this chapter aims to give an insight on the main AUTOSAR Adaptive characteristics.

The requirements set defined for the AUTOSAR Adaptive platform influenced and shaped its characteristics. Ensuring safety, power efficiency, and cost effectiveness poses new technical challenges that the platform addresses by incorporating established technologies relevant to meet the increased computing needs.

C++: C++ is the language of choice for programming Adaptive applications. Starting from a high-level perspective, C++ has emerged as the preferred programming language for developing new algorithms and application software in performance-critical and complex domains within the software industry.

The adoption of C++ offers several advantages, including the ability to leverage its extensive features and libraries for efficient algorithm implementation. The language's rich set of features, including object-oriented programming, templates, and powerful standard libraries, enables developers to write clean, modular, and reusable code. This facilitates rapid

development and maintenance of complex software systems, increasing productivity and reducing time-to-market.

Service Oriented Architecture (SOA): To support complex applications and ensure maximum flexibility and scalability in processing distribution and resource allocation, the Adaptive Platform adopts a service-oriented architecture.

In SOA a system comprises a collection of services, where one service may utilize another, and applications can use one or more services based on their specific requirements. Services are provided by application components through a service interface, over a network. Server components may offer services to its clients, who will subscribe to or use them.

Service oriented architecture has the characteristics of system-of-system, which are also present in Adaptive. For example, a service can be located on a local ECU where an application is running, or it can reside on a remote ECU that is also utilizing an instance of the AP. Regardless of the location, the application code remains the same, and the communication infrastructure handles the differences, ensuring transparent communication between services.

Parallel processing: Parallel processing in distributed computing allows for efficient utilization of computing resources. The SOA of Adaptive aligns well with this parallelism, as different applications can share sets of services. With the advancements in multicore processors and heterogeneous computing, there are opportunities to leverage this parallel computing power. Adaptive is designed to scale its functionality and performance as these technologies evolve.

Leveraging existing standard: AUTOSAR Adaptive choses to adopts existing standards instead of inventing new ones, whenever it is possible, in order to prevent unnecessary duplication and complexity. By leveraging these established standards, Adaptive can expedite its development process and take advantage of the existing ecosystems that surround these standards, while ensuring compatibility with established standards and promoting interoperability.

Safety and security: Adaptive will be used in vehicles systems where safety and security must be guaranteed, as ensuring the integrity of the vehicle is crucial. The combination of architectural, functional, and procedural approaches taken by AUTOSAR makes the platform inherently safe and secure.

The SOA design enhances the independence and reduces unintended interferences among components, promoting safety and security. Furthermore, the AP provides guidelines, such as the C++ coding guideline, which facilitate the safe and secure usage of complex programming languages like C++. Adaptive incorporates dedicated functionalities specifically designed to assist in achieving safety and security objectives.

Planed dynamics: Adaptive is designed in a way that supports the incremental deployment of applications. By reducing the complexity of software development and integration, this approach enables shorter iteration cycles and encourages exploratory software development phases

2.5 Architecture

AUTOSAR Adaptive architecture has been designed to solve the requirements and goals previously determined for the adaptive Platform. The AP architecture is layered and allows for flexible and dynamic and scalable deployment.

2.5.1 AUTOSAR Runtime for Adaptive applications

The higher level of the Adaptive architecture is the application layer. AUTOSAR applications run on top of the AUTOSAR Runtime for Adaptive applications (ARA). ARA consists of application interfaces provided by Functional Clusters, which are part of either the Adaptive Platform Foundation or the Adaptive Platform Services. Figure 3 shows AUTOSAR layered architecture. On ARA layer, grey boxes are Platform Services and blue boxes Platform Foundation. Adaptive Applications (AA), represented as orange boxes on the figure, can also provide Services.

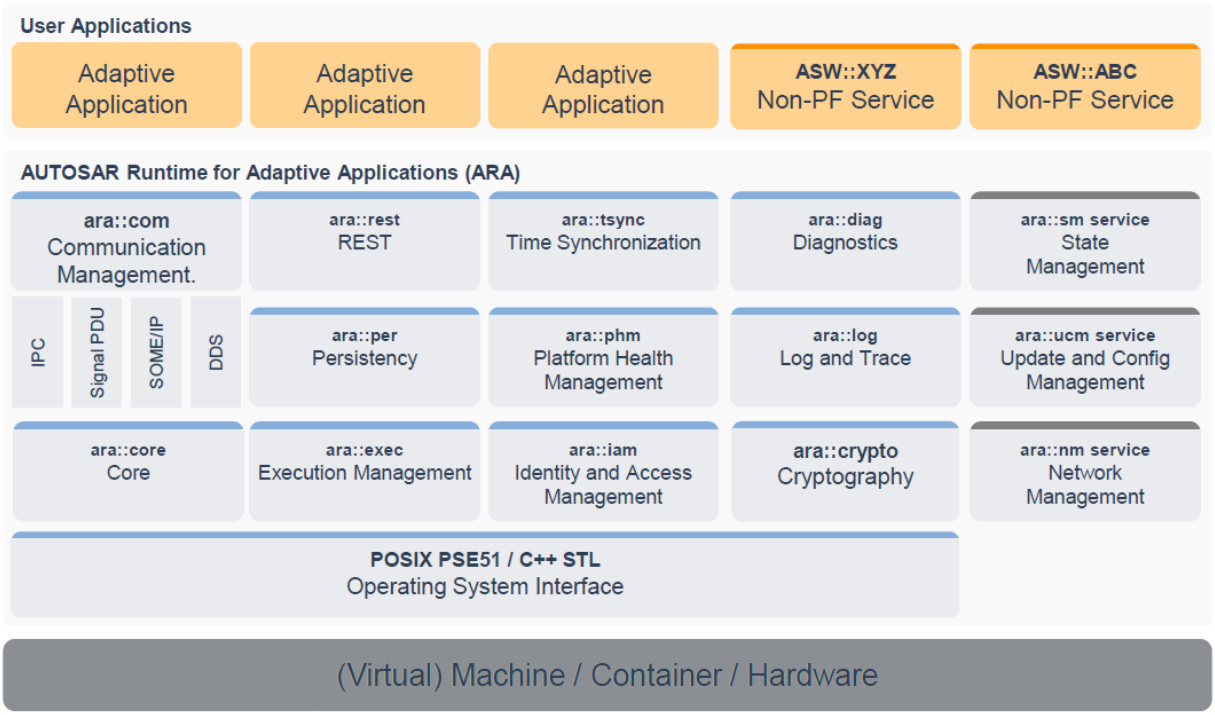

Figure 3: AUTOSAR Adaptive architecture

2.5.2 Functional clusters

A Functional Cluster is a component responsible for providing specific functionalities within the software architecture. It serves as an interface between the Adaptive Applications and the underlying infrastructure of the system.

Functional Clusters are categorized into two types: those belonging to the Adaptive Platform Foundation and those belonging to the Adaptive Platform Services. The Functional Clusters of the Adaptive Platform Foundation offer fundamental functionalities that form the basis of the Adaptive Platform. These functionalities include core features and services required for the execution of Adaptive Applications.

On the other hand, the Functional Clusters of the Adaptive Platform Services provide platform standard services for Adaptive. These services offer additional capabilities that can be leveraged by Adaptive Applications, such as communication protocols, diagnostic functionalities, or security services.

The following Functionals Clusters have been relevant in the development of this project:

- Execution Management: Responsible for managing the execution lifecycle of Adaptive Applications, including starting, stopping, and monitoring application instances.

- State Management: Handles the management and synchronization of application states, enabling the preservation and restoration of application data during runtime.

- Cryptography: Provides cryptographic services such as encryption, decryption, and secure key management to ensure data confidentiality and integrity.

- Persistency: Manages persistent storage and retrieval of application data, allowing data to be stored and accessed across system restarts.

- Update and Configuration Management: Handles the management of updates and configurations for Adaptive Applications, enabling dynamic updates and flexible configuration options.

2.5.3 POSIX Operational system

POSIX stands for Portable Operating System Interface for UNIX. It is a family of standards (the IEEE Std 1003.n and parts of ISO/IEC 9945) that define a standard operating system interface and environment for compatibility between different operating systems. Thanks to POSIX standards, software written for one POSIX-compliant operating system can run on any other POSIX-compliant operating system without requiring significant modifications. (Linux, PikeOS, QNX, Integrity, ... are POSIX OS)

Adaptive, following its characteristics of leveraging existing standards, does not define a new operating system (OS), but defines that the used OS in the Adaptive Platform must be POSIX-compliant. Using an POSIX OS enables application portability between different systems.

This abstraction form the OS offers several advantages. It offers flexibility during the development process of an application, where the OS used can differ from that employed in the ECU of the vehicle, and so becoming independent of specialized hardware. For example, Linux offers a convenient environment to develop Adaptive applications.

POSIX defines different profiles, being the PSE51 the one created for embedded real-time systems. For the Adaptive Platform, PSE51 shall be used as OS interface, however applications may not be limited to this subset and extended beyond its limitations within the POSIX standard.

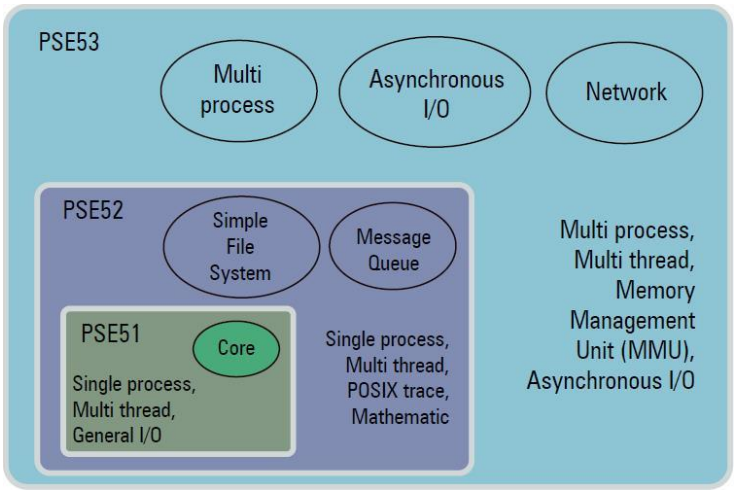

Figure 4: POSIX profiles subsets.

2.5.4 Communications

One of the characteristics of AUTOSAR is that it is a Service-Oriented Architecture (SOA). In an SOA, applications are built by services, which are self-contained and independent components that can be accessed and utilized by other applications. Services provides functionalities beyond the already provided by the operative system. AUTOSAR Adaptive defines a functional cluster, Communication Management which is responsible for all aspects of communication between applications.

A service consists of events, methods, and fields; and is provided through a Service Interface.

The Communication Management software provides mechanisms to offer and consume services, and to establish communication. This communication can be established at design, at startup or at runtime. In the process of Service Discovery applications that provide services registers them at the service register, where applications that consume services can find them.

The Communication Management ensures standardization in presenting services to developers and representing service data on the network. It supports different language bindings for convenient access to service methods, events, and fields. The Communication Management currently supports bindings such as SOME/IP, DDS, IPC, Signal PDU, and custom bindings. The Network Binding handles the serialization and network representation of service data. In the context of C++ Language Binding, proxies and skeletons are generated to facilitate communication between clients and service providers. Proxies offer synchronous and asynchronous calling mechanisms, while skeletons define the implementation of service functionality.

2.6 Update and Configuration Manager

2.6.1 Description

One of the main objectives of the AUTOSAR Adaptive Platform is to allow continuous updates of the software and its configuration in a wireless manner, through over-the-air updates.

Over-the-air (OTA) updates allows the user to remotely upgrade their vehicle’s software in the most convenient moment, avoiding taking the car to time-consuming service garage visits. For vehicle manufacturers, OTA opens the possibility of adding additional features that were not available in the time of vehicle purchase and provides an efficient and convenient way to correct bugs across all vehicles simultaneously.

Wireless communication technologies like UMTS, LTE, Bluetooth, Wi-Fi, and 5G can be used to establish wireless connectivity between the vehicle and a backend cloud system. These connections enable the vehicle to download and install software updates from the cloud and ensure that the fleet's software remains up to date.

The Update and Configuration Manager (UCM) is the functional cluster responsible for managing the software updates and configurations throughout the vehicle's lifecycle, performing modifications while maintaining consistency and functionality of the entire system. The UCM provides an Adaptive Platform service to acquire software information and execute software updates.

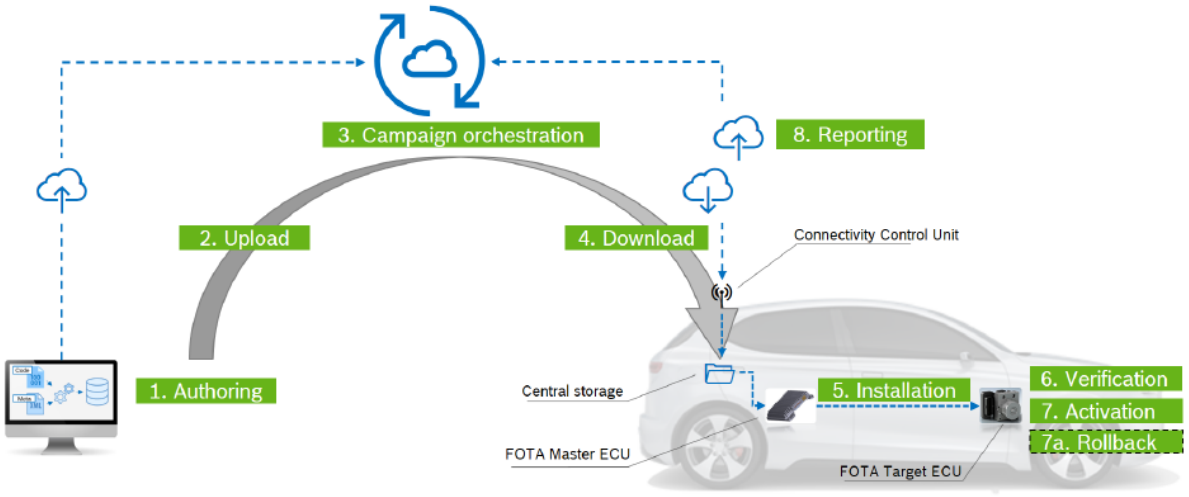

Figure 5: Firmware Over-the-air

Even though the process to perform a software update involves several steps and elements, as it is represented in Figure 5, the Update and configuration Manager is only responsible of managing the process within vehicle ECUs. UCM does not stablishes how data is transferred to the Adaptive platform or how package processing is controlled.

Nowadays vehicle Electronic Control Units (ECUs) may already have on-boar reprogramming capability but is only used in service garage typically to fix bugs or deploy functional improvements. This currently implemented interface could be used by Update Manager master ECU to install new software, even when the vehicle is not physically connected via wires, such as in a service garage setting.

The Update and Configuration Manager will be present in all Adaptive ECUs capable of being updated. The service interface is designed so diagnostic services, the standard component to perform wired software modifications, can be used for downloading and installing software updates to an Adaptive Platform. As it can be seen in Figure 6, the UCM abstracts whether the update is being performed through an OTA client or through diagnostics.

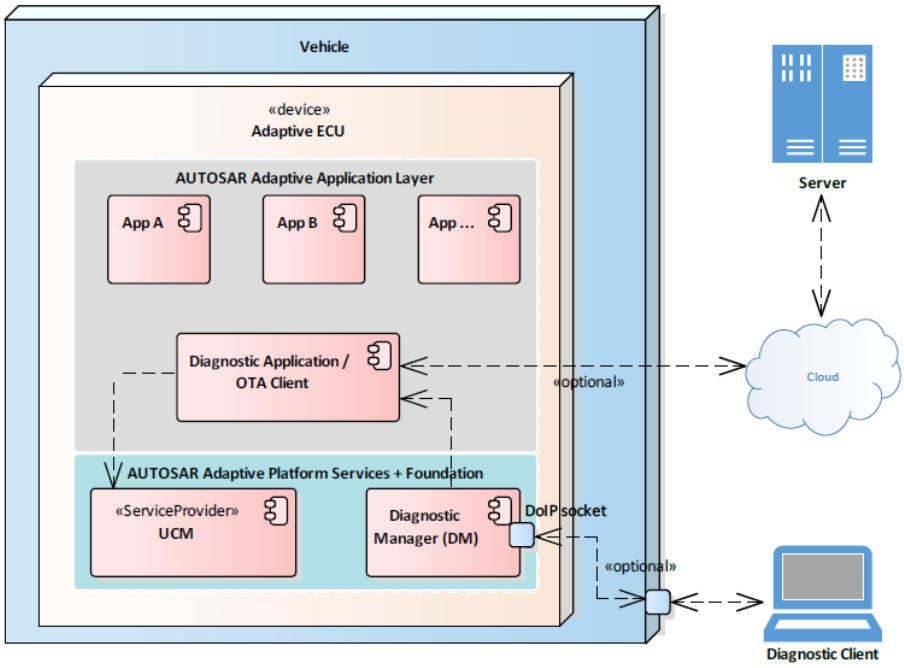

Figure 6: Architecture overview for software update

2.6.2 Components and responsibilities

From a software component viewpoint, it is necessary to stablish an UCM Client and an UCM Manager. Note that these components may be referred with different names depending on the documentation and use case, for the UCM Client it is usually used the name UCM Master when the update is performed via OTA, and Diagnostics Application for a wired update.

In Figure 7 the software components that are involved in the update process are represented. Although only the UCM Client and the UCM Master belong to the UCM functional cluster, to successfully install software several elements are involved. Those dependencies will be explained later on.

The UCM Client and the UCM Manager fulfill different requirements within the vehicle. In the car, only one active UCM Client will be found, whereas each updatable Adaptive ECU will have an instance of a UCM Manager. In case the UCM Client fails, it is possible for an UCM Manager to take its role, as the UCM Master could be considered an UCM instance with added features.

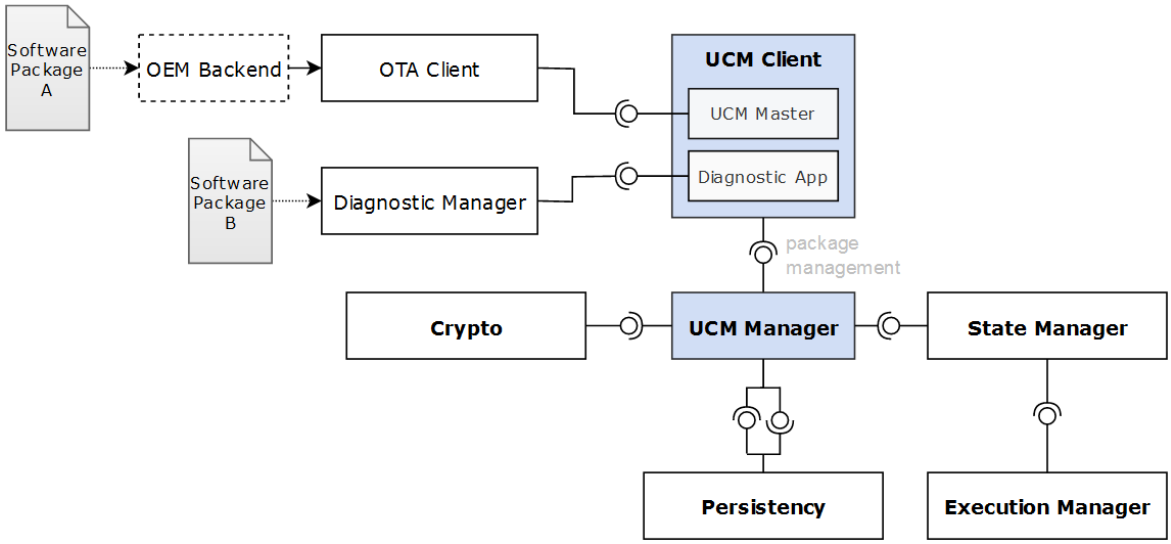

Figure 7: Components involved in the update process

The UCM Master is responsible of coordinating the update process, including receiving packages from the backend or diagnostic tools, parsing, and interpreting them. According to the information received in the software package, it is transmitted or streamed to the corresponding targeted ECU, and its processing, activations, and potential rollbacks orchestrated. All the updating process with its defined characteristics is referred as an "update campaign".

The UCM Master interfaces with various applications within the vehicle. While it acts as a client of the service interface offered by UCM subordinates, it also provides three different service interfaces to the OTA Client, Vehicle Driver interface, and Vehicle State Manager.

The OTA Client is the bridge between the UCM Master and the Backend, and the way through software packages reach the UCM. This communication is as well used to exchange information about the installed software and the available one. The communication between the Backend and the UCM could be triggered by both parts: the backend could push a security update or the UCM Master could request an update or check for new available software clusters.

The UCM Client is also the component that will determine if it is possible to perform an update in a specific time. It could be necessary to interact with the driver of the vehicle to get consent to download an update, as it could have an impact in data transfer cost, or to put the vehicle in a safe state, to guarantee safety it could be asked to stop the vehicle and shutdown the engine.

In Figure 8 those components are represented in the context of the vehicle architecture. Purple boxes represent physical hardware ECUs, and red boxes Adaptive AUTOSAR Machines. In this example, within the same ECU we can find several components of the UCM and shows the concept of UCM Master, being a special UCM instance, communicating with an UCM.

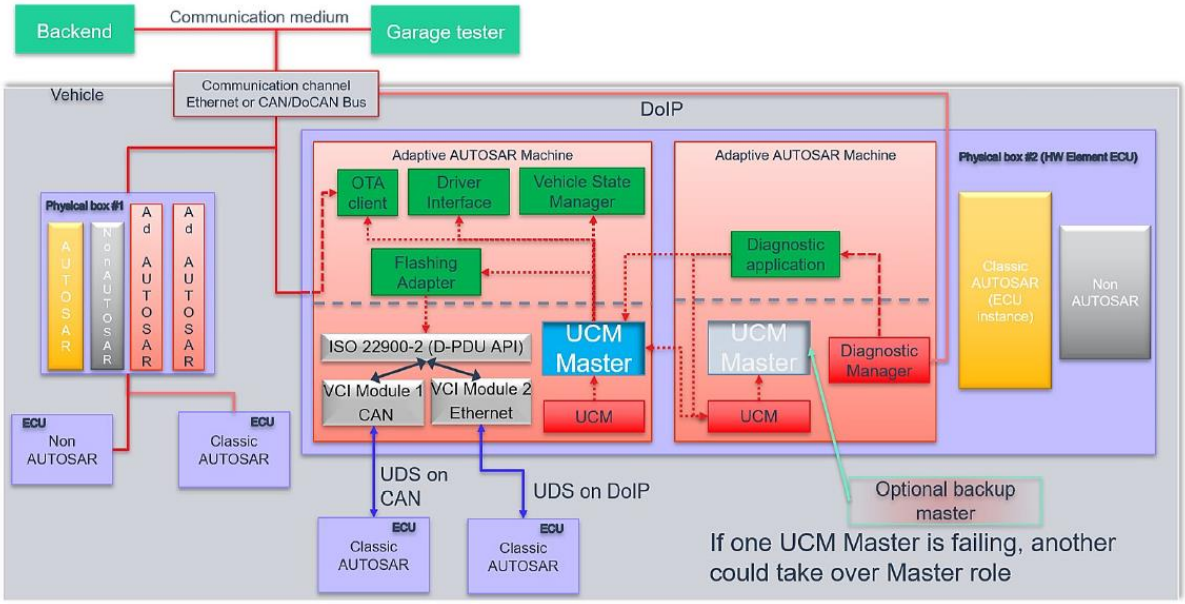

Figure 8: Example of UCM Master architecture overview within a vehicle

To summarize, UCM Master responsibilities are the following:

- Coordinate software update in a vehicle across multiple Electronic Control Units.
- Use platform communication services for interacting with UCM subordinates.
- Ensure it is safe to perform any modification to the vehicle.
- Interact with driver.
- Orchestrate a software update campaign according to the Vehicle Package's Manifest.

And the general responsibilities the Update and Configuration Manager must fulfill:

- Versions reporting.
- Data management.
- Data transfer.
- Software Updates.
- Logging, progress, and status reporting.
- Validation.

2.6.3 Software Package

The element used to contain the software to be updated is the software package. This element is what the UCM shall take as an input.

The Software Package serves as a container for all the necessary elements, including manifests, executables, and other data, needed to manage and update a specific Software Cluster within the software system, such as executables of Adaptive Applications, Kernel or firmware updates or updated configurations. A Software Package can contain several executables, but it is also possible to have no executables when the aim is to update the configuration metadata.

Depending on the role of the Software Cluster, additional data relevant to that particular cluster may also be included in the Software Package. This additional data could be configuration files, resources, or any other necessary files that contribute to the functioning of the Software Cluster.

Each software package shall address only one software cluster, but Software Cluster is not sufficient for itself for installation, it needs to be warped into a Software Package containing a standardized manifest. Whereas AUTOSAR standardized some aspects of the software package, the packaging tool is vendor specific, so it has to be taken into consideration when developing the UCM.

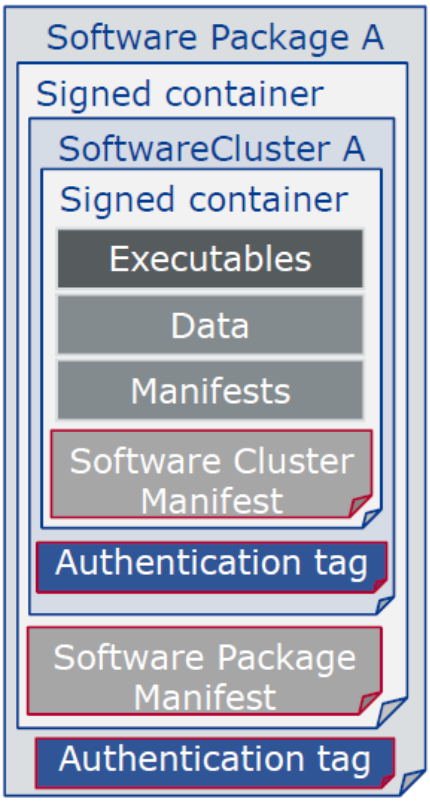

Figure 9: Software Package content description

2.6.4 Updating process

Once a software update is available in the Adaptive Platform the updating process can be started, tigered by an update request. This process has been divided in three distinct phases: data transmission, processing, and activation.

Data transmission: This phase involves transferring one or more Software Packages from the UCM's Client Application to the internal buffer of the UCM. The Software Packages contain the necessary software components for the update. This transfer allows the UCM to access and work with the Software Packages during the update process.

Processing: In this phase, the UCM performs the specified operation (such as installation, update, or removal) on the relevant Software Cluster. The Software Cluster represents a collection of software components that are grouped together. The UCM executes the necessary actions to process the Software Package and apply the desired changes to the Software Cluster. These changes could also involve modifying configuration data, calibration data or manifests. Unlike the first phase only one Software Package can be processed at the same time.

Activation: This phase involves several steps. First, the UCM checks the dependencies of the Software Clusters involved in the operation, referring to the relationships and requirements between different software components. After verifying the dependencies, the UCM activates the Software Clusters, making them ready for execution. Finally, the UCM performs checks to ensure that all the Software Clusters can be executed properly, typically through State Management. This verification step ensures that the updated software functions as expected.

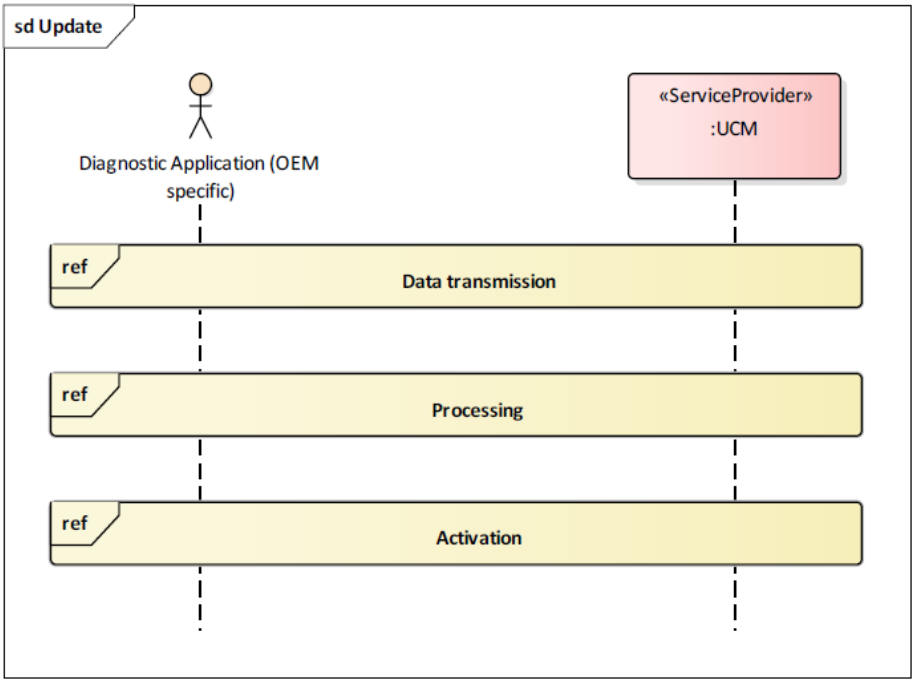

Figure 10: Update process sequence diagram.

Sequence diagrams for all three phases of the update process can be found in Annexes 7.1.

3. Work environment

The content of this chapter has been reduced for confidentiality reasons.

3.1 Software Integration Package

A Software Integration Package integrates software, tools, and demonstrative applications to facilitate the process of creating adaptive applications. AUTOSAR defines the software components and architecture, however, does not implement them. Because of that, there are companies that specialize in creating software solutions to make the development of applications more convenient. As previously explained, the basic software components are reusable and hardware independent, it would not make sense for every manufacturer to develop their own, it is safer and more efficient to purchase the necessary libraries, tools, and Adaptive software components necessary to develop applications.

The SIP is compatible with the AUTOSAR R20-11 version, so it meets its specifications and requirements.

3.2 Authoring tool

The development environment (IDE) used is a modified version of Eclipse for creating Adaptive applications. It allows to model applications as well as programing them. It offers diverse ways of viewing and editing the components and integrates tools for validating and configuring the projects.

When launching the IDE, a workspace needs to be selected. If a workspace does not exist, one can be created. Then the first step is to configure the SIP in Eclipse.

Figure 11: Configuration of the SIP in the authoring tool

The next step is to add an application to the workspace. It is possible to create a new application, however, an existing demonstrative application will be used instead as the starting point of a new project. First, a copy of the example application is created and the folder containing it renamed to the desired new name. Every application has a CmakeLists.txt, in this file the target name needs to be modified to match the new name of the application. At this point, the new project can be added to the workspace from the file menu.

To be able to utilize the Adaptive functionalities of the authoring tool the project needs to be converted to an Adaptive project.

By configuring a project as an Adaptive application, the IDE is able to interpret its files and present them in user-friendly way to the developer.

Figure 12: Authoring tool dashboard.

Different view modes for top level elements are available in the AUTOSAR Model Explorer: the model can be presented as *AUTOSAR elements* or as *AUTOSAR Packages.*

Independently of the view mode, the language or details mode can be presented in two different ways, as a *Form* or as a *DML (Model Language).*

Figure 13: Model Explorer, AUTOSAR Elements

Figure 14: Model Explorer, AUTOSAR Packages

In addition to the modeling functionalities, the authoring tool can also be used as a programing tool, allowing the developer to write and edit the code.

3.3 Build Helper

The Build Helper is a tool provided in the SIP that builds the components of the Adaptive AUTOSAR Stack. It builds the components in the correct order and knows the dependencies, making the building process more convenient.

The building configuration for the Build Helper is done through a JSON file. The file can be found in the Build Helper's configuration directory and contains the recommended options for building applications during the development. Those options can be modified as needed.

To use the build helper the terminal shall be opened in the SIP directory and Command 1 modified according to the name and desired options for the application to be build.

```
BuildHelper/BuildHelper -s . -b BuildLearUCM/ --install BuildLearUCM/install --
cmake-toolchain-file CMakeSupport/toolchain/gcc/gcc-
9/x86_64/linux_debug.toolchain --cmake-options-file
BuildHelper/config/development.json --model-dir Examples/app-lear-ucm/ -t app-
lear-ucm
```

Command 1: Building an application with "Build Helper" in a Linux OS.

Before running the applications, the network must be configured. The IP used by the application must be set and multicast communication enabled, this is done using Command 2.

```
sudo ifconfig enp0s31f6 192.168.7.2 && sudo ip link set lo multicast on && sudo
ip route add ff01::0/16 dev lo
```

Command 2: Network configuration.

3.4 Execution Manager

The application is executed by the execution manager. The terminal shall be opened in the "install" directory of the built application to execute, and Command 3 used to execute it. It is necessary to introduce the options stating the directories of the machine execution configuration file and the logging configuration, through the options "-m" and "-l", which are typically found in the "etc" folder. A more thorough explanation of the directory structure and configuration files will be given later in the document.

```
sudo ./sbin/amsr_em_executionmanager -a ./opt -m ./etc/machine_exec_config.json
-l ./etc/logging_config.json
```

Command 3: Execution of an Adaptive application.

4. Implementation

The content of this chapter has been reduced for confidentiality reasons.

This chapter focuses on the practical development of the project and is divided into three main parts: the UCM application, the software package, and cryptography.

The first part covers the implementation of the application, providing a detailed description of its components, structure, design, and integration. In the second part we delve into the creation of a software package. In this section the design and structure are explained, as well as the processes and tools used for its creation, and the encountered challenges. Lastly, the cryptography section offers a brief theoretical approach to the cryptographic techniques used in the project to follow up with its implementation to guarantee the security of software updates.

4.1 UCM Application

The Update and Configuration Manager demonstrative application is taken as starting point to create a functional application. This application already contains many of the necessary elements, however when trying to validate it, different errors are reported. The first step was to study and understand the model and the relationship between all the involved elements, and to complete the model to solve the different errors. Once it is possible to build the application, the code can be modified as needed.

Figure 15 illustrates the implemented solution for the UCM demonstrative application. It can be interesting to go back to Figure 8 and compare both architectures, as in this case both UCM Client and UCM Manager are found in the same machine. State Management is also implemented, as it is responsible of triggering the update. Persistency and Crypto library elements are also present, despite the fact that they are not fully functional in the current version, their key role and intended functionality can be observed on the application. As all components are found in the same ECU communications will be done using IPC protocol instead of SOME/IP.

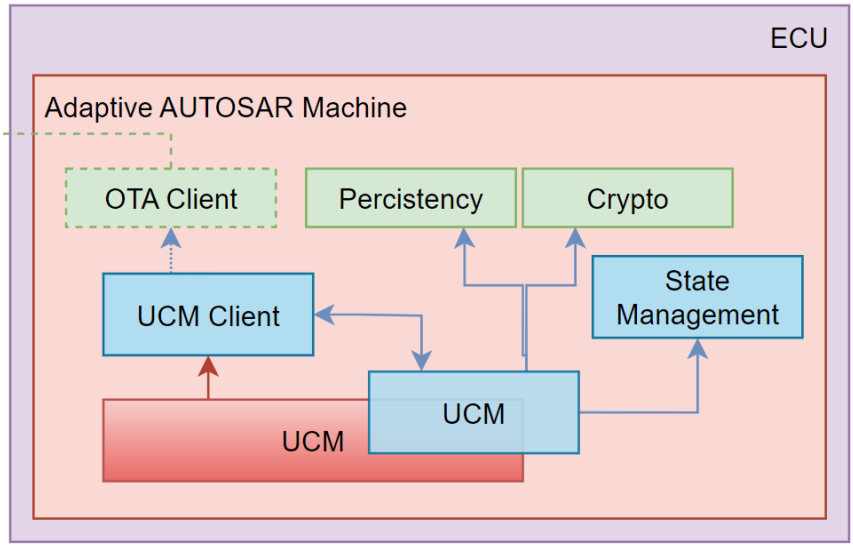

Figure 15: Implemented application and involved components.

Even though the development of the application is explained in a way that makes it seem lineal it is not, so it was necessary to go back to editing the model, building, and testing the application several times prior to completely understand and solve the different problems found during the development. A general overview of the different components is given as well as an insight of the found problems and the solutions.

4.1.1 Modeling of the UCM Application

On this section the most representative elements of the UCM application model are presented and explained. All of them are closely related, as when we talk about modeling, establishing the relationships between elements is an important part of it.

Executables & Processes

The UCM application contains three executables, each one related to one process:

Process	Executable
UpdateManagerProcess	amsr_updatemanager
PackageManagerClientProcess	amsr_updatemanager_client_example
StateManagerExampleProcess	amsr_statemanagement_example

The IDE allows to edit some of the elements of the model in a specialized editor, as it is the case for executables. Figure 16 shows the Executable Editor and its different fields, with update manager executable selected.

Figure 16: Executable editor showing the Update Manager model.

Executables are assigned a category and a software component type. The model of the process is more complex as it contains more elements to be defined. Figure 17 shows the model of the Update Manager process, whose executable has been previously seen.

Figure 17: Modeling of the Update Manager process.

All process needs to be linked to a Function Group, otherwise an error arises. It is necessary to modify the Package Manager Client Process for it to be linked to the Function Group. Once this is done, the process is claimed by the Function Group, as it can be seen Figure 19, meaning the problem is solved. Additionally, the Software Cluster needs to be modified for it to contain the Update Manager Process.

Later, when the UCM is completed and can be launched, it is possible to observe its process through the Linux System Monitor.

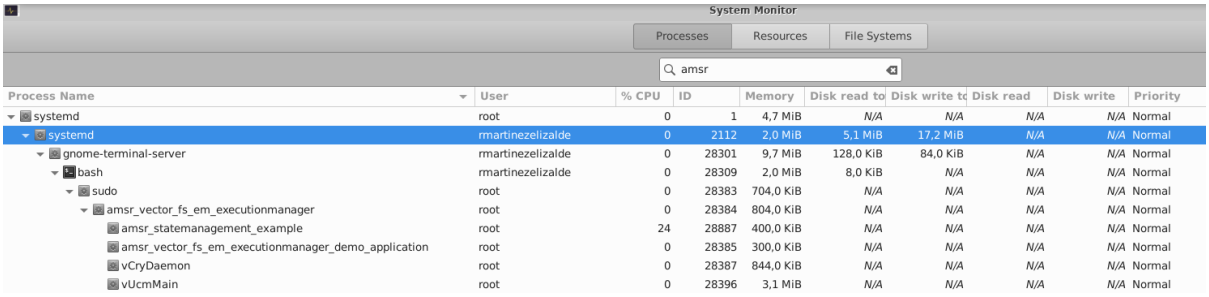

Figure 18: Linux System Monitor, showing the processes of the UCM.

Software Cluster & Function Groups

A Software Cluster groups all the software components that defines an application. It is a basic element all applications have, and it also represents an unloadable software package. It is possible to edit the Software Cluster through the model explorer, but it may be more convenient to use the "Software Cluster editor".

The Software Cluster has a Function Group Set that links the Software Cluster to a machine. A Function Group is defined for a machine, and it is possible for a Software Cluster to have multiple Function Groups. The UCM application has one Software Cluster and one claimed Function Group called *MachineFG*.

In the figure below the Software Cluster editor is shown. Different parameters such as the Target Machine or the Vendor id can be assigned completing the forms or in DML language. Below there is a table containing the processes that are related to the function groups claimed by the Software Cluster.

In the figure below the Software Cluster editor is shown. On the left side the Software Cluster and the Function group elements can be seen, they can be edited by clicking them and modifying the parameters on the right side of the screen. Different parameters such as the Target Machine or the Vendor id can be assigned completing the forms or in DML language. Below there is a table containing the processes that are related to the function groups claimed by the Software Cluster.

During the building process appears an error stating that the software cluster index is invalid. To solve this is necessary to define an Admin Data element with the name "SoftwareClusterIndex" and a value of zero.

Figure 19: Software Cluster editor

Machine

The UCM application has one machine, called "Adaptive Machine". The IDE offers a Machine Editor to easily edit the processors, processes, services, and logging for the machine.

Different elements are defined to model the machine:

- Machines: Defines the specifications of the machine, such as its cores.

- Machines Design: Defines the design of the machine.

- Mode Declaration Groups: Defines the different machine states and the initial state. In this case: verify, running, startup, shutdown, restart and off.

- Function Groups: Defines the function groups for a machine.

- Process to Machine Mapping Sets: Defines in which machine shall the processes run.

Figure 20: Machine editor.

Service Interface

Communication between UCM Client and UCM Manager is done though the service interface *Package Management*. In the model of the service interface are involved different modeling elements, as can be seen in the Figure 21. As well as the "Service Interface" elements, a provided and required service instances are modeled for each protocol, in this case IPC, and ports are mapped to them.

It is important to note that in the model the methods are not implemented, only defined. The implementation of the methods is done using C++ programing and is automatically generated according all the parameters.

All communications are service oriented, so the components ports are to be modeled. The client component, who offers a service, will have a required interface whereas the server, who subscribes to and uses a service, has a provided interface.

Figure 21: Service Interface elements

The Service Interfaces elements defines methods, fields, and events. The following image shows an example of the definition of a method, the transfer start method. It includes a description, declaration of input and output arguments and the related errors.

Figure 22: Example of a Method (Transfer Start)

Error domain

One of the elements that AUTOSAR specification defines are errors. Those errors shall be defined during the model and then used when programing the application, instead of being directly coded. Different methods may report the same error. Figure 23 shows the elements that models the application errors.

Figure 23: Application errors.

4.1.2 C++ code

The UCM Application, as all Adaptive applications, is programed in C++ language.

A portion of the C++ code in an AUTOSAR Adaptive application is automatically generated from the application model, as described in the previous chapter. The generated code can be found in the "src-gen" folders. In addition to the generated code, an Adaptive application relies on Basic Software, which refers to a standardized layer providing essential functionality and services. Typically supplied by a software provider, Basic Software is a significant component of the Software Integration Package. Lastly, the behavior and execution of the application are determined by the programming implementation itself.

The source code can be found in the folders "example" and "app". Images below show the directory and file structure, on the Visual Studio Code explorer.

The "example" folder contains the State Management and the UCM Client components. The State Management implements the update request service interface, creating the skeleton

and providing the service. On the Update Management Client folder is where most of the components are found.

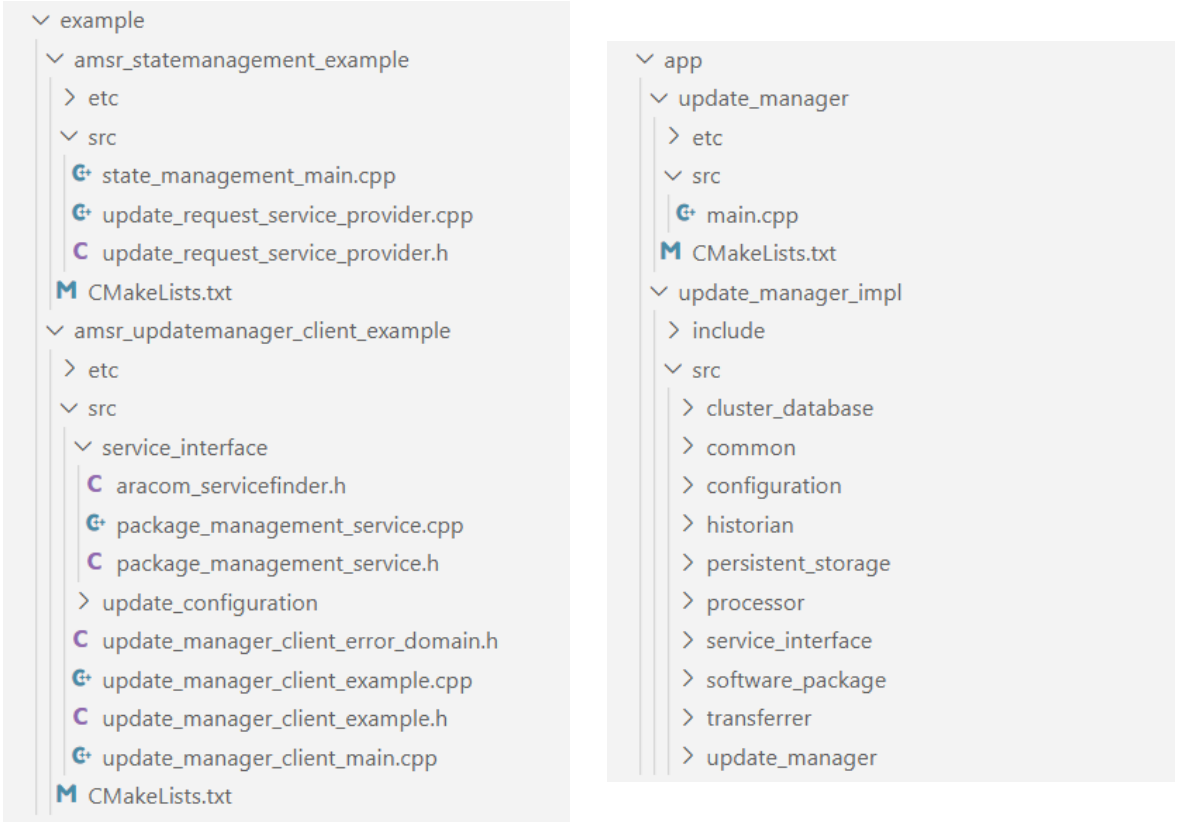

Figure 24: Directories and files of the UCM app ("example" folder)

The main files defining the UCM Client behavior are "update_manager_client_example.cpp" and "update_manager_client_main.cpp". The following code fragment shows the initialization of the UCM Client.

```
UCM Client (code fragment)
```

The update sequence method's is parsed from its corresponding configuration file, and sequentially called. By doing so, it is possible to verify the different method can be used properly.

```
Update sequence (code fragment)
```

For example, the implementation of the Transfer Start method is as can be seen in the following code fragment:

```
Transfer Start method (code fragment)
```

4.1.3 Building and executing the UCM Application

For building the UCM application the "Build Helper" is used. There are two different files that are directly involved in the building process and may need to be modified:

- Development.json: On this file the CMake module options for each basic software component are configured.

- app_lear_ucm.py: The components to be built with the application are defined in this python file, so it is necessary to check the dependencies before starting the build and adding the modules if necessary. Also, the name of the file shall match the name of the application, written in snake case.

When all the files have been modified the application can be build using Command 1 as it has been previously explained in chapter 2.

The execution manager, the software component that together with the operative system is responsible for starting and managing the execution of adaptive application, is not automatically generated when building the Update and Configuration Manager application. Although it would be possible for it to be automatically generated, the most time-efficient solution is to copy an already generated execution manager to the "opt" folder in the UCM application build directory.

4.1.4 Configuration files and directory structure

After the application has been properly modeled and programed, the next step is to generate and built it. The Build Helper tool facilitates this process and creates the directory structure, which includes the essential elements for executing the application. The build directory, whose name and location are chosen when using Command 1, contains separate folders for each of the Basic Software components, along with the generated code folder, "src gen", and the "install" folder. It is essential to understand the content of the "install" folder as it contains the elements necessary for the execution and functionality of the application.

The image below shows the structure of the directory. The "bin" folders always contain the applications binaries, while the "etc" directories contains the configuration files in JSON format. "sbin" contains the execution manager binary necessary to start the execution of the application, how to use it has been previously explained in section 2.

The components that are going to be executed within the UCM application are located in the "opt" directory. All of them will contain a "bin" and a "etc" folder, and eventually some additional directories for each specifical functions. When testing the application, it may be useful to exclude some components form the execution and it may be done deleting them from this directory. We can choose to delete the SOME/IP directory, when the communications are done exclusively using IPC protocol, and the Crypto Stack, when cryptography is not being used.

```
Install
├── bin
├── etc
├── include
├── lib
├── opt
│   ├ amsr_statemanagement_example
│   ├ amsr_updatemanager_client_example
│   │  ├ bin
│   │  └ etc
│   │    ├ […]
│   │    ├ update_configuration.json
│   │    └ […]
│   ├ amsr_em_executionmanager_demo_application
│   ├ amsr_sec_cryptostack_daemon
│   ├ amsr_updatemanager
│   │  ├ bin
│   │  ├ etc
│   │  │ ├ […]
│   │  │ ├ per_key_value_storage_config.json
│   │  │ ├ swcl_db.json
│   │  │ ├ update_manager_config.json
│   │  │ └ […]
│   │  ├ software_packages
│   │  └ var
│   │    └ key_value_storage1.json
│   └ someipd_posix
├── sbin
└── share
```

Figure 25: Directory structure

From those components found in the "opt" directory the most relevant to be explained are the "amsr_updatemanager_client_example" and the "amsr_updatemanager", as they are the UCM Client and the UCM, respectively. During the update process a software package is transmitted from the UCM Client to the UCM; in this current use case, both components are found on the same ECU.

The "amsr_updatemanager_client_example" contains only two folders, "bin" and "etc", as previously stated. In "etc" we encounter various configuration files, some of which may duplicate those found in the [...]/install/etc directory. Since these files are generated automatically, they should match or coincide with one another. By using JSON files to configure the application's execution, it is not necessary to compile the entire project for making adjustments.

One of the most significant configurations files for the UCM Client is the "update_configuration. JSON", as it defines the methods to be called and their order. This file is parsed, and the sequence followed during the update process.

```
Methods sequence (configuration file)
```

The other relevant configuration file for the UCM is the "update_cofiguration.JSON", its content can be seen below. This JSON defines essential information such as paths, limits for the transmissions and wait times.

```
Update cofiguration (configuration file)
```

For the "amsr_updatemanager" the directory structure is slightly different, because as well as the "bin" and "etc" folders we can find the "var" and the "software-packages".

The folder "var" is standard and can be found for other components when additional files are needed for the component to function. In this case the folder var contains a JSON file named "key_value_storage1.json", containing information regarding the storage of software packages, such as the transfer ID of the current software package.

In the "etc" folder different configuration files can be found, some of them matching the ones previously seen. In this case, the most relevant one is the "update_manager_config.json", which contains configuration parameters for the transmission and storage of software packages, as it can be seen below:

```
Update manager cofiguration (configuration file)
```

The folder "software-packages" shall match the one assigned to the parameter "root_path_to_software_packages" in the configuration. Software Packages are temporary stored in this location before they are installed. So, once the application has been executed at least once, it is possible to see different software packages with their manifest in the folder.

4.1.5 UCM Application outputs

UCM application provides information of its execution through the Linux terminal. Each software component is assigned an identifier and a logging level that may be "verbose", "info", "debug" or "error". A complete terminal output can be found in the annexes. In this section, only the most relevant parts of the execution will be commented upon, with some lines removed for the sake of clarity and easier understanding.

Once the UCM application is launched, the different components are found and UCM state switch to Startup state. The UCM Adaptive application, once it is running, successfully stablishes IPC connection.

```
[VUcm main][INFO] Initialization completed. Switch to Startup state.
[Demo EMDA][DEBUG] AdaptiveApplication::Initialize
[Demo vexe][DEBUG] Sending application state from PID:28385 with state: 0
[Demo EMDA][DEBUG] AdaptiveApplication::Run
[Demo vsmc][DEBUG] Establishing IPC connection to domain = 1000, port = 2000
[Demo vsmc][DEBUG] Establishing IPC connection succeeded.
```

Now that the connection is stablished offered services may be discovered, and proxy and skeleton is created for the "Package Management" service interface:

```
[XVUm XvUM][INFO] UpdateManagerClientExample::Initialize.
[VUcm vUcm][INFO] main:90: Initializing
[XVUm XvUM][INFO] Establishing connection to the Package Management service interface...
[VUcm vcom][INFO] [28396: UpdateAndConfigurationManagement/PackageManagement]
Skeleton:385: Skeleton created for Service 'PackageManagement'.
```

```
[XVUm vcom][INFO] [28397: ProxyServiceDiscovery]OfferService:231: OfferService for
instance identifier Ipc:1
[XVUm XvUM][INFO] Found Package Management service interface instance.
[XVUm XvUM][INFO] Connection to Package Management service interface is established.
```

Once all components have been initialized and communications successfully stablished the update process is stared. On the fragment below, the result of calling the different methods is seen, from the viewpoint of the UCM Master. Next method to be called is "ProcessSwPackage" which returns "Operation not Permitted".

```
[XVUm XvUM][INFO] UpdateManagerClientExample::Run.
[XVUm XvUM][INFO] TransferStart returned successfully.
[XVUm XvUM][INFO] TransferData returned successfully.
[XVUm XvUM][INFO] GetHistory returned successfully.
[XVUm XvUM][INFO] GetSwClusterInfo returned successfully.
[XVUm XvUM][INFO] GetSwClusterChangeInfo returned successfully.
[XVUm XvUM][INFO] GetId returned successfully.
[XVUm XvUM][INFO] GetSwClusterDescription returned successfully.
[XVUm XvUM][INFO] GetSwPackages returned successfully. Number of Software Packages
available: 5
[XVUm XvUM][ERROR] OperationNotPermitted
```

The application, now that the sequence cannot be completed, closes the connection and the process is terminated.

In summary, in the current status the UCM application establishes connection using IPC communication with all the necessary components, subscribing to, and using the offered services. Through the service interface "Package Management" the different methods are sequentially called, and successfully returned until "ProcessSwPackage" is called, which returns an error, "Operation not permitted".

The error is searched in the AUTOSAR Software Specifications; three different cases may arise that error for the "ProcessSwPackage" method, as can be seen on the fragment below. Even though this problem may be due to incoherence in the software package data it is also likely to be caused by incomplete development of the basic software or the UCM components. Due to time constraints the problem has not been further investigated.

The Software Integration Package that has been used is integrated for development and test purposes, so some of their content is not completed and its complete functionality validated. As well as that, it is uses AUTOSAR release R20-11, not the latest one, limiting some functionalities

4.2 Software Package

4.2.1 Introduction

Even though AUTOSAR defines the structure and requirements of Software Package, many aspects are left to be determined and implemented by the vendor. In this section the implemented solution will be more specifically approached, as the general description of a Software Package has already been seen in section 2.6.3

As developers testing the creation of a software package and the available tools, in this chapter the chosen decisions regarding the implementations, the difficulties found during said implementations, and finally the next steps that should be followed in order to improve and add more functionalities to the software package.

4.2.2 Format description

The software package contains a header, and a manifest, and a payload. All contained in a signed package. When creating the software package structure, it must be taken into consideration the UCM application and what does it expect to receive. As the implementation of the application is the developer's responsibility, so it is to guarantee that the software package is within the applications requirements.

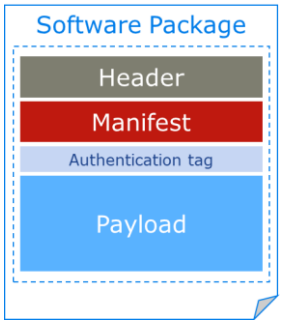

Figure 26: Software package format

It is possible for the software package to contain elements beyond the ones described in this section, however as the components being studied are the UCM Master and the UCM it is out of the scope to define them. As follow the vendor implementation of software package elements is described:

4.2.2.1 Header

The header contains information about the package in a binary format. It contains the software version, the manifest size, the payload size, and the signature algorithm ID. This information is provided when modeling the software cluster or, in the case of the manifest and payload size, acquired when creating the software package.

There are two versions of the header and the one used must be specified in the first byte of the header, being the acceptable values 1 or 2.

Table 1: SW Package Header version 1 parameters.

Table 2: SW Package Header version 2 parameters.

4.2.2.2 Manifest

The manifest is a set of data that describes the content of the software package and includes the model elements required for integration. The manifest is by AUTSOAR definition a modeling element in ARXML format. Yet, other formats can be used to represent this data, and in this case JSON format is used. So, the JSON file is a representation of the ARXML definition of the software cluster metamodel.

4.2.2.3 Authentication tag (Signature)

For the authentication tag, a signature is used. The signature is a binary file used to guarantee the software package's integrity and protect the vehicle. The propose and integration of this element will be explained in greater detail in section 4.3.

4.2.2.4 Payload

Because it is possible to implement a Software Package without a payload, to facilitate the creation of a first software package with testing and demonstrative purposes the payload has been omitted. Doing so, the process will not be affected. In the future a payload could be added as previously explained in chapter 2.6.3. Nevertheless, the implementation of a Software Package without a package is useful as it can be installed in order to add a Software Cluster to the database.

4.2.3 Steps to generate a Software Package

As in general terms AUTOSAR does not specify the exact content of the software package neither does specify how to create them. The solution to integrate a software package is responsibility of the developer. The software provider includes in the SIP different tools and information on how to create a software package. One of the aims during the development of the project has been understand how those tools can be helpful and fulfill our requirements, as well as determining its maturity and constraints.

Software Packages are created using a generator included in the build environment, following the previously specified vendor specific format. The steps to generate a Software Package are those enumerated below and will be described in greater detail in the next sections.

1. The software cluster is modeled and the manifest in JSON format generated.
2. The software package generator is used to create a file containing the manifest and the header.
3. This file containing the header and the manifest is signed, using a supported signature algorithm. The signature process of the software package is described in section 4.3.
4. The software package generator is used again, but with different parameters, to generate the final software package, containing the header, manifest, payload and signature file.

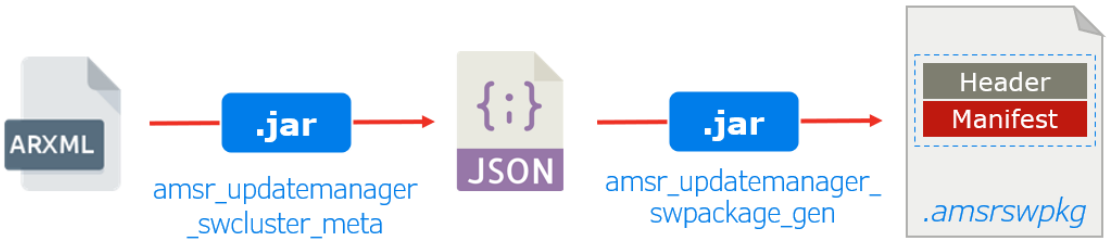

Figure 27: Steps to create a Software Package

As previously explained, a software package can contain information to perform an update to the application or to the platform and themselves. It can also be used to remove content from the platform. The contents of the Software Package created with demonstrative proposes will be the example application HelloWorld. So, the aim would be to install said application to an adaptive platform and its model modified to serve this propose.

4.2.3.1 Model of the Software Cluster

Considering that the starting point is a functional application, the element Software Cluster is already present in the model. However, the Software Cluster does not have all the necessary components to generate an uploadable Software Package.

The first step is to complete the model of the application for it to contain the elements describing the software package. The generator used in the second step requires some fields of the admin data type to be defined to functionate properly. The developer must define this data when modelling the software cluster, as it is parsed from the JSON manifest generated in this step. So, it is mandatory to define the following elements:

Element	Description
Name	Name of the Software Cluster.
Version	Version of the Software Cluster.
Files	Files in the Software Package, multiple files can be included.
Vendor	Vendor of the Software Cluster.
Description	Description of the Software Cluster.
Vin	Vehicle identification number.
Action	Update action: Install, Update or Remove.

Table 3: Mandatory meta-data elements of a Software Package.

In the Technical Reference of the Update and Configuration Manager an example manifest in format ARMXL is given:

Figure 28: Example of the manifest in ARXML format.

At this point the difficulty was how to model the content of the software cluster, it is not standardized by AUTOSAR, so Admin-Data elements are used instead. First, we tried to use the example manifest of the software provider in the technical reference of the UCM, but doing so we discovered it is not possible to modify the ARXML directly as it is protected against being manually written. So, when trying to introduce the values manually they would not be saved, it is necessary to create them one by one in the model explorer.

To model the manifest in the model explorer, we need to create Admin-Data elements in the Software Cluster we want to include in the Software Package. As it can be seen in Figure 29, the SoftwareCluster contains already the elements: Claimed Function Groups, Contained Processes and Admin-Data. By right-clicking on Admin-Data it is possible to create an Sdg element, this kind of element is the one used to model data beyond the standard elements.

Figure 29: Modeling an updatable Software Cluster

Once all the elements are defined the JSON file can be created by executing the generator *amsr_updatemanager_swcluster_meta.* In this case, the generator for creating the software cluster JSON file is integrated in the authoring tool, and automatically executed when generating a project. In Figure 30 the buttons for validating and generating a project are circled in red. It is recommended to validate the project before generating it. It is also possible to generate it using the Build Helper by building the application.

Figure 30: validation and generation icons.

The next figure shows the obtained JSON file containing all the elements previously modelled.

Figure 31: JSON file of a Software Cluster

4.2.3.2 Header and Manifest generation

Once the Software Cluster is modelled and in JSON format, the file containing both the header and the manifest can be generated. In this second step, another generator is used, the *amsr_updatemanager_swpackage_generator*. This same generator will be used again to create the final software package.

The version of this generator contained in the SIP was not functional, so after discarding the possibility of not being using it correctly the software provider was contacted and informed us that the issue would be checked. A second version of the generator was provided. Because the generator was downloaded, Linux does not recognize it as an executable, hence the need to use Command 4 to grant execution permissions prior to running it.

```
sudo chmod +x amsr_updatemanager_software_package.jar
```

Command 4: Linux execution permission

The second version of the generator did not work properly neither, but in this case, it could be solved by adding a file, *"Log4j.properties"*, to the directory where the generator is found. The logger properties file can be found in the following directory, where the original version of the generator was provided.

To use the generator, it has to be executed using Command 5. The parameter "-ohm" establishes that only the header and manifest will be generated. The other parameters determine the directories to be used and the signature type: The option "-d" provides the path to root directory of the software cluster data files, option "-m" provides the path to the metadata file and option "-o" the output directory. The manifest signing type is configured with option "-mt", and the signature algorithm to be used with option "-s".

```
java -jar amsr_updatemanager_swpackage_gen.jar -d tmp/gensw/ -m
  swcl_package_metadata.json -mt K_FIXED -o tmp/ohm -ohm -s 5
```

Command 5: Generating the header and manifest.

At this point it is important to remember that some of the elements of the software cluster model are mandatory (Table 3) for this generator to function correctly. This information is not provided when executing the generator, but after studying the java source code of the generator it was found.

The result when a header and manifest file is correctly generated is the one seen in Figure 32, and the format of the file will be ".amsrswpkg".

Figure 32: Terminal, software package generator

4.2.3.3 Signature of the software package

The Software Package must be authenticated for the UCM to accept it. To authenticate it a binary signature from the previously generated header and manifest file needs to be created. This is not done with the tools included in the SIP but with OpenSSL. In chapter 4.3 the basic theoretical concepts of asymmetric cryptography are seen, as well as that, the steps, and commands to generate a signature are explained through an example. So those steps have been followed again to generate the signature of the header and manifest. At this point we have two files, the header and manifest and the binary signature.

Figure 33: Header and manifest file and its signature.

4.2.3.4 Final Software Package generation

The last step is also performed with the generator *amsr_updatemanager_swpackage_generator* but configuring different parameters: the parameter "-ohm" used in the second step is now replaced with "-ms" and the signature file is also provided.

```
java -jar amsr_updatemanager_swpackage_gen.jar -d tmp/gensw/ -m
  swcl_package_metadata.json -mt K_FIXED -o tmp/gensw -s 5 -ms
tmp/gensw/app_example_hello_world_1.00.00.0000.amsrswpkg.signature
```

Command 6: Generating the final Software Package.

The generated file will have the same extension as the first file containing only the header and manifest, so changing the parameter "-o" to a different directory would be recommended to prevent the first file being overwritten.

In this last step the generator combines the signature binary, the header and manifest, and eventually a payload to create a final signed package.

4.3 Cryptography

To guarantee the safety of the vehicle the software package must be signed by the OEM. The OTA updates could be used to perform a malicious attack to the vehicle, using it as an access point and resulting in fatal consequences. The vehicle must check who the sender of the package is, as well as that it has not been intercepted and modified. Asymmetric cryptography is used to sign the software package and guarantee that only authenticated packages will be processed and installed.

In asymmetric cryptography, also known as public-key cryptography, there are two related keys: a public key and a private key. The purposes of these keys are different when it comes to encryption or signing. In encryption the sender of a message uses the public key to encrypt it, this message can only be decrypted by the private key. Only the intended receptor of the message owns the private key, guaranteeing the message will remain private, even if someone intercepts it.

The purpose of a signature is different: there is no need to keep the message secret, but to enable the recipient to verify who it comes from and that it has not been modified during transmission. Malicious actors could intercept messages and modify them or could generate their own. Signature provides a way to detect those unauthorized modifications, enabling the recipient to verify the authenticity and integrity of the data. As the use of asymmetric cryptography is different in this case, so it is the purpose of the keys. The sender uses the private key to sign the message, and the private by the receiver to verify the signature. A verified signature confirms that the message comes from the sender and has not been modified.

The algorithm to be used to sign the software package, as stated by the software provider, and so supported by the tools used to develop the UCM application, is RSASSA-PSS with SHA2-256.

Table 4: Signature algorithm identifications.

Hash functions and signatures are relevant concepts in cryptography. A hash function is a mathematical function that given an input of any size generates an output of a fixed length, this output is known as hash or digest. The output is typically a sequence of bits or bytes representing the unique digital fingerprint of the input data. The SHA2-256, considered to be

a secure and reliable hash function, generates a 256 bites output. It is a deterministic hash function, meaning that given the same input the output will always be the same. It is designed in a way that a slight change in the input produces a completely different hash value, a propriety known as avalanche effect, and it is unfeasible to obtain the same digest from to different inputs. Hash functions are one-way functions, they operate in a way that makes it almost impossible to reverse the process and obtain the original message from the hash.

RSASSA-PSS (RSA Signature Scheme with Appendix – Probabilistic Signature Scheme) is a digital signature algorithm based in the RSA public key cryptographic algorithm, it uses the RSA algorithm for key generation but has its own specific signature scheme. A signature algorithm uses a combination of cryptographic techniques such as hashing and encryption to create a unique digital signature. RSASSA-PSS is only used to generate signatures and cannot be used to encrypt data. RSASSA-PSS introduces randomness into the signature, and it is possible to obtain two different signatures from the same message, even when using the same key. This algorithm can only encrypt messages of smaller or equal size of the RSA key, meaning that, generally, the hash value of the message will be the one encrypted.

The procedure to sign a file is as follows:

1. Generate an RSA key pair.
2. Calculate a message digest of the file to sign using a hash function.
3. Sign the digest using the private key.

And the procedure to validate the signature:

0. The original file, the signature and the public key is provided.
1. Calculate the digest of the original file using the same hash function used to sign.
2. Decrypt the signature using the public key, obtaining the original digest.
3. Compare the calculated digest with the original digest, if they match, the signature is valid.

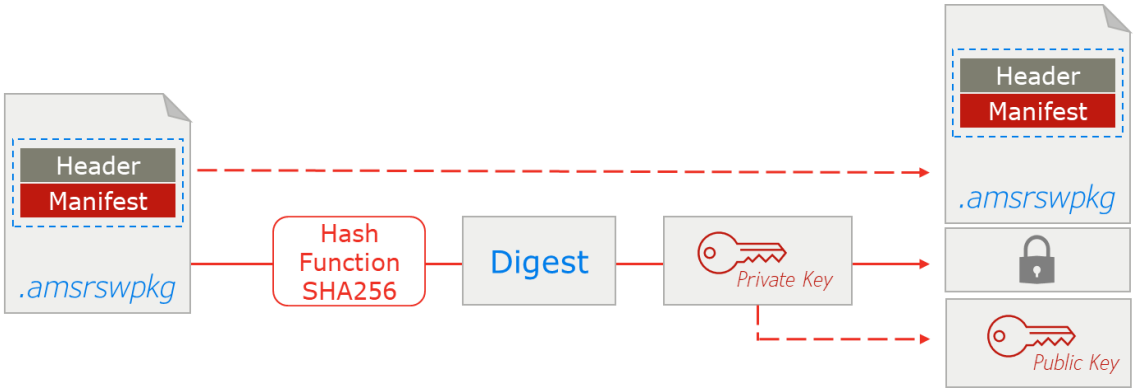

Figure 34: Software Package signature process.

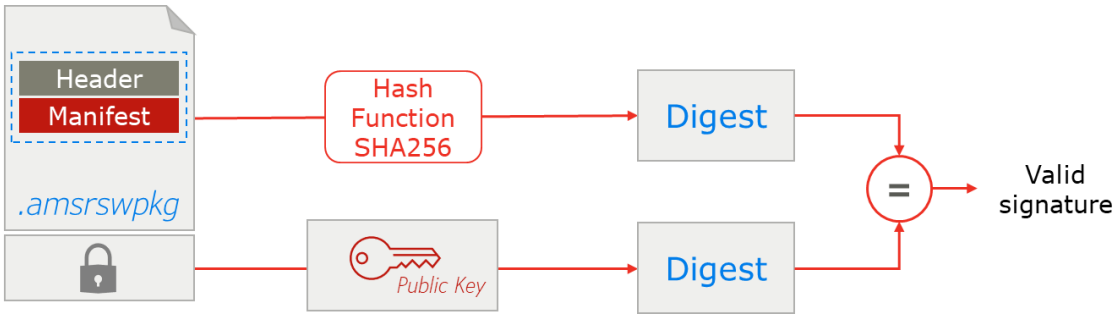

Figure 35: Software Package signature verification process.

4.3.1 Signature using OpenSSL

To generate the signature the software OpenSSL has been used. OpenSSL is an open-source library, available for different operating systems, including Linux, which provides tools and functionalities for encrypting, decrypting, working with digital signatures and certificates, keys, and performing other security related operations.

To understand and illustrate how the signature process works some tests have been done prior to signing the Software Package. The procedure to generate a signature does not depend on the file to sign, therefore, those commands and steps are also valid to sign a software package and have been used to do so.

We begin by creating the text file that will be used to perform the test, as it is easier to modify than a software package.

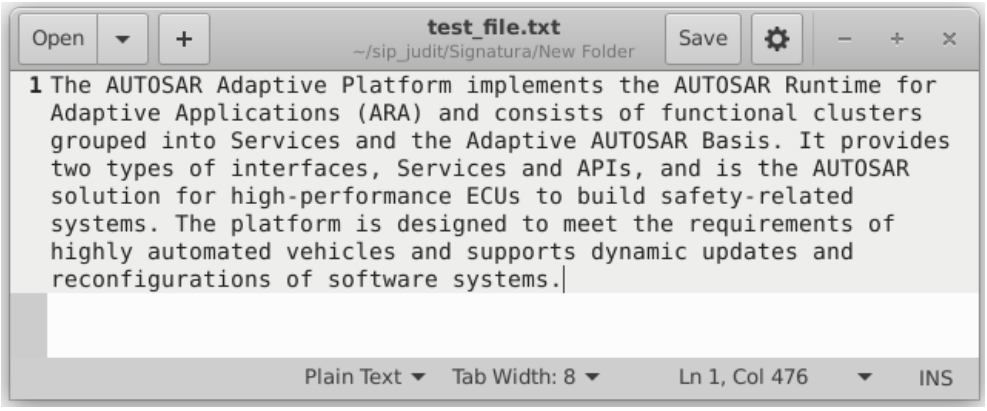

Figure 36: Test text file, containing AUTOSAR description.

The first step is to generate the private RSA key needed to sign the file. This is done using Command 7. Once a private key is available it is possible to extract the corresponding public key, using Command 8. The public key is not needed at the time of generating the signature and could be extracted later.

```
openssl genpkey -algorithm RSA -out private_key.pem
```

Command 7: Generate private key (OpenSSL)

```
openssl pkey -in private_key.pem -pubout -out public_key.pem
```

Command 8: Obtain public key (OpenSSL)

The second step is to generate the hash of the file to sign, this is done Command 9. Specifying the binary option is necessary as it is the required format in the next steps, without the results would be incorrect.

```
openssl dgst -sha256 -binary -out test_file_dgst.sha2 test_file.txt
```

Command 9: Digest (OpenSSL)

Lastly, the signature is generated according to the specified requirements using Command 9. In OpenSSL, the *pkeyutl* command is used to perform low-level public key operations and does support RSA algorithm. The operations and the parameters supported depends on the used algorithm, for the RSA algorithm, and more specifically the RSA-PSS algorithm, which is a restricted version of it, the sign and verify operations are supported and are the ones needed in this current use case.

```
openssl pkeyutl -sign -inkey private_key.pem -in digest.sha256 -out
signature.bin -pkeyopt digest:sha256 -pkeyopt rsa_padding_mode:pss -pkeyopt
rsa_pss_saltlen:-1
```

Command 10: Generate signature (OpenSSL)

As it can be seen on Figure 37, all the files needed are already present: the original file, the private and public keys, and the signature file. The signature file is the signed digest, and its size is 256 bytes.

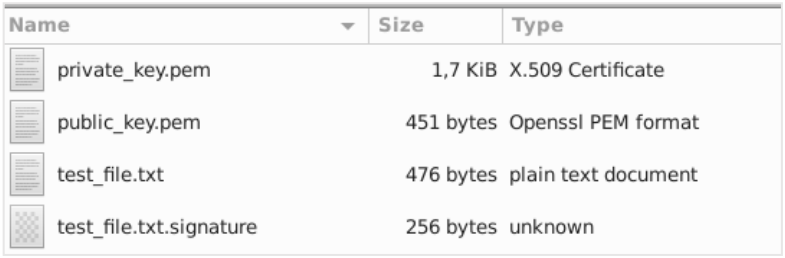

Figure 37: Files after signature process.

The verification of the signature would be performed by the message receptor; however, it is also possible to use OpenSSL to verify its authenticity and ensure it has been generated correctly, Command 11 shall be used. The result of this operation is going to be "Verified Ok" or "Verification Failure". As the file "test.txt" has not been modified the result is "Verified OK", as expected.

```
openssl dgst -sha256 -verify public_key.pem -signature test_file.txt.signature test_file.txt
```

Command 11: Verify signature (OpenSSL)

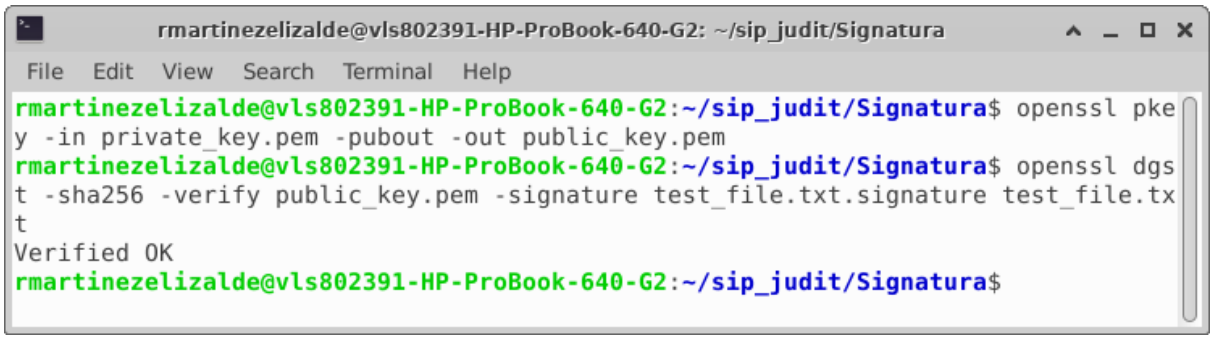

Figure 38: Signature verification OK, terminal output.

Since SHA2-256 should generate a completely different digest given a slight change in the input, we will modify the text file, by deleting the last character, to observe the impact on the result.

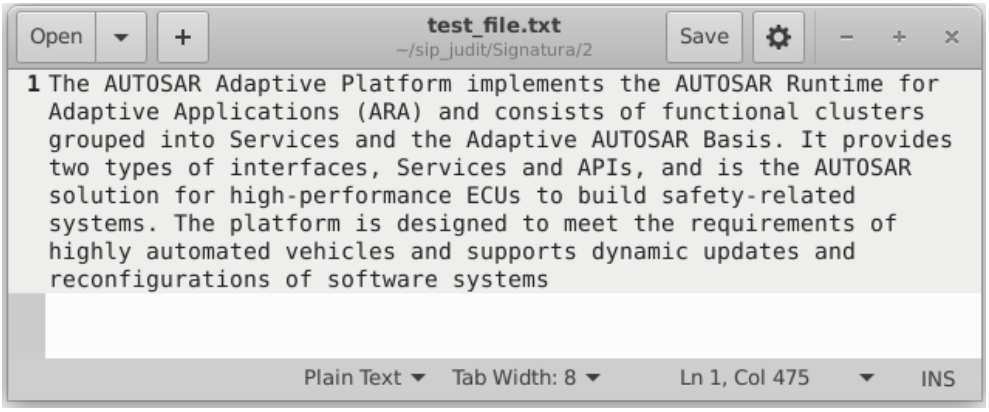

Figure 39: Test text file, containing AUTOSAR description, modified.

Using Command 10, the digest of the modified file is generated, so that it can be compared to the digest of the original test text file. In Table 5, we can see the digest of the original file and the digest of the modified file, represented in a sequence of 64 hexadecimal numbers, as expected, they do not match.

file	SHA2-256 hash value
original	6e726802120e6551c097452b9d9cb338dbea2b176ae073341dbeda724a01fdf4
modified	959227648a408a3a11a1bd68161516664306e5e4bbab87931faea9e2b461809f

Table 5: Hash values of the original and the modified test files.

The last step is to check that the modified file cannot be verified with the original signature. As it is not possible to obtain the private key from the public one neither the signature, a new signature cannot be generated. The result of trying to verify the modified file is "Verification Failure", indicating that it cannot be guaranteed that the file is authentic and therefore safe.

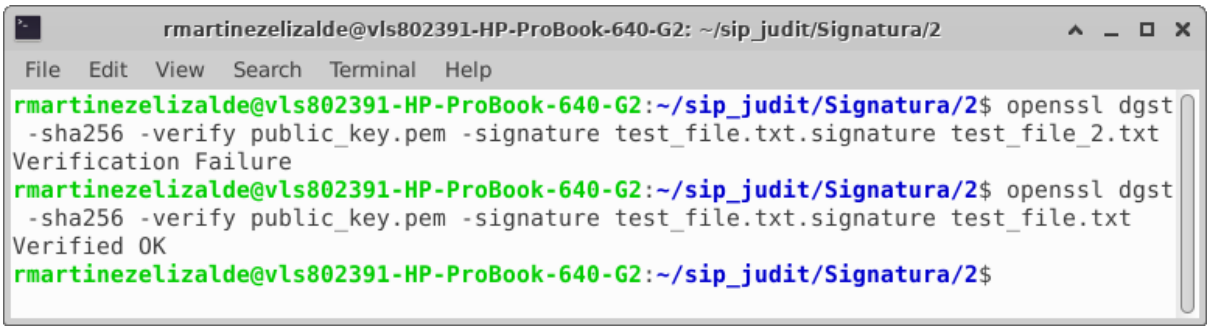

Figure 40: Failed verification of the modified file.

4.3.2 Providing the public key to the Adaptive Platform

The validation of the received software package is done in the ECU by the Cryptography Software Cluster, so the public key must be provided in a format that is easily accessible for it in the Adaptive platform. To do so, the software provider includes a tool in the SIP that can be used to convert a key in pem format to JSON notation.

This tool is generated simultaneously with the Cryptography module when building an application that includes it. However, the default configuration does not include the generation of the tool, and the option needs to be modified in the corresponding CMake.

Once the tool has been built, Command 12 is used to convert the key to a JSON format, which has to be completed with the additional information needed by the UCM application. It is important to be careful with the input parameters for the tool to work correctly, as it has some limitations and does not provide feedback information regarding any incorrect inputs.

```
java -jar keycon.jar -p keys/private_key.pem -a RSA -f pkcs8_unencrypted -b 256
```

Command 12: Key converter

This key is stored in what is known as "Slot" and will be used by the Cryptography module to verify the signature of the received software packages.

5. Conclusions

The objectives planed at the beginning of the project were to analyze the AUTOSAR Adaptive Standard with a specific focus on the functional cluster known as "Update and Configuration Manager," and to design and implement a demonstrative application based on this cluster. Furthermore, the project aimed to create a Software Package and incorporate the necessary cybersecurity mechanisms. For the demonstrative application, two use case were planed: the first is to establish communication using IPC and the second case using SOME/IP. Most of the objectives have been successfully accomplished, however the implementation of the UCM applications encountered challenges and constraints that prevented its fully development.

The first objective of analyzing the AUTOSAR Adaptive Standard has been fulfilled. This analysis has played a crucial role in the subsequent development of the application. In its turn, applying the concepts in a practical context has further solidified the comprehension of the standard.

The objective of creating a protected software package has also been completed. The created package follows the AUTOSAR Adaptive specifications and incorporates cybersecurity mechanisms to protect its integrity. To implement these cybersecurity mechanisms, an asymmetric cryptography signature and its corresponding key pair have been generated.

A demonstrative application for the Update and Configuration Management has been implemented, built, and executed. The application is able to stablish communication between components through Inter-Process Communication and to use the Package Manager service interface. However, not all the methods were called successfully, and the update process could not be completed. Due to the time constraints those issues were not solved.

With the accomplishment of those objectives, we have demonstrated the process of performing a software update on an Adaptive Platform, showcasing the ability to update and deploy vehicle software dynamically. OTA updates provide the opportunity to add features to vehicles post-purchase and address issues that may arise later, all without the need to bring the car to the workshop. Over-The-Air updates will be a standard vehicle feature in the near future, as they have already proven to be a reality. Nonetheless, while we have illustrated the implementation of this functionality and its feasibility, for a commercial deployment further development would be necessary. More complete and mature solutions and tools would be required to fully implement the update process and address all the additional considerations in a real-world use case.

From a personal standpoint, through this project I have gained expertise and enhanced understanding of automotive industry and its technologies. It offered me the opportunity to work on an innovative field and to grow professionally, improving both technical and project management skills, and gaining experience while seeing the impact and relevance AUTOSAR have in the automotive industry and its future tendencies.

6. References

[1] "AUTOSAR" [Online]. Available: https://www.autosar.org/. [June, 2023].

[2] AUTOSAR. "Layered Software Architecture". (September 2020). Available: https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf [Accessed: April, 2023].

[3] AUTOSAR. " Explanation of Adaptive Platform Design". (September 2020) Available: https://www.autosar.org/fileadmin/standards/R20-11/AP/AUTOSAR_EXP_PlatformDesign.pdf [Accessed: April, 2023].

[4] AUTOSAR. " Explanation of Adaptive Platform Software Architecture ". (September 2020). Available: https://www.autosar.org/fileadmin/standards/R20-11/AP/AUTOSAR_EXP_SWArchitecture.pdf. [Accessed: April, 2023].

[5] AUTOSAR. " Specification of Update and Configuration Management ". (September 2020). Available: https://www.autosar.org/fileadmin/standards/R20-11/AP/AUTOSAR_SWS_UpdateAndConfigManagement.pdf. [Accessed: April, 2023].

[6] AUTOSAR. " Requirements on Update and Configuration Management ". (September 2020). Available: https://www.autosar.org/fileadmin/standards/R20-11/AP/AUTOSAR_RS_UpdateAndConfigManagement.pdf. [Accessed: April, 2023].

[7] AUTOSAR. " Specification of Manifest ". (September 2020). Available: https://www.autosar.org/fileadmin/standards/R20-11/AP/AUTOSAR_TPS_ManifestSpecification.pdf. [Accessed: April, 2023].

[8] Tischer, M. " AUTOSAR Adaptive: The Computing Center in the Vehicle" (September 2018). Available: https://cdn.vector.com/cms/content/know-how/_technical-articles/AUTOSAR/AUTOSAR_Adaptive_ElektronikAutomotive_201809_PressArticle_EN.pdf [Accessed: April 25, 2023].

[9] OpenSSL. "OpenSSL: The Open Source Toolkit for SSL/TLS" [Online]. Available: https://www.openssl.org/. [Accessed: April 25, 2023].

[10] Ristic, I. "OpenSSL Cookbook," Feisty Duck Ltd. [Online]. Available: https://www.feistyduck.com/library/openssl-cookbook/online/. [Accessed: Apr. 25, 2023].

7. Annexes

7.1 Update and Configuration Management sequence diagrams

AUTOSAR Sequence diagrams for the update process, extracted from AUTOSAR Update and Configuration Management specifications.

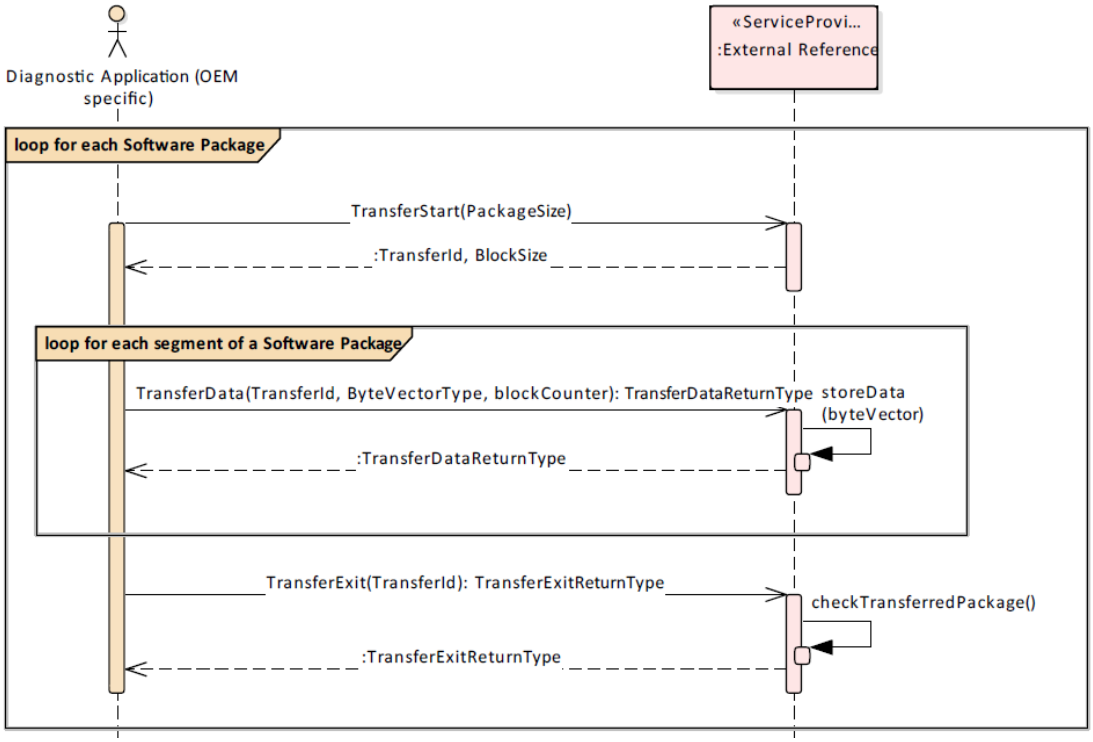

Figure 41: Data transmission sequence diagram.

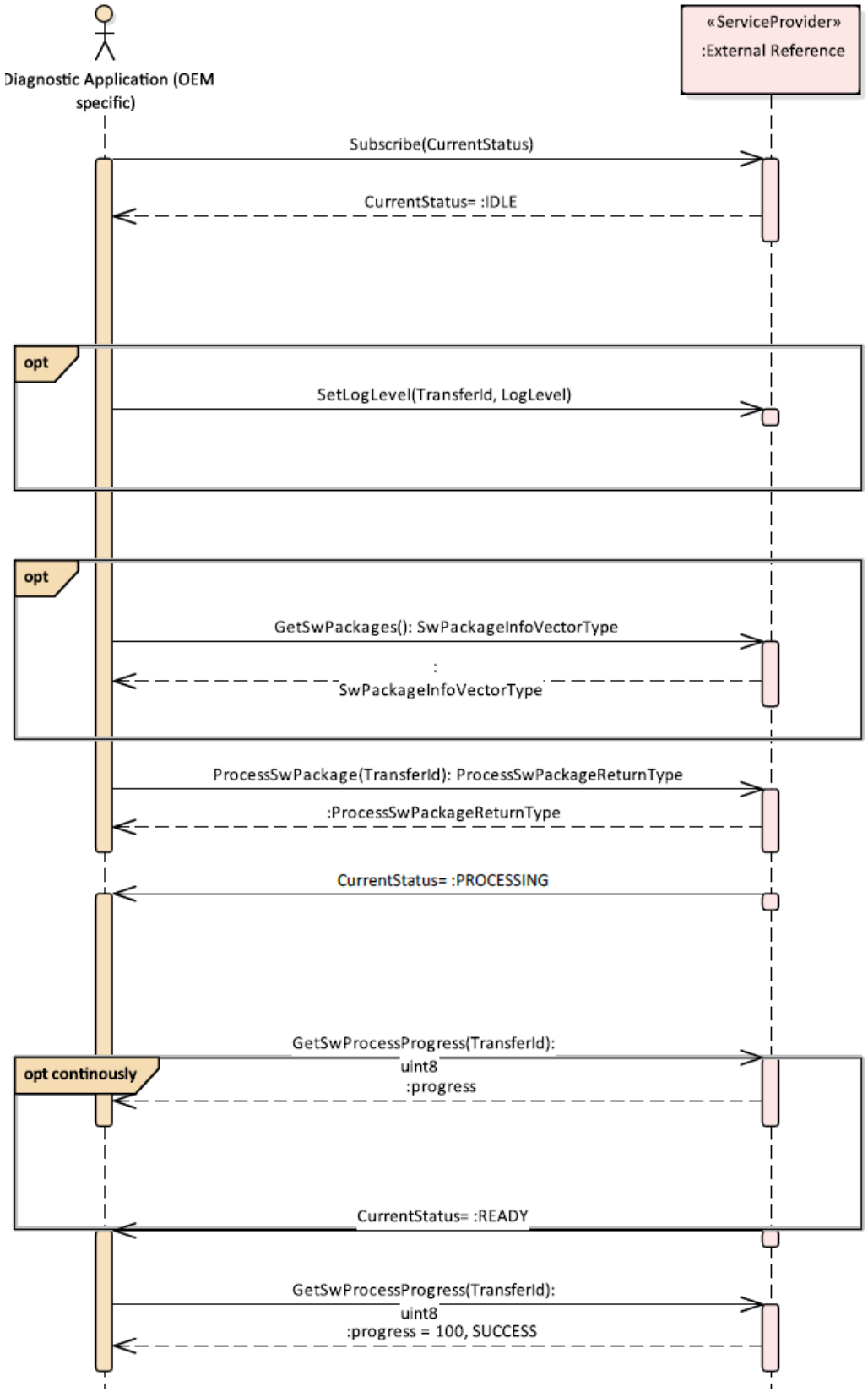

Figure 42: Package processing sequence diagram.

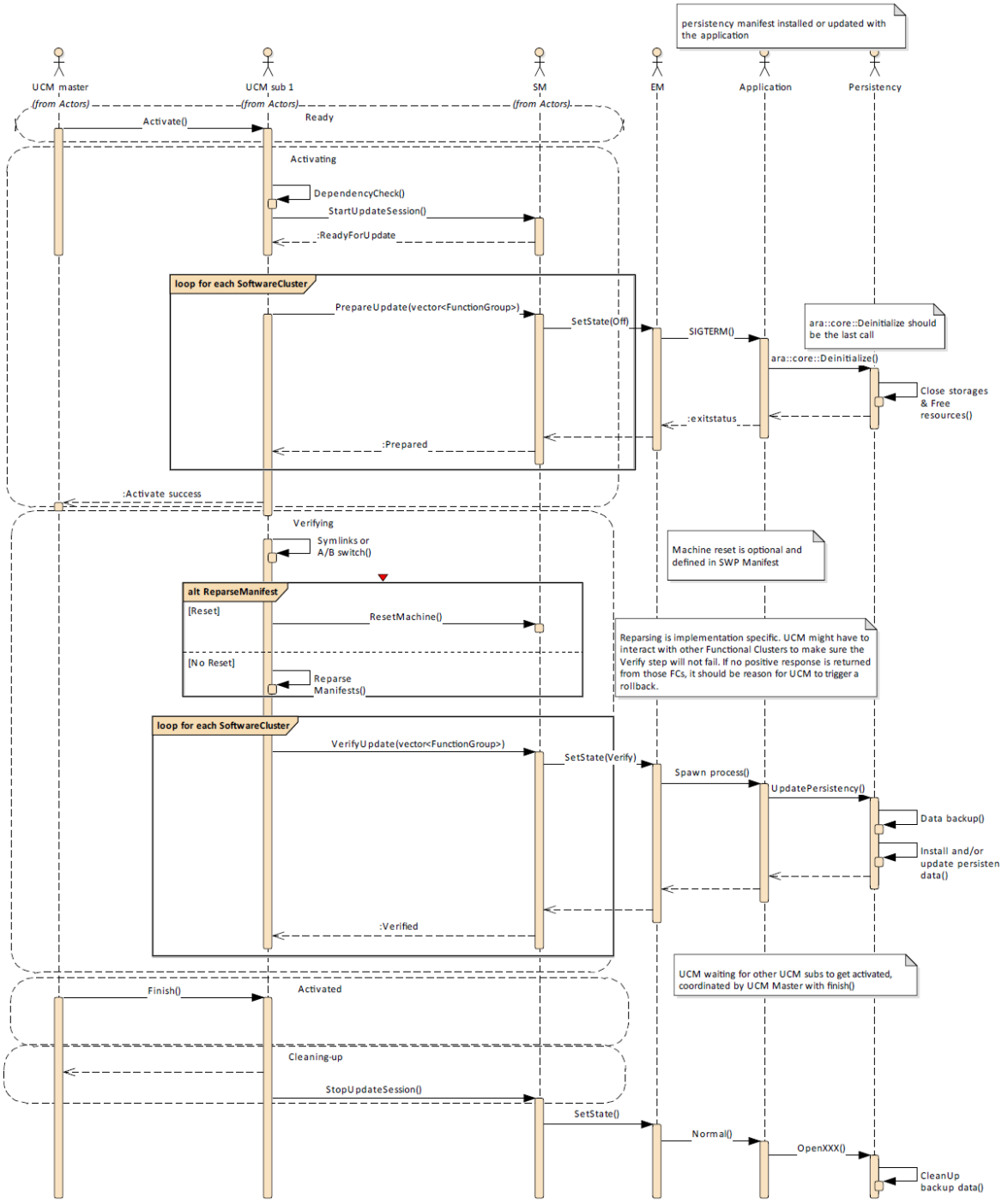

Figure 43: Activation process sequence diagram.

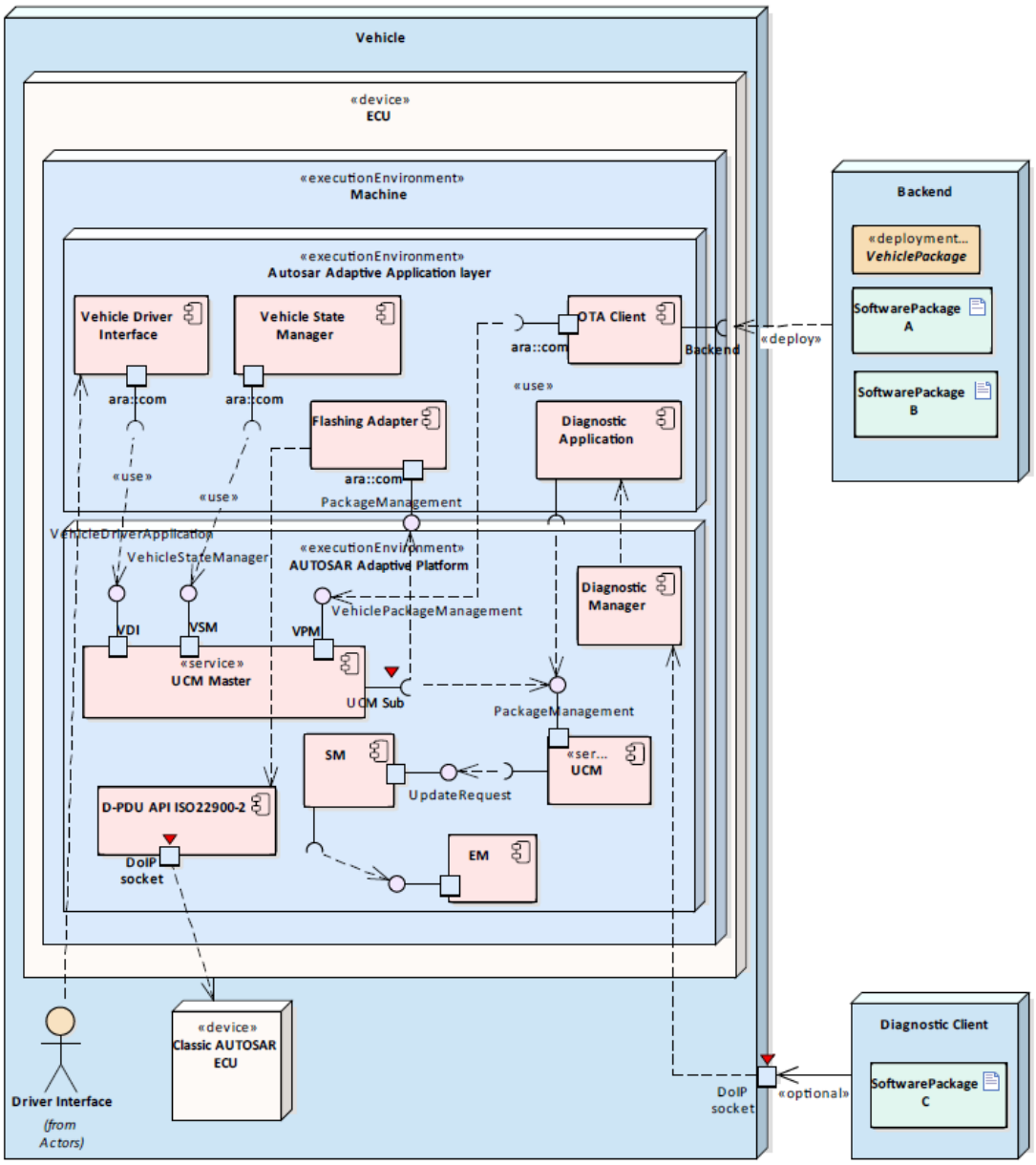

Figure 44: Vehicle Update Architecture

7.2 UCM Application terminal output

UCM Application terminal output