INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Abordagem de Anotações para o Suporte da Gestão Energética de Software em Modelos AMALTHEA

RICARDO MIGUEL MARTA BELO GARCIA GOMES
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

An Annotation Approach for the Support of Software Energy Management in AMALTHEA Models

Ricardo Miguel Marta Belo Garcia Gomes

A dissertation submitted in partial fulfilment of
the requirements for the degree of Master of Science,
Specialisation Area of Software Engineering

Supervisor: António Manuel de Sousa Barros
Co-Supervisor: Tiago Diogo Ribeiro de Carvalho

Porto, October 15, 2021

Dedicatory

To my father.

Abstract

The automotive industry is continuously introducing innovative software features to provide more efficient, safe, and comfortable solutions. Despite the several benefits to the consumer, the evolution of automotive software is also reflected in several challenges, presenting a growing complexity that hinders its development and integration. The adoption of standards and appropriate development methods becomes essential to meet the requirements of the industry. Furthermore, the expansion of automotive software systems is also driving a considerable growth in the number of electronic components installed in a vehicle, which has a significant impact on the electric energy consumption. Thus, the focus on non-functional energy requirements has become increasingly important.

This work presents a study focused on the evolution of automotive software considering the development standards, methodologies, as well as approaches for energy requirements management. We propose an automatic and self-contained approach for the support of energy properties management, adopting the model-based open-source framework AMALTHEA. From the analysis of execution or simulation traces, the energy consumption estimation is provided at a fine-grained level and annotated in AMALTHEA models. Thus, we enable the energy analysis and management of the system throughout the entire lifecycle. Additionally, this solution is in line with the AUTOSAR Adaptive standard, allowing the development of energy management strategies for automatic, dynamic, and adaptive systems.

Keywords: Automotive Software, Energy Analysis, Model-Based Software, AMALTHEA framework

Resumo

A indústria automotiva encontra-se constantemente a introduzir funcionalidades inovadoras através de software, para oferecer soluções mais eficientes, seguras e confortáveis. Apesar dos diversos benefícios para o consumidor, a evolução do software automóvel também se reflete em diversos desafios, apresentando uma crescente complexidade que dificulta o seu desenvolvimento e integração. Desta forma, a adoção de normas e metodologias adequadas para o seu desenvolvimento torna-se essencial para cumprir os requisitos do setor. Adicionalmente, esta expansão das funcionalidades suportadas por software é fonte de um aumento considerável do número de componentes eletrónicos instalados em automóveis. Consequentemente, existe um impacto significativo no consumo de energia elétrica dos sistemas automóveis, sendo cada vez mais relevante o foco nos requisitos não-funcionais deste domínio.

Este trabalho apresenta um estudo focado na evolução do software automotivo tendo em conta os padrões e metodologias de desenvolvimento desta área, bem como abordagens para a gestão de requisitos de energia. Através da adoção da ferramenta AMALTHEA, uma plataforma *open-source* de desenvolvimento baseado em modelos, é proposta uma abordagem automática e independente para a análise de propriedades energéticas. A partir da análise de traços de execução ou de simulação, é produzida uma estimativa pormenorizada do consumo de energia, sendo esta anotada em modelos AMALTHEA. Desta forma, torna-se possível a análise e gestão energética ao longo de todo o ciclo de vida do sistema. Salienta-se que a solução se encontra alinhada com a norma AUTOSAR Adaptive, permitindo o desenvolvimento de estratégias para a gestão energética de sistemas automáticos, dinâmicos e adaptativos.

Palavras-Chave: Software Automotivo, Analise Energética, Software baseado em modelos, plataforma AMALTHEA

Acknowledgements

First, I would like to thank my family, girlfriend, and closest friends for continuously supporting me during my academic journey.

Additionally, I must express my gratitude to Harald Mackamul, Lukas Krawczyk, Raphael Weber, and Benjamin Beichler from the Panorama Project, for the support offered throughout this project.

Finally, I am thankful to professor Luís Miguel Pinho and Luís Miguel Nogueira for giving me the opportunity to participate in this project, and to my supervisor, professor António Barros, and my co-supervisor, professor Tiago Carvalho, for all the guidance and help provided.

Index

List of Figures

List of Tables

List of Code

Acronyms and Symbols

List of Acronyms

ADAS	Advanced Driver Assistance System
AHP	Analytic Hierarchy Process
AMQP	Advanced Message Queue Protocol
API	Application Programming Interface
ATDB	AMALTHEA Trace Database
AUTOSAR	Automotive Open System Architecture
BTF	Best Trace Format
CPU	Central Processing Unit
DPF	Direct Proportional Formulas
DSRM	Design Science Research Methodology
ECU	Electronic Control Unit
EMF	Eclipse Modelling Framework
ISO	International Organization for Standardization
LR	Linear Regression
MB	Model-Based
OEM	Original Equipment Manufacturer
PSS	Proportional Set Size
RBF	Radial Basis Function
SOA	Service Oriented Architecture
SVM	Support Vector Machines
SVR	Support Vector Regression
W	Watts

1 Introduction

The automotive Industry is well known for its economic and social relevance, providing products that are used daily by millions of people as their main means of transportation. Automotive systems can be compared with other products, such as personal computers or smartphones, in terms of belonging to a global industry. However, cars are considered safety-critical systems, as a failure in a vehicle is associated with a significant risk of severe injury or loss of life. Additionally, these products tend to be considerably expensive, with costs ranging from tens to hundreds of thousands of euros.

The automotive industry is constantly evolving, either on the manufacturing processes and on the quality of the products. The amount of software used in modern cars is growing dramatically, with numerous computing systems executing millions of lines of code (Doughty-White & Quick, 2015). Several challenges have been raised, related to the requirements and goals of the automotive industry, demanding complex solutions and a proper quality maintenance (Guissouma, et al., 2018). As a result, automotive systems are becoming a hugely focused research field of software engineering, with efforts to overcome the industry's challenges through the specification of new technologies, methodologies, and standards.

1.1 Context

In its inception, the automotive industry focused on producing mainly mechanical systems. However, due to the technological evolution, as shown in Figure 1, cars evolved over time into multidisciplinary systems, with progressively more and more software features (Haghighatkhah, et al., 2017).

Around the 1970s, software was introduced in automotive systems to perform minimal functions. Since then, the amount of electronics and computing systems has been dramatically increasing. In fact, it has recently taken even greater importance in cars (Broy, 2006), composed of a massive amount of software when compared to several other vehicular systems (Doughty-White & Quick, 2015).

Automotive computing systems are currently evolving into systems presenting cyber-physical properties, with embedded software components, physical components based on sensors (Lee, 2007), many computing units, and internal networks. This comes from the development and inclusion of several new features (Dajsuren & Van den Brand, 2019), such as the Advanced Driver Assistance Systems (ADAS), infotainment components and connectivity with external entities.

Consequently, with the need of providing high-quality and competitive products, more and more software capabilities are added to these systems. Demanding challenges are raised, to

comply with regulation and quality standards imposed by the industry, implying a great focus on the software's non-functional properties (Haghighatkhah, et al., 2017). One example is the ISO 26262 standard, intended for functional safety of road vehicles (ISO, 2011).

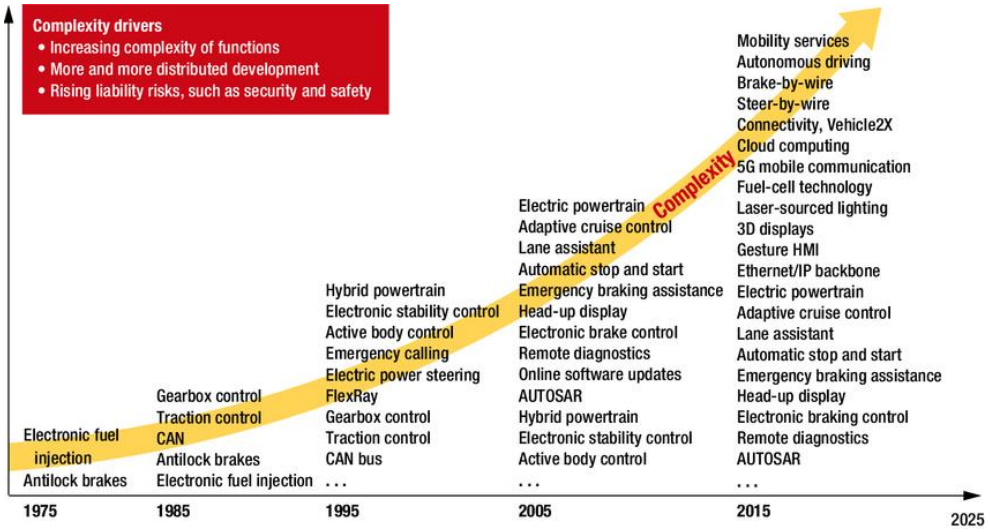

Figure 1 Increase of Automotive Software features and complexity, from (Ebert & Favaro, 2017)

Typically, an automotive software project is developed by several suppliers in a decentralized way. In the end, the Original Equipment Manufacturer (OEM) integrates the components (Dajsuren & Van den Brand, 2019), which communicate with each other through specific networks and protocols. This distribution of labour provides cost and risk distribution (Pretschner, et al., 2007). However, the integration of the products became more and more challenging due to the ever-increasing software complexity and to the dependencies between components.

The efforts to mitigate the rising difficulties focused on the fields of system specification, development lifecycle and complexity management, which became popular research subjects. Model-Based Engineering (MBE) approaches have been shown to improve the development of complex systems, with a greater traceability level, supporting the entire lifecycle of the project (Ambrosio & Soremekun, 2017), and allowing automatic code generation (Schmidt, 2007).

In this context, the European project PANORAMA – *Boosting Design Efficiency for Heterogeneous Systems* – has the goal "to research Model-Based methods and tools to master development of heterogeneous embedded hardware/software systems in collaboration with diverse and heterogeneous parties by providing best practice, novel analysis approaches, and guidance for development" (Panorama, 2021) with a focus on the automotive industry. For such purpose, this project aims to extend current approaches, exploring already existing open-source tools, developed to operate within heterogeneous environments, namely AMALTHEA/APP4MC (Eclipse APP4MC, 2021).

This work was carried under the scope of the PANORAMA project, in partnership with Robert Bosch GmbH, the Dortmund University of Applied Sciences and Arts, the University of Rostock, Vector Informatik GmbH, and the Eclipse Foundation Europe, among others.

1.2 Problem Statement

With the evolution of automotive products toward software-intensive systems, challenges are rising due to their increasing complexity, not only related to the functionalities of the vehicle, but also the non-functional requirements, such as real-time response, safety, and energy efficiency.

The adoption of Model-Based (MB) approaches allows the system execution, simulation, and specification analysis, supporting continuous optimization and testing from the earlier stages of the project throughout its entire life cycle. Additionally, it also provides adaptive capabilities during execution time, due to an increased automation level. Consequently, efforts have been made to provide these capabilities, with solutions such as the AMALTHEA/APP4MC framework (Eclipse APP4MC, 2021).

There is the need of creating model property annotations with information from the system's hardware and the executing software. This information can be used to offer analysis that supports and suggests changes to the system, ensuring the desired execution environment, and the realization of the imposed requirements.

Such analysis is also mandatory considering the energy domain, to increase efficiency, which is becoming more significant with the growing electrification and software usage of automotive systems. However, AMALTHEA/APP4MC traceability solutions are mostly centred on the software's timing behaviour.

In this way, this project focuses on studying and providing an information annotation approach related to non-functional energy requirements, supported by the AMALTHEA platform. In addition, the solution must consider automotive software development standards and methodologies suitable with dynamic and adaptive environments.

1.3 Objectives and Contributions

The main objective of the project consists of providing an energy properties annotation approach through Model-Based Engineering methodologies for dynamic automotive software applications, using the AMALTHEA/APP4MC open-source toolchain.

To specify a well-defined structure to this work, easing the achievement of its main goal, a logical decomposition in a set of smaller and isolated objectives was defined:

- Study and comprehension of automotive software systems and development methodologies.
- Study of software energy metric extraction mechanisms and management techniques applied to the automotive software domain.
- Development of an energy metric extraction mechanism relying on data that can be obtained through the analyses of AMALTHEA simulation and execution traces.
- Design and Implementation of a solution applied to the AMALTHEA framework for the analysis of execution or simulation traces, and the annotation of energy properties in AMALTHEA system models obtained through the adopted metric extraction mechanism.

The fulfilment of the partial goals described here determines the accomplishment of the main objective of the project. As a result, its main contribution consists of the definition of an approach, using the AMALTHEA/APP4MC open-source framework, capable of overtaking the challenges raised by the addressed problem.

Figure 2 describes the features offered by our solution.

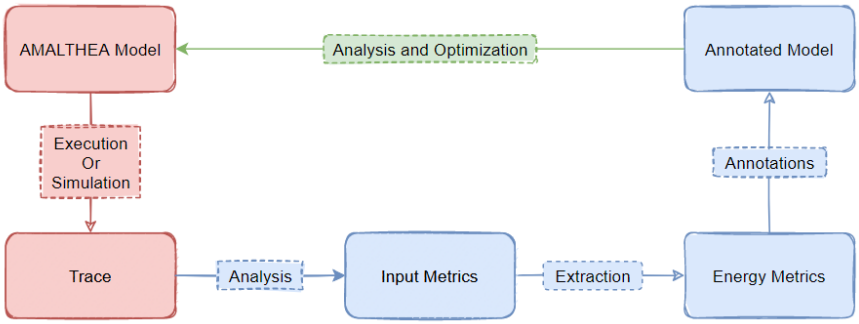

Figure 2 Flow Chart of the Solution

Features represented in red are already supported by APP4MC. This framework provides the mechanisms to specify AMALTHEA models and traces. Features represented in blue are provided by our approach. It performs the analysis of system traces and computes the energy properties of the system with the metrics obtained from the analysis, annotating them back in the model. As a result, it supports further energetic analysis and optimization using the AMALTHEA platform, represented in green on the figure.

1.4 Approach

To successfully carry out the project, the approach was established regarding the defined objectives, which already propose a logical structure to provide a solution to the problem.

In the first place, to increase the knowledge related to the automotive software domain, a detailed state of the art study and comprehension was performed considering the complexity

and evolution of automotive software systems. Finally, the most relevant challenges, development methodologies and standards were also undertaken.

On the other hand, an analysis regarding non-functional energy requirements was conducted. Energy management approaches in the automotive software industry were addressed, followed by energy metrics estimation and extraction techniques.

Afterwards, an analysis of the AMALTHEA/APP4MC toolchain was conducted. The framework's relevant aspects, such as the capabilities and development methodologies, were deepened. Thus, it became possible to analyse the solution, specifying the domain of the problem, the desired features, and allowing the requirements gathering process.

Subsequently, the solution design was performed according to the requirements and constraints collected during the previous stages of the project. On the one hand, it considers the analysis of traces, system models, and the annotation of energy properties, using the APP4MC platform. On the other hand, energy estimation mechanisms were also specified to support the solution.

Accordingly, the implementation phase was carried out next in an iterative way. Functionalities were developed and validated cyclically, allowing the solution's continuous refinement.

The development of the energy estimation mechanism was performed and integrated in the solution, along with the implementation of methods to analyse and obtain data from AMALTHEA traces and models. In addition, an annotation mechanism was specified to include the obtained energy information in the models under analysis.

Finally, after the implementation was concluded, the final product was evaluated, assessing the success of the project through the specification of quality indicators and suitable evaluation methodologies, covering all of its relevant aspects.

1.5 Work Methodology

To conduct the project, the Design Science Research Methodology (DSRM) was adopted, considering the nature of the problem, and the defined objectives. DSRM consists of a systematic research methodology suitable with technological projects (Freitas Junior, et al., 2017). It aims to provide the development of a product that accomplishes a specific goal, usually through concepts, models, methods, and instantiations (March & Smith, 1995).

In a similar way to the approach adopted in this project, a Design Science Research is conducted through an iterative process. Although this methodology defines an ordered structure, each stage of the research can be conducted within the order that suits best the researcher's needs (Peffers, et al., 2007). The DSRM is composed of 6 different stages, as illustrated in Figure 3.

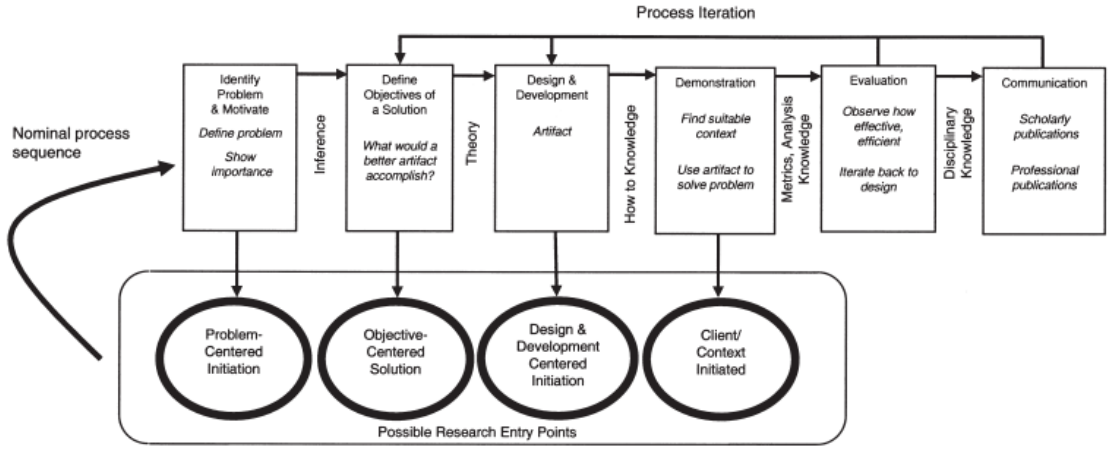

Figure 3 DSRM Model, from (Peffers, et al., 2007)

Problem Identification and Motivation:

When a DSRM is conducted, it aims to provide a solution to a problem. Consequently, in the first place, the problem has to be defined and contextualized, along with the motivation to conduct the research and the value of the solution (Peffers, et al., 2007).

During this project, the **Problem Identification and Motivation** stage is described in the Introduction, State of the Art and the Value Analysis chapters.

Objective Definition:

After specifying the problem, the approach adopted to conduct the research must be defined, along with its objectives. The objectives of a research may be measured in a quantitative or qualitative manner, and they must consider the context of the research and related works (Peffers, et al., 2007).

In this document, the **Objective Definition** is specified in the Introduction.

Design and Development:

The **Design and Development** stage consists of structuring the artifacts that solve the addressed problem. It includes the definition of the features that must be offered by the solution, its design, and the implementation (Peffers, et al., 2007).

This stage of the methodology is described in the Solution Analysis, Architecture and Design, and the Experimental Implementation chapters.

Demonstration:

With the development of the solution provided, it is important to demonstrate the application of the obtained artefacts. This stage is usually conducted through procedures such as experimentations, simulation, or case studies, among others (Peffers, et al., 2007).

In this project, the **Demonstration** of the solution is provided through its application to the simulation of an automotive use case, which is described in the Experimentation and Evaluation chapter.

Evaluation:

According to DSRM, during the **Evaluation** process, the behaviour and attributes of the solution have to be verified. Quality indicators and evaluation procedures can be defined in order to assess the ability of the solution to accomplish the objectives and requirements of the project in the expected way, solving the problem (Peffers, et al., 2007).

In this document, the **Evaluation** stage of the project is described in the Experimentation and Evaluation chapter.

Communication:

After conducting the research, the developed artifacts must be disseminated, considering the problem at hands, the context of the project, the obtained results, and the contributions (Peffers, et al., 2007).

For this project, the **Communication** process is conducted through the elaboration of this document.

1.6 Document Structure

This report is structured in the following manner:

Chapter 2, presents the State of the Art, composed of a background on the main concepts addressed during the project, along with the most relevant approaches and standards in the automotive industry. Additionally, a description of the adopted tool and a critical analysis of alternatives is provided.

Chapter 3, presents the Value Analysis, where the opportunities are identified and analysed in detail, along with the partial evaluation of alternative approaches to the problem. Finally, considering the target-audience of the project, the Value Proposition is described, and the solution's functions are examined, with the aim of increasing the value to the consumer, while reducing associated costs.

Chapter 4, presents the Solution Analysis. The analysis of the problem is provided considering the domain of the project, its major components, the definition of use cases, and the requirements specification, through the FURPS+ model.

Chapter 5, presents the Design of the approach, where the adopted and alternative architectures are analysed, along with the description of the solution's design, considering different perspectives according to the 4+1 view model.

Chapter 6, presents the Implementation stage. Here, the environment, adopted conventions and major concepts related to the development of the solution are addressed. Finally, the implementation of each major feature of the approach is also explained, along with the limitations that were experienced.

Chapter 7, presents the Experimentation and Evaluation, where the validation of the project is performed. The research hypothesis is formulated, along with information indicators, sources and methodologies used to perform the evaluation. Finally, the conducted experiments and the obtained results are discussed.

Chapter 8, presents the Conclusions. A brief summary of the problem and concepts addressed during the project is provided, along with the description of the proposed solution. The objectives fulfilment is assessed, followed by the overall results of the approach. In the end, limitations and future work are discussed.

2 State of the Art

The automotive industry holds a significant social dimension, delivering products that are used by millions of people as the main mean of transportation. Consequently, manufacturers are constantly making efforts to provide high-quality and competitive solutions.

Software tends to be globally used in numerous products, and cars are no exception. Initial approaches to the adoption of software features in the automotive industry were conducted around the 1970s. Such features consisted only of basic components with minimum communication, using Electronic Control Units (ECU) to support their functionalities, along with sensors and actuators to provide feedback to physical events (Broy, 2006).

Meanwhile, in the 1990s, automotive software continued to expand. At this point, the integration between inner electronic components and even some external entities was already required (Staron, 2017), with the introduction of infotainment and safety functions, such as the GPS and the Adaptive Cruise Control.

Lately, most of the innovations in automotive systems are related to software. In the current century, features such as the ADAS were introduced (Staron, 2017), allowing the car to perform complex and autonomous decisions. Nowadays, the automotive industry offers products presenting up to hundreds of millions of lines of code (Doughty-White & Quick, 2015). Subsequently, automotive systems became a quite focused software engineering research field, aiming to provide solutions for the several challenges and ambitious goals of the industry.

This chapter describes the state of the art of the project. First, it provides a background in automotive software systems, focusing on its main aspects, along with the AUTOSAR standard. Afterwards, Model-Based automotive software engineering is addressed, followed by the importance of the energy domain in the industry, and energy requirements management and metrics extraction approaches. Finally, the AMALTHEA/APP4MC framework is deepened and critically analysed, considering related works and technologies.

2.1 Automotive Software Challenges and AUTOSAR Classic

With the demand for the inclusion of more and more software components in automotive systems, one of the greatest concerns is the increase of complexity (Antinyan, 2020). Currently, automotive software implies the use of numerous ECUs and sensors communicating with each other, representing most of the development cost of a car (Broy, 2006).

Additionally, the automotive software industry can be described by five different aspects (Pretschner, et al., 2007):

1. **Heterogeneity of Software:** The components of an automotive system are referred to various domains and utilities, presenting different functions built by multidisciplinary teams.

   Consequently, automotive software is usually associated with a vast number of non-functional requirements. For example, some requirements may be related with reliability and safety, referring to the control of mechanical components. However, the car may as well be connected to external devices. Thus, security is also important, along with the need of adopting technologies with the ability to perform the desired communications.
2. **Distribution of Labour:** The development of a car consists of several isolated parts produced by different suppliers, while the OEM is responsible for other stages, such as design and assembly. As a result, the produced components are modular and reusable, decreasing risks and production costs.

   Now, the OEM is also responsible for the integration of software, which is becoming increasingly challenging, not just because it is made by various suppliers, but also due to the increasing dependencies and complexity of the system's components.
3. **Distribution of Software:** Modern cars are composed of a considerable amount of software and hundreds of hardware devices, in a certain way, due to the distribution of labour.

   Therefore, automotive systems are distributed. One function may operate over different ECUs, combining different domains and components.
4. **Variants and Configurations:** Another major aspect is the fact that cars may present different specifications based on the customers, their cultures, and tastes. So, automotive software must provide a high customization level.

   In addition, the extensibility of software is necessary, due to the considerable lifecycle of the vehicle. It must allow modifications to update the product with new features and improvements, which becomes a challenging task considering the complexity of the products, and the fact that numerous components are outsourced.
5. **Unit-Based Cost:** Automotive products are mass produced. There is a huge concern not only with the quality and number of products sold, but also with the production costs.

   A slight unit cost reduction may become relevant considering the amount of manufactured unities. However, a highly optimized product may present an increased complexity, compromising its reusability.

To guarantee the quality of automotive products and overcome several challenges raised by the industry, it may become necessary to define software development standards (Martínez-Fernandez, et al., 2015). Therefore, a partnership between various OEMs and suppliers was established, delivering the Automotive Open System Architecture (AUTOSAR), a reference architecture for automotive software systems.

AUTOSAR was created to standardize automotive software development while promoting software reuse, quality, flexibility, and maintenance during the entire life cycle of the product (Fennel, et al., 2006).

Software development according to this architecture consist of two main phases, the system development, and the ECU development, promoting reuse and cost-efficiency through the separation between software and hardware (Bo, et al., 2010).

The System Development phase is conducted through the specification of software components (SW-C). The SW-Cs are responsible for the AUTOSAR applications, (AUTOSAR, 2008), containing a set of runnables, which represent a "sequence of instructions" (Bo, et al., 2010) with variable length and complexity.

Additionally, other types of components exist and are described next. The last three component types belong to the hardware infrastructure and are defined during the ECU development phase (AUTOSAR, 2008):

- **Sensor/Actuator Components:** handle specific dependencies related to sensors and actuators.
- **Generic Components**: specifies a set of components connected between each other.
- **ECU Abstraction Components:** defines an interface to specific features of an ECU.
- **Complex Driver Component:** provides direct access from an application to certain hardware components.
- **AUTOSAR Services:** standard services offered by the AUTOSAR architecture.

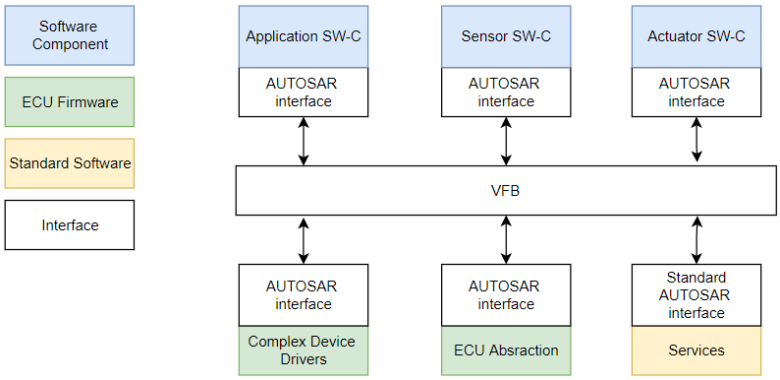

Figure 4 AUTOSAR Components connected to the Virtual Functionality Bus

The independence between hardware and software components is achieved by the Virtual Functional Bus (VFB) (AUTOSAR, 2008), which allows the communication between components and the execution environment, decoupled from the hardware of the system.

In addition, communication also relies on the definition of ports. Each port belongs to a single component, and it is defined as the "point of interaction between a component and other component" in the AUTOSAR technical overview (AUTOSAR, 2008).

To handle the data transmitted across a port, AUTOSAR interfaces must be defined, following two possible patterns (Vector Informatik GmbH, 2018), as represented in Figure 5. The Client-Server interface specifies data transmission through a request logic, where the component that receives the data asks another to perform an operation, returning a certain output. On the other hand, the Sender-Receiver interface defines a communication fashion where a component performs an operation and transmits data to the receiver without any request.

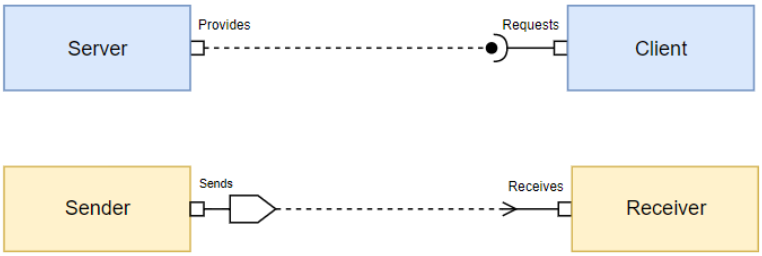

Figure 5 AUTOSAR Communication Patterns

The ECU development phase is carried out considering the AUTOSAR ECU Software Architecture (AUTOSAR, 2008). This architecture consists of three main layers, the Application Layer, the Runtime Environment, and the Basic Software tier, as shown in Figure 6.

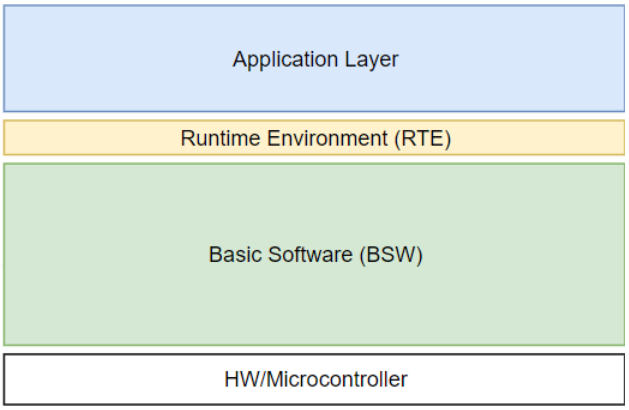

Figure 6 AUTOSAR Layered ECU Architecture (AUTOSAR, 2017)

The Application Layer, or Software Layer, defines the SW-Cs, which are mapped to the ECU and communicate with Basic Software Components through the Runtime Environment (RTE), where the VFB is implemented.

The Runtime Environment delivers interfaces to handle the communications of an ECU, allowing the software layer to remain hardware independent and configured according to the requirements of the system.

Under the RTE, there is the Basic Software Layer (BSW), a standardized layer with services that are used by the Application Layer. The Basic Software Components consist of system services related, for example, with memory management or communication frameworks. Additionally, it also specifies the operating system and the Microcontroller Abstraction Layer, which

delivers interfaces to other Basic Software Components and manages the hardware (AUTOSAR, 2008).

AUTOSAR was created to overtake several obstacles raised by the environment where the automotive software industry is placed. It was concluded that this architecture provides greats benefits. It standardizes automotive software development, promotes software reuse, interoperability, improved communication between teams, and reduced costs, between others (Martínez-Fernandez, et al., 2015).

This architecture presented positive results, being globally accepted by the industry (AUTOSAR, 2021). Beyond that, it is continuously facing new challenges and suffering improvements. The latest version of the standard was released in November of 2020 (AUTOSAR, 2020a), focusing on aspects such as the use of more powerful ADAS, vehicle motion control interfaces, and improved network and communication mechanisms.

2.2 Modern Automotive Software and AUTOSAR Adaptive

Most of the recent advances in the automotive industry continue to be related with software-intensive features. There is a great focus on concepts like connectivity and autonomous functions (Guissouma, et al., 2018). Consequently, it becomes necessary to deliver highly available applications, support dynamic deployments, and the adaptability to different execution environments (Fuerst & Bechter, 2016).

Additionally, there is a need of investing in communication technologies. For instance, to provide access to the cloud, and in-field over-the-air updates, which support automatic and wireless software updates. Such features rely on high-bandwidth communication mechanisms, being also related with challenging requirements, namely in the cyber-security field (Ebert & Favaro, 2017).

The AUTOSAR community has been facing new challenges. There is a need of fulfilling new requirements to support high-performance computing (HPC), increasingly heterogeneous systems, more distributed functions, and the integration of AUTOSAR systems with non-AUTOSAR components and Cloud computing platforms (Fuerst, 2015).

As a result, studies were targeted to AUTOSAR (Fuerst & Bechter, 2016), revealing that the Classic platform is not suitable with these features. It mainly covers the internal infrastructure of the vehicle. Thus, AUTOSAR Adaptive was proposed (AUTOSAR, 2021), seeking intelligent and adaptive software, while coexisting with the Classic platform (Embitel GmbH, 2020), which already offers relevant support on other domains of automotive software.

The Adaptive platform (AUTOSAR, 2020b) delivers more computing power, maintaining the requirements already imposed by the industry and, at the same time, fulfilling the new requirements, with a great emphasis on the hardware capability, communication protocols and the adoption of alternative and more powerful programming languages.

In first place, ethernet becomes vital to provide high-bandwidth communications, due to the transmission of longer messages. Although it is already supported by the Classic platform, it only consists of an optimized version to execute internal and simpler communications. Moving on, there is also the need of comprising more capable hardware through the use of manycore processors, which are flexible, programable, and complex to include using the classic platform.

The adaptive platform follows a service-oriented architecture (SOA) approach. The system is composed of several services used by the applications or other services. Consequently, it allows the scalability and distribution of the system, supporting parallel computation, and taking advantage from the use of the established communication protocols.

Additionally, AUTOSAR Adaptive endorses the incremental deployment of applications. The system is managed dynamically, enabling cyclic integration, and improving the development process and continuous validation during the whole lifetime of the product, in order to reduce development costs.

Finally, the Adaptive platform allows the coexistence between heterogenous ECUs, from Adaptive and Classic systems, or even non-AUTOSAR components, as represented in Figure 7.

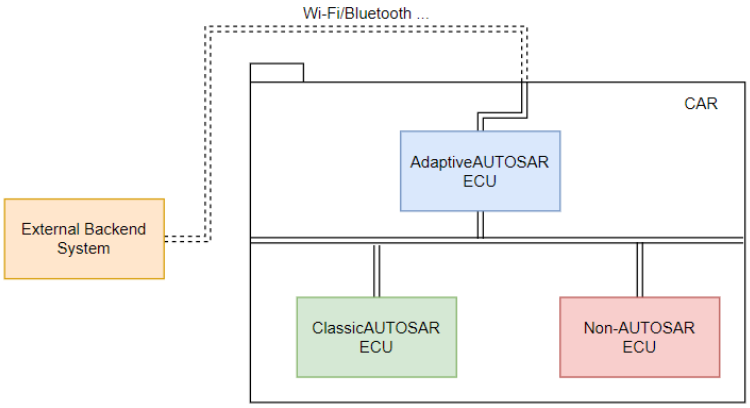

Figure 7 Coexistence of multiple platforms on Automotive Systems (AUTOSAR, 2020b)

Since Adaptive ECUs and Classic ECUs use different communication mechanisms, gateway ECUs can be defined to offer communication between these types of ECUs. The gateway ECU is responsible for processing signals from Classic ECUs into services that can be consumed by the Adaptive ECUs. Otherwise, it may also perform this task by converting Classic ECU's signals into UDP packets and transmitting them to the Adaptive ECUs (Embitel GmbH, 2020).

Like the Classic platform, Adaptive AUTOSAR is structured in layers. It defines software applications that execute on top of the AUTOSAR Runtime for Adaptive Applications (ARA) layer, which provides access to a set of functional clusters supplied by the Adaptive Platform Foundation and the Adaptive platform services (Fuerst & Bechter, 2016).

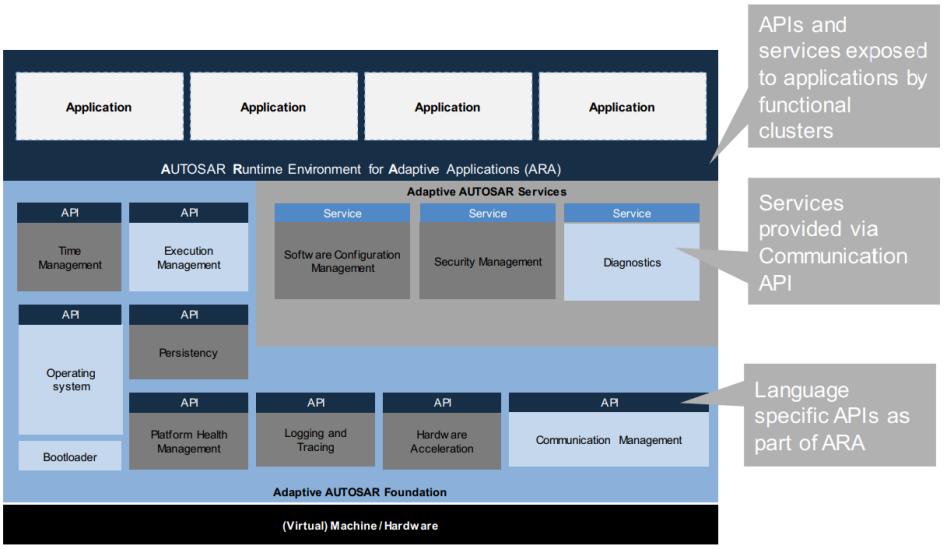

Figure 8 Summary of the Adaptive AUTOSAR Architecture, retrieved from (Fuerst & Bechter, 2016)

This architecture offers support for the integration of applications into the platform with high level of abstraction through programming language interfaces (AUTOSAR, 2020b). The Adaptive Platform Foundation defines processes that provide libraries as a simple function execution method, usually best suited for internal operations. The Platform Services define functions that may be executed remotely, through server-oriented communications.

Additionally, the applications, ECU and OS functionalities can be managed and accessed through the functional clusters. One example is the Communication Management, which specifies Service Oriented Communications, routing requests and replies for intra and inter-machine communications.

The Adaptive platform's operating system is based on POSIX, following the IEEE1003.13 Standard (IEEE, 2004). It provides standard and common functions to the developer, improving the management of software processes. Consequentially, the AUTOSAR applications become increasingly portable, depending on the OS or the APIs defined at the lower layers of the architecture (Fuerst & Bechter, 2016).

From the OS perspective, Adaptive Applications are independent (AUTOSAR, 2020b) and instantiated by an executable, composed by a set of processes, with a unique name and resource allocation. Similarly, the Functional Clusters and services are also composed of processes.

System processes may interact with each other, and the Adaptive Applications can only perform communication through the ARA, even though other applications may use alternative communication interfaces, or direct communication.

Finally, regarding the hardware infrastructure, Adaptive AUTOSAR Platform considers the hardware platform as a machine (AUTOSAR, 2020b). However, this machine can be

considered as a physical node, a fully virtualized machine, a container, or any other type of environment. Like so, an instance of AUTOSAR Adaptive runs on a single machine, belonging to a certain piece of hardware, that may be also composed of other machines.

Efforts are being made to offer new features and keep AUTOSAR up with the technological evolution of automotive systems. Like the Classic platform, Adaptive AUTOSAR is still under development and maintenance, at a less mature stage. The last updates on the architecture were released recently (AUTOSAR, 2020c), focusing on documenting certain specifications, along with the development of new functional clusters and APIs, such as network management interfaces, requirement specification methodologies, and, beyond other features, communication, and security mechanisms, considering, for example, cryptography and intrusion detection management.

There is already a considerable amount of projects using Adaptive AUTOSAR, proving its potential while, at the same time, raising important issues that help the improvement of the platform.

V2X communications were already tested considering the Adaptive architecture (Šandor, et al., 2018). This problem was addressed using the Message Queue Telemetry Transport Protocol (MQTT) in a publish/subscribe fashion, which consists of an asynchronous messaging protocol where a system provides a message to a queue labelled with a topic for routing purposes. Then, systems that subscribe that same topic, are able to consume the messages that were published in that queue.

This solution was developed through the implementation of an MQTT API under the Adaptive Runtime Environment, used to publish and subscribe messages on vehicular systems, and revealing a promising option to establish communication with external entities.

Adaptive AUTOSAR was also tested on ADAS, focusing on vision-based algorithms (Lazic, et al., 2018). This research was conducted using the service-oriented attributes of the architecture and implementing a camera service and an object detection service. The achieved results were successful, delivering an optimized execution, with parallel accesses to the camera service, an object detection algorithm processing images at a considerable frame rate, and a portable implementation, due to the abstraction provided by the platform.

In addition, adaptive code generation was also applied to the architecture, allowing the focus only on the problem and not on implementation details (Stojanovic., et al., 2019), with the automated implementation of vehicular functions, and pattern generation (Nikolic, et al., 2019).

2.3 Automotive Software Model-Based Engineering

The term Model-Based Engineering refers to a software development methodology centred on the specification of models that represent the system, or part of it, from a particular perspective and intent (Mellor, et al., 2003).

MB approaches offer a great level of automation and abstraction, supporting the development of software systems with increased quality and productivity (Selic, 2006). This methodology allows the development team to focus on solving design problems, instead of implementation issues that may arise with the use of programming languages. Additionally, MB techniques deliver improved documentation, design support, maintainability, and simplified standardization (Torchiano, et al., 2012).

The increasing complexity of automotive software systems demand carefully selected development methodologies, and Model-Based approaches have proven to be capable of fulfilling the industry's requirements. Consequently, they became a standard automotive software development technique (Schmidt, 2007).

Challenges were identified since the beginning of MB automotive software engineering, since the models and tools adopted on distinct development processes were not well integrated (Broy, 2006). A System can be developed through several models, each one with a different intent, and a Model-Based framework must focus on every domain of a problem, supporting dynamic environments (AMALTHEA, 2011), and seamless developing processes, in order to allow smooth transitions and the integration between different tools and models of a project. In addition, the "verification and validation (V&V) of the design decisions in early development stages" (Holtmann, et al., 2011) must be established, avoiding unexpected failures and increased costs.

The design of an automotive system begins with the OEM modelling high-level requirements. Then, the produced artifacts are progressively detailed until system models are defined, completely specifying software components. Finally, these components can be implemented by each supplier.

This industry did not have the necessary maturity when it comes to Model-Based approaches (Broy, et al., 2010). Efforts were made to enable integration and continuous validation of the product, the adopted tools, and processes, which can be achieved through standards such as the SPICE® Process Reference Model (Automotive SIG, 2021), an automotive model for computing software development processes.

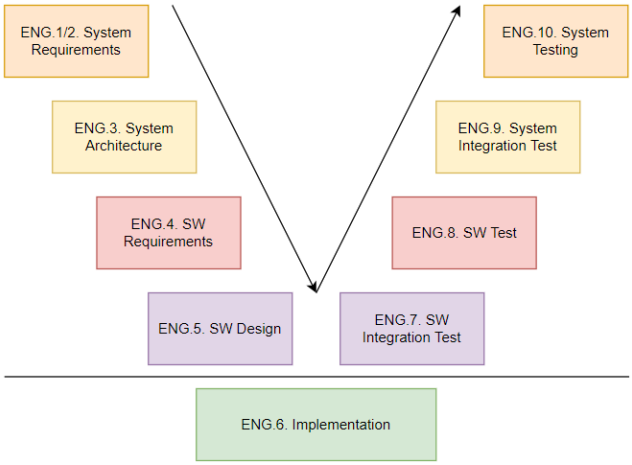

Figure 9 Automotive V-Model based on SPICE® (Holtmann, et al., 2011)

The process starts with the requirements analysis (Holtmann, et al., 2011), possibly handwritten and carried by stakeholders and the client. However, unstructured requirements do not allow model definitions and transformations, so, the requirements must be processed by the development team into a set of structed system requirements integrated in the system model.

As a result, automatic validation becomes possible, along with the integration between requirements engineering and the upcoming stages. It supports the automatic transformation of the requirements model into the system architecture, describing its basic behaviour and main components. The output models must be refinable, and validation can be guaranteed by model simulation tools.

With the conclusion of the System's Architecture design, the software requirement analysis must be carried out. Specific requirements for each system's software component must be defined to proceed with the software design phase (Holtmann, et al., 2011), which, in the automotive industry, is usually supported by AUTOSAR.

Then, the implementation can be conducted, and the produced models may be provided as input to code generators. However, once again, to support continuous validation, it is important to perform model simulations to evaluate the system configurations before conducting the next development phases (Holtmann, et al., 2011).

Finally, with the implementation concluded, the final stages of the project can be carried out, focusing on the evaluation and the integration of all the implemented components.

Model-Based software engineering is widely adopted by the automotive software industry. It is considered as a promising embedded software development technique (Broy, et al., 2010), and the introduction of MB seamless approaches supports smooth developing processes compliant with the industry standards, such as the ISO 26262 (ISO, 2011). Additionally, MB techniques cover the whole lifecycle of software projects, providing features such as the continuous validation and integration of the product (Wolff, et al., 2015a).

In this way, many models and tools are used on automotive software Model-Based engineering (Broy, et al., 2010), and the automotive system models must comprise the multidisciplinary features of the products and processes, instead of a single domain.

Furthermore, Model-Based approaches also focus on complexity management support, due to the constant introduction of software features on automotive systems (Voss & Eder, 2018). Efforts have been made to achieve system traceability, with frameworks such as TORUS, focusing on Cyber-Physical Systems (Dowdeswell, et al., 2016) and also AMALTHEA, supporting flexible development processes, simulation, and the iterative refinement of the system (Wolff, et al., 2015a).

MB methodologies are proving their value in many ways, allowing processes that are suitable with the market. However, several challenges continue to be faced, especially in the last decade. One example is the variability management, due to the need of providing high customization on the automotive industry (Thomas, et al., 2011).

Progresses have been made through the adoption on Products Lines Engineering techniques to provide complexity and variability management, allowing process automation and requirements mapping according to the vehicle's variants and configurations (Polzer, et al., 2012).

Exploratory experiments were applied to MB software development techniques studying Product Line Engineering approaches on automotive systems, and it was proved that it is possible to successfully achieve variability management on complex automotive software systems using this methodology (Bilic, et al., 2018).

On another way, Model-Based technics also focus on testing and validation, from requirement management (Iqbal, et al., 2020), to other stages of the project, namely with the adoption of artificial intelligence mechanisms to perform fault detection (Krejčí & Novák, 2017).

In conclusion, Model-Based Engineering can be considered as a suitable automotive system engineering methodology. It provides several benefits, while conforming with the industry standards. Several projects were conducted to improve MB processes in automotive software, and techniques that, in the past, were only partially used during the development process are now evolving to the entire lifecycle of the product. However, other approaches did not prove to have enough maturity to be adopted (Broy, 2006) and challenges are continuously raised, along with the efforts to improve the adopted methods and technologies.

2.4 Automotive Energy Management and Measurement

This section addresses the importance of the energy domain in the automotive industry. In the first place, it focuses on the energy management and analysis of automotive software systems. Finally, a background on software energy measurement tools and techniques is also provided, along with a critical analysis of alternative approaches.

2.4.1 Energy management in Automotive Software Systems

There is a global effort on reducing gas emissions, with policies being defined to demand less energy consumption (European Commission, 2007). Accordingly, it becomes essential for the automotive industry to leave a smaller energy footprint.

Car makers began to pay more attention to the energy efficiency of the manufacturing processes (Franz, et al., 2016), along with studies about the creation of more efficient solutions (Chiara & Canova, 2012). Additionally, with the rise of electric and hybrid vehicles, projects were also conducted to examine the use of renewable energy in the automotive industry (Rutten & Cobbenhagen, 2019).

On the other hand, modern automotive software expects high-performance computing systems (Fuerst & Bechter, 2016), introducing functions that are transforming the vehicles into more advanced and autonomous products, attended by a huge increase of electronic devices (Armstrong, et al., 2020).

Energy management has always been important for the automotive industry, with software being introduced to provide engine control functions (Broy, 2006). However, the increased amount of software and electronic components is accompanied by an also increased energy consumption (Armstrong, et al., 2020).

Besides that, energy consumption optimization is also important due to the need of executing software functions while the vehicle's engine is off, relying on limited capacity batteries. So, smart power saving mechanisms must be provided, for example, through wake-up periodic policies, managing which components may be standby or shut down, which is reflected in less energy consumptions (Culshaw & Winter, 2007).

AUTOSAR already specifies energy management paradigms, with mechanisms to provide the integral or partial shutdown of system components (Barthels, et al., 2012) and partial networking, managing the active connections between ECUs (AUTOSAR, 2019), and avoiding the use of unnecessary devices and communications.

On another way, there is the possibility of managing the ECU power consumption and performance through the manipulation of the CPU frequency and cores voltage (Barthels, et al., 2012), which may also be integrated into power-aware scheduling policies.

In fact, every time software functions are implemented, more and more electronic devices tend to be included on the vehicle. Cars are becoming more energy costly than ever, which is becoming unsustainable. If, on one hand, software features allow efficient driving. On the other hand, the devices used to offer such capabilities are contradicting that efficiency with an increased resource consumption (Fairley, 2018), as well as the space needed to include this number of electronic devices, and the resultant weight of the vehicle, which starts to be concerning (Hunsley, 2019).

Like so, a potential approach to provide more resource friendly automotive systems consists of increasing their heterogenous features. The number of ECUs can be reduced through the combination of completely different functions on the same hardware components, which is feasible, for instance, using virtualization (Hunsley, 2019).

Finally, in terms of solutions targeting the AMALTHEA Platform, there is a reduced amount of projects considering energy management and optimization techniques. An Integer Linear Programming (ILP) approach, which provides optimal solutions described in mathematical equations, was employed to perform automatic software mapping on hardware components, targeting an increased energy efficiency of systems specified in AMALTHEA models (Krawczyk, et al., 2015).

This approach demonstrates the applicability of automatic optimization mechanisms to AMALTHEA models. However, it consists of a static technique, contemplating only AMALTHEA system models to perform the energy management of the system. On another way, our approach, considers the analysis of software execution or simulation traces. Additionally, the ILP approach calculates the energy costs of the system exclusively through the CPU usage of the executing software, which is not considered the best approach for modern software systems (Singh, et al., 2013).

It is becoming continuously more important to provide the improved energy management of automotive software systems. Areas related to this industry, such as IoT, tend to focus more on the energy domain, due to their increasingly electrified systems. Concepts that are largely relatable to automotive software, such as over-the-air computation, embedded systems, and the aim of reducing gas emissions (Singh, et al., 2020), are becoming more and more important, and may support the evolution and focus on the energy efficiency of automotive software in the future.

2.4.2 Software Energy Measurement Tools and Approaches

Generally, software systems are responsible for considerable greenhouse gas emissions and energy consumptions, not only when applied to the automotive industry. To support the application of energy management approaches, it becomes necessary to define energy metric extraction techniques (Noureddine, et al., 2013), allowing the development of more efficient and energy-aware solutions.

Energy measurements can be performed in a reliable way through the adoption of peripheral hardware (Acar, et al., 2017). On the other hand, there are already software-based solutions which can be used to measure the energy consumption, like Tegrastats (Nvidia, 2021), a power monitor available for NVIDIA boards. However, it is impossible to perform fine-grained measurements with these approaches since they only provide metrics at the system or hardware component levels.

As a result, alternative methods were introduced to provide process-level energy estimation (Noureddine, et al., 2013), such as the PowerAPI (Bourdon, et al., 2011), which presented positive results, with an error margin of nearly 3% (Noureddine, et al., 2013).

This approach offers a solution where the CPU power consumption of a process is represented by the following formula:

$$P_{CPU(d)}^{PID} = P_{CPU}(d) \times U_{CPU}^{PID}(d) \quad (1)$$

For an interval d, the power consumption of the Process PID is equal to $U_{CPU}^{PID}(d)$, the CPU usage of the process, multiplied by $P_{CPU}(d)$, the total CPU power consumption of the system (Bourdon, et al., 2011).

Additionally, the calculation of the total power consumption of the CPU can be estimated according to the following equation:

$$Power_{CPU}^{f,v} = C \times f \times V^2 \quad (2)$$

Here, V represents the CPU's voltage and f represents its frequency, both dynamic values. Finally, the capacitance C of the CPU is a constant calculated through the Thermal Dissipation Power values (Noureddine, et al., 2013).

Later, network and disk energy measurements were also included in this type of approach, in a similar way to tools like pTop, a Linux energy monitor (Do, et al., 2009). Thus, the disk power consumption of a process can be obtained with an equation similar to this one:

$$Power_{process}^{disk} = Read \times Power_{Read} + Write \times Power_{Write} \quad (3)$$

$Read$ and $Write$ represents the number of bytes, or amount of time spent by the process on reading and writing operations. $Power_{Read/Write}$ is the default power consumption of the software performing read or write operations (Noureddine, et al., 2013).

The estimation of the network power usage is performed following the same logic:

$$Power_{process}^{network} = \frac{\Sigma_{i \in states} t_i \times P_i \times \mathrm{d}}{t_{total}} \quad (4)$$

It implies the sum of the network interface's consumption on several states i, for example sending or receiving packets. The variable d represents the monitoring time, P_i the power that the network consumes performing a state i operation, and t_{total} is the total interface's operating time.

This type of approach asserts that the power consumption of a process is directly proportional to its hardware usage (Noureddine, et al., 2013). However, it does not provide much freedom to choose different metrics to estimate the energy consumption of software, since it becomes necessary to establish an equation for each comprised hardware component or metric.

Finally, the energy consumption is induced from energy-related information, such as the CPU voltage and the Thermal Dissipation Power values, which can be obtained through the hardware components documentation (Noureddine, et al., 2013), or measurement tools to which the formulas could be applied. However, this information is not always easily obtainable, and since AMALTHEA traceability solutions mainly focuses on timing properties, especially when it comes to simulation mechanisms, this type of approach may become complex to apply.

Alternative approaches exist, allowing metric estimation through the linear regression of the power consumption and the hardware resource consumption information, such as the Mantis project (Economous, et al., 2006), which proposes a fast and accurate solution to estimate the energy consumption of executing software. In addition, recent works using PowerAPI also support the use of regression models to perform the CPU power consumption estimation through hardware performance counters (Colmant, et al., 2019), which provided a more reliable and comprehensive solution than the one described before.

Regression models can be extended to encompass more hardware components and input data, comprising multiple metrics related to the CPU, memory, disk, and network information, among other components (Singh, et al., 2013; Economous, et al., 2006; Colmant, et al., 2019).

In the first place, it is necessary to conduct a learning process for the calibration of the estimation model. The power consumption of the system and the hardware's resource usage data must be measured. Additionally, benchmarking mechanisms may be employed to stress the system and provide variations to its execution environment.

After collecting the training data, the linear model is obtained through the relation of the collected input metrics and the expected power consumptions:

$$P = P_{Idle} + \sum (n_i \times u_i) \quad (5)$$

With the model established, the power consumption can be calculated through P_{Idle}, the constant consumption of the system in idle mode, summed with the hardware resources information u_i, multiplied with the values estimated by the model for each input metric, n_i (Economous, et al., 2006).

As a result, the power consumption estimation mechanism does not depend on any energy-related data. It is only necessary to provide the hardware resource usage values of the executing software. However, such approaches assume that a linear model is suitable to predict the power consumption of software systems, which may not be verified in more complex environments.

To overcome this problem, alternative approaches can be applied with the aim of achieving more reliable results through a similar process (Singh, et al., 2013), as represented in Figure 10. However, instead of adopting a linear model, a nonlinear regression can be performed

using techniques such as the Support Vector Machines algorithm (SVM) generalized for regression problems, the Support Vector Regression (SVR).

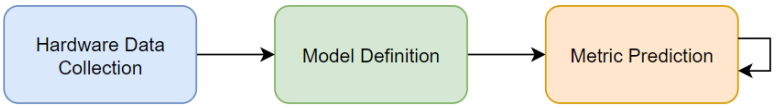

Figure 10 Power Prediction approach on Mantis (Economous, et al., 2006)

SVM is a machine learning algorithm (Pisner & Schnyer, 2020) widely used on software domains due to its great flexibility. It consists of a supervised mechanism, since it is trained in a similar way to the linear regression model, through exemplifying the input data and the expected output values. In this way, it becomes capable of learning and deliver truthful classification patterns and regression solutions.

Like the linear regression approach, this one also generates a model capable of estimating the power consumption through metrics related to the CPU, disk, memory, and network components:

$$P = F(cpu, disk, mem, net) \tag{6}$$

The results of the Support Vector Regression algorithm, comprising these hardware components, were compared to the use of a linear model. An experimental setup was performed for both methods with the same execution environment, samples, and input parameters. In the end, the SVM approach turned out to be more accurate, presenting errors of about 4%, while the linear model presented 15% (Singh, et al., 2013).

In conclusion, a considerable number of approaches allow the estimation and monitorization of the energy consumption of software systems. Some approaches focus only on certain hardware components, and others are not suitable to provide process-level estimation (Acar, et al., 2017), or to operate in complex environments (Singh, et al., 2013). However, there are already techniques that allow the use of several input metrics, and the implementation of reliable solution to perform energy metrics extraction at different granularities.

This table summarizes and compares the process-level approaches addressed in this section, according to the following quality attributes:

- **Average Error:** Indicates the mean error percentage of the approach.
- **Hardware Coverage:** Specifies the hardware coverage of the approach.
- **Energy information Dependency:** Describes how the approach depends on energy-related information of the system.

Table 1 Comparison between Energy Measurement Approaches

	Direct Proportion	**Linear Model**	**SVM**
Average Error	3%	15%	5%
Hardware	Focus on CPU,	Free	Free

Coverage	later applied to I/O and Network components		
Energy Information Dependency	Estimation depends on hardware-specific energy information.	Calibration phase depending on system-level power consumption.	Calibration phase depending on system-level power consumption.

2.5 Eclipse AMALTHEA/APP4MC

Automotive software is constantly evolving, and it is carried out by multidisciplinary and cross-organizational teams. Consequently, great efforts are being made to establish suitable software development methodologies with solutions like AMALTHEA (Eclipse APP4MC, 2021), an open-source Model-Based framework published in Eclipse as APP4MC (Eclipse Foundation, 2021).

APP4MC was established with the support of a number of OEMs and supplying organizations, delivering a platform compatible with the most relevant automotive software standards (Eclipse APP4MC, 2021).

In the automotive industry, continuous integration, and validation across the entire lifecycle of the project is becoming essential (Holtmann, et al., 2011), and this platform provides all these features throughout an iterative design flow (Wolff, et al., 2015a).

In first place, structured requirements are detailed, easing their further validation and the initial specification of the system's features, dependencies, and variants. Then, the architecture is defined based on AUTOSAR, and the behaviour of the software components described. Finally, these components are modelled into tasks and mapped to the hardware and OS infrastructure (Wolff, et al., 2015a).

Additionally, AMALTHEA models can also be used by code-generators to automatically implement the system, along with execution traces and simulations to verify its behaviour from the initial stages of the development process.

This framework provides toolchain tailoring. It presents high extensibility and the compatibility with several third party tools, fulfilling the requirements of large projects that rely on a broad set of technologies, (Wolff, et al., 2015b) presenting different features, complexity, abstraction levels, and cross-organizational teams.

To achieve this goal, one of the most important aspects is the fact that it is Eclipse-based. The Eclipse Foundation provides a great variety of open-source tools that cover all the development phases of a software project. (Eclipse Foundation, 2021). Beyond that, even if a tool is not compatible with AMALTHEA by default, there is always the possibility to easily establish the interaction between them (Wolff, et al., 2015b), as represented in Figure 11.

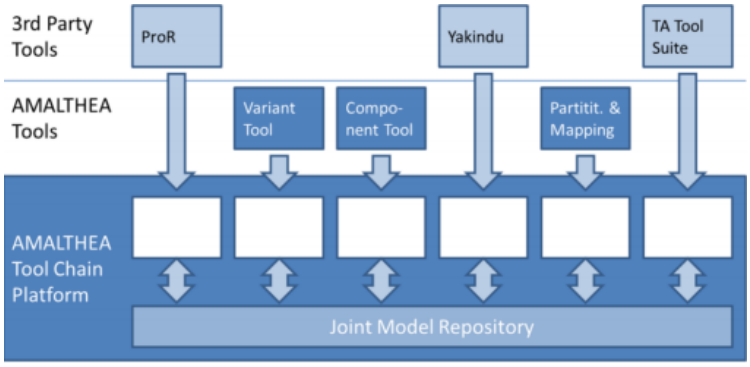

Figure 11 AMALTHEA Toolchain Tailoring Example, retrieved from (Wolff, et al., 2015a)

In this way, APP4MC relies in a central model repository composed of two main models, the System Model, and the Trace Model, as represented in Figure 12 (Eclipse APP4MC, 2020). AMALTHEA models are used to store and exchange data between tools, processes, and team members (Wolff, et al., 2015a), interacting with the model repository to import or export information through the specified interfaces.

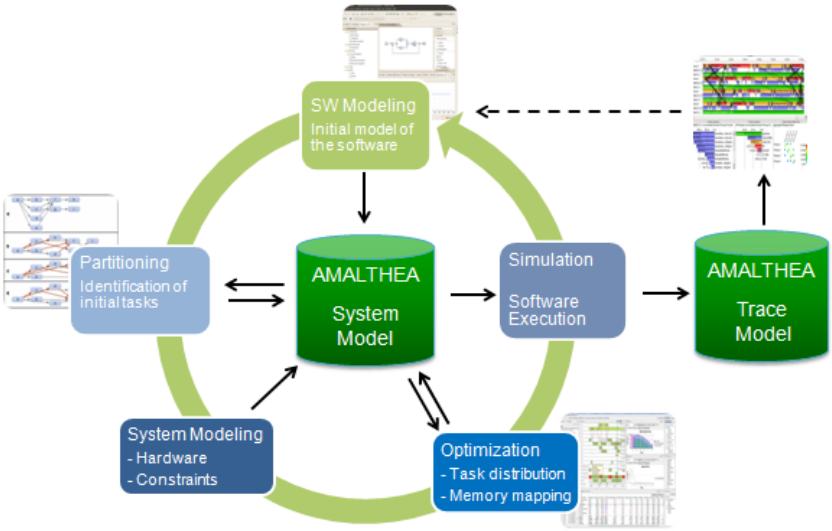

Figure 12 Amalthea Data Models, retrieved from (Eclipse APP4MC, 2020)

The System Model is usually associated with 4 tasks comprised by AMALTHEA (Eclipse APP4MC, 2020):

- **Software Modelling:** Used to specify the behaviour of the software elements of a model.
- **Partitioning:** Responsible for the definition of the relationship between the software elements of the system, allocating them into a set of tasks and defining, for example, their execution order.
- **System Modelling:** Supports the definition of the hardware structure and the constrains of the system, allowing its accurate specification.

- **Optimization:** Consists of a process that can be performed through manual or automatic procedures, influencing the partitioning and the mapping of the software components into the hardware infrastructure.

Finally, this model acts as an input to perform the **Simulation** of the system, which provides information about its execution. The results from a simulation or execution are usually stored in Trace Models, being used for further analysis of the system and providing relevant information to its optimization and continuous refinement (Eclipse APP4MC, 2020).

2.5.1 Related Projects and Technologies

On automotive software engineering, specific technologies are often used to support the various processes of the product development lifecycle (Wolff, et al., 2015b)**.** The integration of these tools is becoming more and more important, especially with the increasing variety of solutions available on the market.

Therefore, although it is mandatory to use APP4MC in the project, this section presents other projects and products that are relatable to the main aspects addressed on the AMALTHEA framework. Five quality attributes are considered to provide a critical summary of the solutions presented here:

- **Open-Source Platform:** Open-source platforms provide free products to be used by anyone according to their needs. Usually, these solutions deliver open and standard methodologies, presenting interoperability and extensibility to the user demands and third-party solutions.
- **Development Life Cycle Coverage:** One of the main goals of seamless frameworks is to support the entire software development process, being suitable with heterogeneous systems and allowing the integration between different teams and stages of a project.
- **Tool Tailoring Support:** Another major concern on seamless toolchains is the ability of increasing the efficiency of projects composed by multiple and cross-organizational teams. Those teams may use different tools according to their needs, so, it becomes essential to provide as much freedom as possible in terms of tool adoption.
- **Active Support:** With the constant evolution of automotive software, for example, due to the introduction of the AUTOSAR Adaptive platform, it is important for a framework or approach to be actively managed.
- **Automotive Standards Support:** Automotive software projects present specific processes and requirements, according to certain standards. The adopted approaches, in this domain, must be prepared to support such features.

In the first place, IBM offers the **Engineering Systems Design Rhapsody** framework (IBM, 2021), also known as Rational Rhapsody. It consists of a commercial Model-Based products

family focused on providing support to the continuous integration and traceability of embedded software solutions, supporting automotive standards such as AUTOSAR.

Rational Rhapsody was designed to provide an efficient agile development environment across the lifecycle of the development process. Like AMALTHEA, this framework is integrated on Eclipse, easing its adaptability to third party software. Still, it consist of a tool suit, unlike APP4MC. Therefore, it aims to provide all the necessary tools to conduct a project.

On the other hand, other platforms focusing on providing the integrated management of a project's lifecycle usually focus on supporting the integration between heterogeneous tools. This type of approach is usually relatable to APP4MC, being based on central repository models that store, manage, and convert data between the project lifecycle stages and heterogeneous tools (Amalfitano, et al., 2017).

The **MOGENTES Project** is an example of this type of approach. Carried out between 2008 and 2011 (CORDIS, 2017), this project proposed a Model-Based solution using a repository model that allows the seamless integration of tools used during different development stages, enhancing traceability in order to provide the automated verification and test generation of the system (Polgár, et al., 2011).

MOGENTES is suitable with automotive and railway systems, supporting standards such as ISO 26262 and AUTOSAR (FTSRG, 2021). Additionally, it delivers tool integration through Eclipse mechanisms (Polgár, et al., 2011). However, it requires each tool to present an exposed interface to provide homogeneity, which may be complex to achieve.

Another seamless Model-Based project is the **CESAR Project** (Griessnig, et al., 2011), created in 2009 and conducted until 2012 (CORDIS, 2017). It focuses on industries like automotive, railway, and avionics, proposing a toolchain for embedded systems through a platform called Reference Technology Platform (RTP), where heterogeneous technologies can be integrated with each other, totally covering the development process, and delivering safety analysis, validation, and traceability.

Among CESAR, the RTP was used by the ARTEMIS Industry Association on other projects, such as the Crystal project (CRYSTAL, 2021). However, the tool's support could not be found.

In addition, tool integration can also be provided by Application Lifecycle Management tools (Amalfitano, et al., 2017) such as **ModelBus** (Fraunhofer FOKUS, 2020). Like AMALTHEA, one of the main features of this platform is the support of heterogeneous tools, automated toolchains, and collaboration between different teams. This tool is currently available for download in the project's website, and the last available version was released in 2016.

ModelBus is an open-source Model-Driven platform, presenting a service-oriented architecture and providing EMF-based models (Fraunhofer Fokus, 2014). It supports model transformation, automation, and traceability through the lifecycle of the project. Therefore, it delivers features that are highly suitable in the automotive industry, although not focusing on specific automotive standards support. To provide automotive-oriented support, some

adaptation may be needed, with the CESAR project, which was described before, presenting an example of the use of ModelBus in the automotive industry, to implement the RTP (Hein, et al., 2013).

IBM also provides a commercial lifecycle management platform, the **IBM Engineering Lifecycle Management (ELM)** (IBM, 2021). This tool offers a seamless development process to all the development teams, with continuous traceability solutions, and enabling simulation and prototyping through modelling languages.

Even though it is not exclusively designed for automotive systems, direct integration with other IBM tools is offered, such as the Rational Rhapsody. On the other hand, the ELM was designed to provide extensibility, allowing the integration of external tools needed by the development team.

Finally, the JetBrains Meta Programming System (MPS), a platform used to specify domain-specific languages (DSL), provides an open-source extension which presents similar features to the ones supported by AMALTHEA, the **MBEDDR** (MBEDDR, 2013). This platform offers a flexible and extensible approach for the specification of embedded software systems, being suitable with the development of automotive software, although there is no evidence of direct support to automotive standards.

The last version of MBEDDR was released in 2018. However, the platform is accessible and active support is offered. Finally, this framework supports the development process throughout the project's entire lifecycle, providing means to conduct, for example, the requirements definition, evaluation, and documentation process. Additionally, it allows the specification of customized development environments through the integration of external tools.

To summarize the analysis of AMALTHEA and the platforms presented here, the compliance level of the quality attributes defined before is presented in the table below. Each attribute can be considered as completely fulfilled (Yes), partially or hard to be fulfilled (Partial), or not fulfilled at all (No).

Table 2 Evaluation of AMALTHEA's related approaches quality attributes compliance

	Open-Source	**Life-Cycle Management**	**Tool Tailoring**	**Active Support**	**Automotive Standards Support**
APP4MC	Yes	Yes	Yes	Yes	Yes
IBM Rational Rhapsody	No	Yes	Partial	Yes	Yes
MOGENTES	Yes	Yes	Partial	No	Yes
CESAR	Yes	Yes	Yes	No	Yes
ModelBus	Yes	Yes	Yes	Yes	No
IBM ELM	No	Yes	Yes	Yes	Yes
MBEDDR	Yes	Yes	Yes	Yes	No

3 Value Analysis

During the World War II, General Electric proposed the concept of Value Analysis. It consists of a process intended to provide valuable products to the customer, considering a problem, an opportunity, and alternative solutions or related works. The Value Analysis is a systematic analysis and evaluation process (Neap & Celik, 1999) regarding the product's purpose and establishing its value associated with the customer needs. Finally, the Value Analysis aims to provide as much value as possible, while minimizing the associated costs (Rich & Holweg, 2000).

This chapter describes the Value Analysis of the project, starting with the Innovation Process, where the opportunity is identified, and possible approaches are studied. Then, it focuses on the Solution Value, where the value proposition is clarified. Finally, the Functional Analysis is conducted, establishing the essential functions of the solution, in order to successfully fulfil the project's goals.

3.1 Innovation Process

The innovation phase of a product can be defined by three mains stages (Koen, et al., 2002):

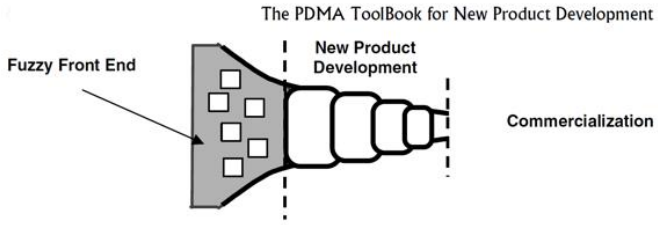

Figure 13 Innovation Process (Koen, et al., 2002)

The first stage is the **Fuzzy Front-end**, ranging from the initial problem definition and analysis to the product development approval, proposing and evaluating approaches (Murphy & Kumar, 1997). After that, the **New Product Development** can be performed, and finished by the **Commercialization** phase, where the solution is promoted and distributed in the market (Koen, et al., 2002).

The Fuzzy Front-end has an extreme impact on the product, due to the freedom and reduced costs related to implementation changes. Unlike the next stages, it tends to be more informal and experimental, due to the lack of information and research on the early stages of the innovation process (Herstatt & Verwoen, 2001).

Therefore, the results of the front-end phase are not always predictable, and efforts were made to provide a standard and organized approach, with the New Concept Development (NCD) model (Koen, et al., 2002).

The NCD provides a systematic and well-structured methodology, with a careful and goal-oriented plan. As a result, the whole process becomes controlled and predictable, with a definable commercialization date, a studied budget, and carefully analysed expectations.

3.1.1 Opportunity Identification

The opportunity identification is the first step of the NCD (Koen, et al., 2002). It consists of identifying an opportunity to solve a problem or provide beneficial results driven by the ambitions of a project or business. Several motivations may induce the opportunity identification, such as covering new domains or reducing costs.

In the project, the problem is explained in more detail in the Introduction, during the Problem Statement. In short, during the last years, the amount and complexity of software in automotive products have increased abruptly, along with the need of providing more energy efficient solutions.

The AMALTHEA/APP4MC Model-Based framework covers the entire development lifecycle of automotive systems. However, it mostly encompasses the timing properties of the system, especially considering the platform's simulation and traceability solutions.

Consequently, there is a need of providing energy properties annotation approaches through AMALTHEA models and automotive software development methodologies, integrated with energy metric extraction mechanisms to support the adoption of improved energy analysis and management techniques.

3.1.2 Opportunity Analysis

The **Opportunity Analysis** is performed after the problem identification (Koen, et al., 2002). This stage can be applied according to formal or informal methodologies, verifying if the identified opportunity is valuable enough in order to justify the development of the product.

In this project, this process was conducted through the application of a **SWOT** analysis, a technique that aims the support of strategic management and decision support on projects and organizations. It considers two main elements, the internal environment, related to the organization or the project itself, and the external environment, usually related to the industry and market where it is positioned (Guerel & Merba, 2017).

Consequently, the SWOT is widely used. It provides a simple approach, allowing the identification of the most critical factors of a project through the following elements:

- **Strengths:** Internal Factors that influence the project positively

- **Weaknesses:** Internal Factors that influence the project negatively
- **Opportunities:** External Factors that influence the project positively.
- **Threats:** External Factors that influence the project negatively.

Some aspects can be very crucial to the SWOT analysis, such as the product, the organization, and the industry. In this case, one important factor is AMALTHEA/APP4MC. It is mandatory to adopt this framework on the project, which influences both internal and external factors.

The SWOT analysis of the identified opportunity is summarized in Figure 14.

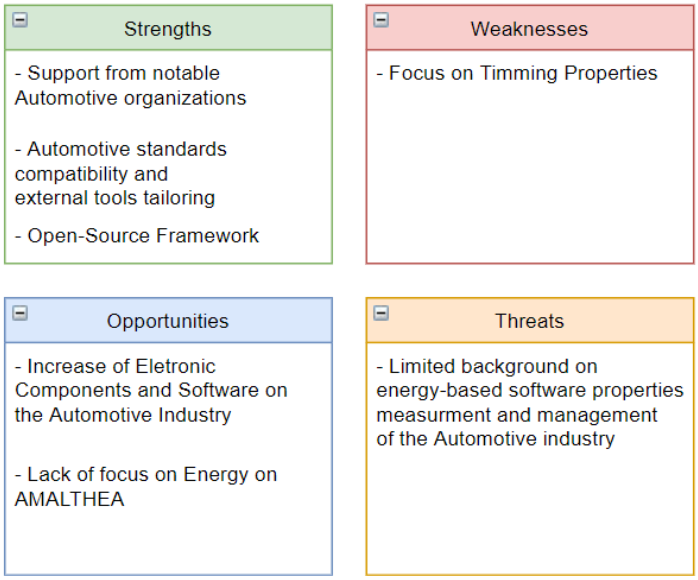

Figure 14 Application of the SWOT Analysis to the project

In the first place, **Strengths** are represented. The identified strengths are mostly related to the AMALTHEA framework, starting with the fact that this platform is maintained by important members of the automotive industry. So, the solution is likely to be adopted by relevant audiences. In addition, AMALTHEA is designed to be fully compatible with automotive software, supporting the industry's most important processes and standards, which is a benefit, since the project focuses on automotive software development.

Finally, AMALTHEA is open-source and offers external tool compatibility, with freedom to use and adapt the platform according to the project needs. On the one hand, it possibly reduces the limitations of the adopted approach. On the other, the results and contributions may become valuable to a broader community.

In terms of **Weaknesses**, AMALTHEA's simulation and traceability mechanisms are mostly focused on the timing properties of the system. It may be more difficult to use the platform applied to energy domains, raising possible obstacles to the project.

Moving into the identified **Opportunities**. First, the amount of software and electronic components on cars is steeply growing, along with the energy consumption of the vehicle. So,

there is a need of supporting improved energy management techniques. Consequently, the lack of energy-focused approaches on AMALTHEA, besides being a weakness, also becomes an opportunity, since the project focus on improving energy management support, providing relevant value to the community.

Finally, **Threats** were considered due to some challenges finding a significant background and State of the Art on energy measurement and management approaches applied to the automotive industry. Most of the work found was mainly related to concepts such as fuel consumption and manufacturing processes and not on software, which might suggest some obstacles to fulfil the project goals.

3.1.3 Idea Genesis

At this stage, with the opportunity analysis concluded, it becomes necessary to evolve from opportunities to concrete approaches to the problem, through the **Idea Genesis** (Koen, et al., 2002). This stage consists of a continuous and iterative process since ideas and approaches may be changed during development and knowledge acquisition processes.

In first place, it is mandatory to use the APP4MC toolchain. So, the analysis of alternative platforms is irrelevant. On the other hand, a study on energy measurement and estimation approaches is provided in Section 2.4.2. As a result, the Idea Genesis is focused on software energy consumption estimation techniques. The implementation of such mechanism is a major component of the project. Additionally, there are few or no requirements and constraints limiting the possibility of selecting various alternatives.

In conclusion, the state of art study identified the following alternative approaches. Each approach presents distinct features, allowing a quantitative comparison between them:

- **Energy Consumption estimation through Direct Proportion Formulas (DPF).**
- **Energy Consumption estimation through Linear Regression Models (LR).**
- **Energy Consumption estimation through Support Vector Machines (SVM).**

3.1.4 Idea Selection

According to the NCD, the analysis of the alternative ideas is performed during the **Idea Selection** phase (Koen, et al., 2002). One problem can be solved with multiple approaches and several selection methods may be applied, influencing the value of the final product.

The idea selection of the project is conducted with the Analytic Hierarchy Process (AHP) (Saaty, 2008). To perform idea selection using AHP, in first place, a problem must be defined, which allows the construction of a decision's hierarchy. It specifies the main objective at a top level, the criteria of a suitable approach at mid-level, followed by alternative approaches at a lower level.

Finally, it is necessary to create a pairwise comparison matrix, to establish the importance weights of the criteria elements. After all, for each criterion, the alternatives are also compared in pairs and the obtained results define which is the most suitable alternative to solve the problem.

The decisions hierarchy of this project is represented in Figure 15.

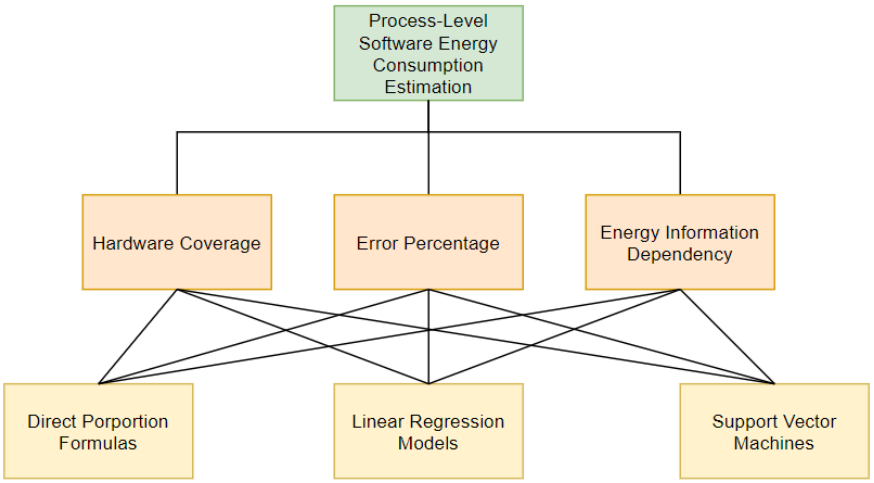

Figure 15 Decisions Hierarchy of the Project

The main goal of the idea selection is to establish an approach to perform **process-level software energy consumption estimation**. To evaluate and choose the most suitable solution to this problem, three criteria are considered:

- **Hardware Coverage (HWC):** The amount of hardware components considered by the approach, or the ability to comprise more input hardware metrics.
- **Error Percentage (EP):** The error margin presented by the solution.
- **Energy information dependency (EID):** How the approach rely on energy information to provide metrics estimation.

The next step consists of establishing the criteria' importance throughout a pairwise comparison, using the values specified on the fundamental scale of absolute numbers.

Table 3 Fundamental Scale of Absolute Numbers, based in (Saaty, 2008)

Importance	**1**	**3**	**5**	**7**	**9**	**2,4,6,8**
Definition	Equal	Weak	Strong	Very Strong	Absolute	Intermediate values

The most important criteria are the **HWC** and **EP**. The adopted mechanism must be able to comprise as much information as possible to perform the metrics estimation, providing more comprehensive results, which highly influences the reliability and flexibility of the mechanism. On the other hand, it must be as more accurate as possible, presenting a low error margin. Consequently, the hardware coverage and error percentage present the same importance.

Finally, the **EID** is only weekly less important than the others. The use of energy information to perform the metrics estimation may limit the applicability of the adopted mechanism to AMALTHEA, namely, when simulation traces are used. Thus, it was still considered as an important factor for the idea selection process.

The comparison between the criteria can be verified in the following table:

Table 4 Pairwise Comparison Matrix of the Criteria

	HWC	**EP**	**EID**
HWC	1	1	3
EP	1	1	3
EID	1/3	1/3	1
Sum	**7/3**	**7/3**	**7**

The following step is to normalize the comparison matrix, since the criteria priorities are calculated through their eigenvector, which consists of the mean of each normalized matrix's line (Bologa, et al., 2016).

Table 5 Normalized Comparison Matrix and Priority Vector

Normalized	**HWC**	**EP**	**EID**	**Priorities**
HWC	0,429	0,429	0,429	**0,429**
EP	0,429	0,429	0,429	**0,429**
EID	0,143	0,143	0,143	**0,143**

With the priorities defined, it is now necessary to perform the consistency check. In the first place, the maximum eigenvalue, λmax, must be calculated, through the multiplication of the comparison matrix with the priorities vector. In the end, the λmax is equal to the average of the division between the result and the eigenvector:

$$Ax = \lambda_{max} x = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 1 & 3 \\ \frac{1}{3} & \frac{1}{3} & 1 \end{bmatrix} \times \begin{bmatrix} 0.429 \\ 0.429 \\ 0.143 \end{bmatrix} = \begin{bmatrix} 1.286 \\ 1.286 \\ 0.429 \end{bmatrix} \quad (7)$$

$$\lambda max = Average\left(\frac{1.286}{0.429} + \frac{1.286}{0.429} + \frac{0.429}{0.143}\right) = 3 \quad (8)$$

Then, the calculation of the Consistency Index (CI) is performed:

$$CI = \frac{\lambda \max - n}{n - 1} = \frac{3 - 3}{3 - 1} = 0 \quad (9)$$

Finally, the Consistency Ratio (CR) can be obtained through the division of CI with the Random Average Consistency Index, defined through the criteria' size, which is 3. If the obtained value is less than 0.1, the criteria weights are consistent.

Table 6 Random Consistency Index Table

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RI	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

$$CR = \frac{0}{0.58} = 0\ (< 0.1) \tag{10}$$

In this way, since $0 < 0.1$, the criteria' weights are consistent and the identified alternatives comparison is now possible (Bologa, et al., 2016).

As in the criteria comparison, the next phase consists of the alternatives pairwise comparison, for each criterion. Each pairwise comparison matrix is then normalized, and the eigenvector is calculated.

First, the **HWC** criterion was evaluated. In terms of hardware coverage, the **SVM** and the **LR** alternatives, present the same values, with a strong difference to **DPF** solutions, which presents less coverage and flexibility to include metrics. So, the comparison is represented in the following table:

Table 7 Pairwise Comparison of the Alternatives for HWC

HWC	**DPF**	**LR**	**SVM**	**Priorities**
DPF	1	1/7	1/7	0.067
LR	7	1	1	0.467
SVM	7	1	1	0.467

The next evaluated criterion is the **EP**. In terms of error percentage, the **DPF** solution presents a value of 3%, **SVM** presents 5% and **LR** 15%. It was considered that the **DPF** is almost negligibly better than **SVM** and moderately better than **LR**. Accordingly, **SVM** is weakly better than **LR**:

Table 8 Pairwise Comparison of the Alternatives for EP

EP	**DPF**	**LR**	**SVM**	**Priorities**
DPF	1	4	2	0.557
LR	1/4	1	1/3	0.123
SVM	1/2	3	1	0.320

In terms of the **EID**, both **LR** and **SVM** present the same energy information dependency, only during a hardware calibration phase. On another way **DPF** depends on energy information to

perform the metrics extraction. Therefore, it can be considered that **LR** and **SVM** are strongly less dependent of energy information to perform metric estimation than **DPF**:

Table 9 Pairwise Comparison of the Alternatives for EID

EID	DPF	LR	SVM	Priorities
DPF	1	1/5	1/5	0.091
LR	5	1	1	0.455
SVM	5	1	1	0.455

Finally, after the comparison between the alternatives, it is necessary to obtain their composed priorities, multiplying the obtained comparison values with the criteria's eigenvector (Bologa, et al., 2016). The highest value of the resultant vector corresponds to the most suitable alternative:

$$\begin{bmatrix} 0.067 & 0.557 & 0.091 \\ 0.467 & 0.123 & 0.455 \\ 0.467 & 0.320 & 0.455 \end{bmatrix} \times \begin{bmatrix} 0.429 \\ 0.429 \\ 0.143 \end{bmatrix} = \begin{bmatrix} 0.280 \\ 0.317 \\ 0.402 \end{bmatrix} \begin{pmatrix} DPF \\ LR \\ SVM \end{pmatrix} \quad (11)$$

According to the obtained results, it is possible to conclude that the **SVM** approach appears to be the most suitable alternative to fulfil the objective, considering the defined alternatives, criteria, and the importance assigned to each one of them.

3.2 Solution Value

At this point, the description of Fuzzy Front-End is concluded with the analysis of opportunities and ideas to provide a solution to the addressed problem. In this way, this section, approaches the concepts of Value and Perceived Value, along with an analysis of the project, and the specification of its Value Proposition.

3.2.1 Value and Perceived Value

Value is fundamental for any product, business, or solution. This concept has been widely studied by several organizations, and it is considered as something that influences the customer's desire to obtain a product according to his needs, defined as the cost of a product with a marginal value depending on the customer (Neap & Celik, 1999).

Customer satisfaction is highly relatable to the value of a product. However, it does not necessarily help the organization to integrate new customers, it only allows the company to improve its performance from the perspective of already existing ones (Eggert & Ulaga, 2002).

Like this, the Perceived Value, or Customer Value, is considered as the way that the customer recognizes the value of a product. It can be subjective, varying between customers according

to different aspects. On another way, the perceived value can also be the perception of the product value to the customer, from the organization's perspective (Woodall, 2003).

Perceived Value becomes increasingly important to the market. It is different than customer satisfaction, since it allows the evaluation of a product after and before its adoption, analysing not only the solution provided by the company but also alternative ones, focusing on the actual and potential customers through the adoption of a strategic orientation (Eggert & Ulaga, 2002).

Finally, the Perceived Value can be defined as a set of drivers related with two main factors, the customer benefits, and sacrifices, which can be associated products, services, or even relationships (Lapierre, 2000).

3.2.2 Value Proposition

The Value Proposition describes products and services offered by a company and how they provide value to the customer (Petrovic & Kittl, 2003), attracting new clients and increasing the satisfaction of the existing ones.

When new products and ideas arise, it is necessary to create the Value Proposition, which increases the value perceived by potential consumers. It is considered that a proper Value Proposition consists of a small statement, usually up to two sentences (MaRS, 2012). So, it must provide, in a summarized way, the value of the solution, which is possible by answering a set of questions:

- What is the Product?
- From whom is the value provided?
- What value is provided?
- Why is the product unique?

Therefore, in this section, the project is analysed, and these questions are answered, serving as input to the Value Proposition Statement. It is important to mention that this work does not deliver a specific product, but mostly the establishment of an approach. To describe its value proposition, the project itself along with APP4MC, the adopted framework, are considered.

What is the Project? This project is conducted to study an approach to the annotation of non-functional properties information focusing on the energy domain. It is applied to automotive software development methodologies and models, using the AMALTHEA/APP4MC framework.

For whom is the value provided? The target customer can be considered as the automotive software development community, or more specifically, the users of the AMALTHEA framework.

What Value is provided? In the previous chapters, 1 and 2, the problem statement is deeply described and analysed, supporting this question.

In review, the automotive industry has suffered considerable changes. A dramatic software complexity and usage growth is being experienced, with functions constantly being introduced, and increasing the electrification of cars. Besides, there is a great concern with gas emissions and the adoption of electric and hybrid cars. So, challenges were raised related to energy management and consumption of automotive systems.

AMALTHEA/APP4MC is a Model-Based toolchain that supports the entire lifecycle of automotive software projects. APP4MC is composed of open-source technologies and allows free integration of external tools, providing high flexibility. In this way, the development process allows the seamless integration between the various teams working for the same projects, which, in this industry, may belong to different organizations and focus on distinct domains, or stages of the product development lifecycle.

However, APP4MC is mainly focused on the timing properties of the system, especially when the simulation and traceability solutions of the platform are considered, presenting a lack of energy requirements coverage.

Therefore, there is a need of providing non-functional energy properties annotation approaches on automotive software dynamic and adaptive systems, using the AMALTHEA/APP4MC toolchain, in order to support an improved energy management of automotive software systems.

Why is this product unique? APP4MC presents a small amount of support and projects related to management and extraction of energy properties, namely when it comes to the analysis of system traces, which mostly encompasses the timing behaviour of the system.

On the other hand, as described in 2.4.1, a great part of the work performed on the automotive industry on energy management is related to manufacturing processes and engine-level consumption, so, there is also a great need of investing in the energy management of automotive software systems.

Value Proposition Statement:

AMALTHEA/APP4MC is an open-source flexible toolchain that allows seamless approaches to automotive software projects, developed by cross-domain and cross-organizational teams. However, there is not much work related to energy properties management using this framework, which is becoming progressively more important due to the increasing software complexity and electrification of automotive systems. So, this project provides an energy properties annotation approach, focusing on automotive software development methodologies for dynamic and heterogeneous environments, using the AMALHTEA framework.

3.3 Functional Analysis

To conclude the Value Analysis, a functional analysis was performed, adopting the Function Analysis System Technic (FAST) Diagram. This diagram is used on value management since its creation, providing a simplified and graphical view of the system and its functions, along with the promotion of creativity and the focus on technical concerns, which contributes to a better understanding of the problem and a suitable solution (Borza, 2011).

The concept of function is associated with an operation offered by the system, usually defined by an **"Active Verb + Measurable Noun",** when applied to FAST diagrams. Thus, it imposes a clear definition of tasks and the logical relationships between them (Borza, 2011).

This diagram presents two main function categories, the **Basic Function**, as a main feature of a product, and the **Secondary Function**, as other functions that contribute to the Basic Functions satisfaction. The Secondary Functions may be considered as **Dependent Critical Functions**, which are mandatory to fulfil a Basic Function, or **Independent Functions**, which are not mandatory, although they improve the performance of Basic Functions.

In another way, Secondary Functions can be part of the **Design Criteria**, as performance requirements appliable to the entire system, and directly related to a Basic Function. Additionally, they may be considered as **All-The-Time Functions**, similar to Design Criteria, although as a property delivered by the product, such as minimum quality levels not directly related to a Basic Function.

Finally, the FAST diagram uses a right-to-left convention, with the high-order function on the left and the low-order one on the right. From higher-order to low-order functions, the question **How** is responded. On the opposite way, the answered question is **Why.**

In conclusion, from the higher-order function to the lower-order function, there is always a critical path. Secondary functions may be introduced on the diagram, bellow or above the critical path functions, executing **When** the critical path function to which they are connected is performed (Borza, 2011).

Figure 16 presents the FAST diagram developed in the scope of the project.

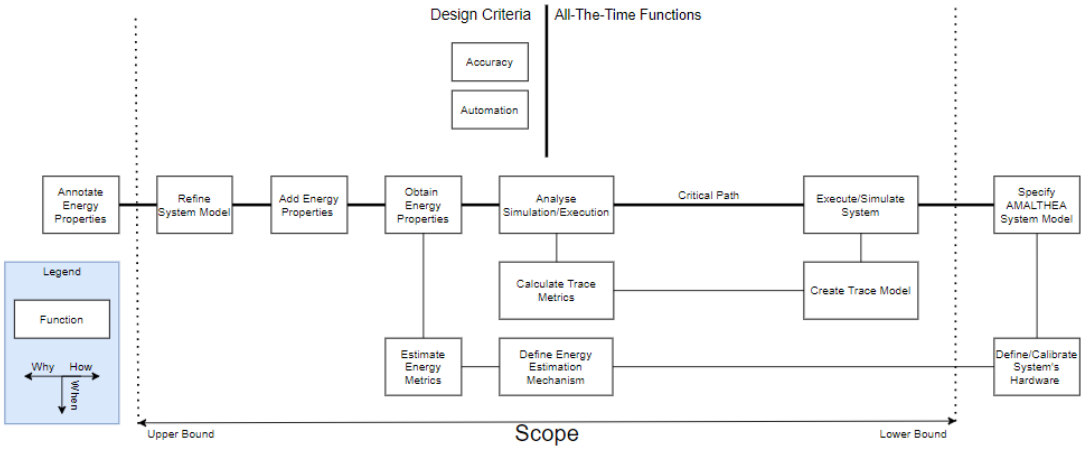

Figure 16 Project's Approach FAST Diagram

On the left side of the diagram, the higher-order function is represented, consisting of the **energy properties annotation** in AMALTHEA/APP4MC models. Within the scope of the project, model annotations are achieved by refining the AMALTHEA System Model under analysis through the inclusion of energy properties.

During the energy properties obtention process, the energy estimation model defined and calibrated according to the specification of the system's hardware infrastructure must be employed to perform the estimation of energy metrics.

Additionally, this process relies on a set of hardware resource usage metrics calculated with the information stored in AMALTHEA trace models, during the analysis of the system's execution or simulation. In turn, the trace model is created when the system under analysis is executed, through the monitorization of a simulation, or the execution of the implementation of a system specified in AMALHTEA.

Finally, considering the design criteria, the approach must provide an accurate metric estimation mechanism. In addition, the entire process must be automated, avoiding manual steps and matching the requirements imposed by automotive adaptive and dynamic software environments.

4 Solution Analysis

To conduct the development of a software product it is necessary to perform the solution analysis, understanding the problem and identifying the requirements (Larman, 2002), which highly influences the future stages of the project.

This chapter presents the analysis of the approach, starting with a domain analysis considering the AMALTHEA System Model and the study of the AMALTHEA Trace Database. Finally, the use case definition is provided, along with the description of the solution's functional and non-functional requirements.

4.1 AMALTHEA System Model

The analysis of the problem's domain is crucial for a better understanding of the environment where the solution is incorporated, considering the most relevant concepts, elements, and relationships. Since the main goal of the approach is to provide an annotation mechanism to the support of energy management and analysis of AMALTHEA models, the domain analysis targets the AMALTHEA System Model offered by APP4MC.

The System Model (Eclipse APP4MC, 2020) was developed using the Eclipse Modelling Framework (EMF), an open-source tool that allows the definition of metamodels, also providing code generation mechanisms (Eclipse Modeling Framework, 2021). Additionally, the system model is composed of a set of logical sub-models, which may be referenced between each other or external models, promoting reusability, and avoiding duplication (Eclipse APP4MC, 2020).

Figure 17 represents a domain model resulting from the analysis of the AMALTHEA System Model. Only the elements and relationships that are relevant to the project are described and analysed in the following sections, each one belonging to one of the System Model logical parts, represented in different colours.

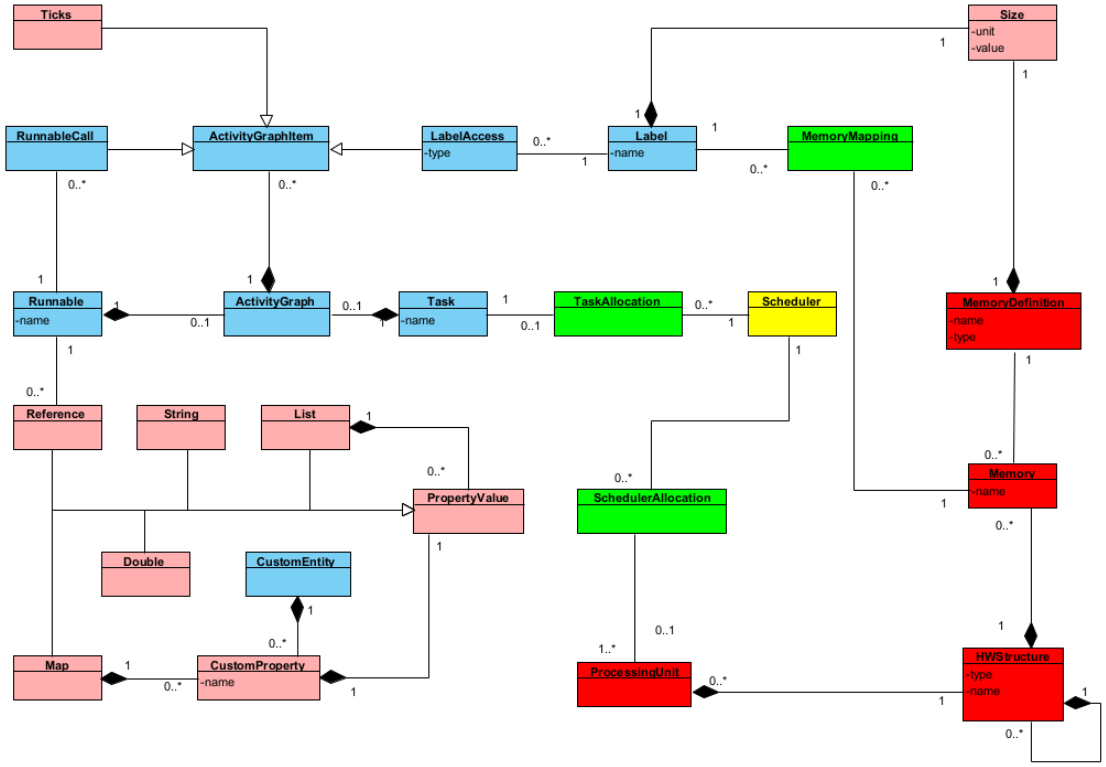

Figure 17 Domain Model based on the AMALTHEA System Model

4.1.1 AMALTHEA Model Basics Entities

The AMALTHEA Model Basics entities are represented in the model in pink. Contrarily to the rest of the described components, the Model Basics do not consist of an AMALTHEA sub-model. Still, it provides several classes to describe specific attributes and information of elements from the other logic models (Eclipse APP4MC, 2020).

Custom Properties:

The custom property consists of a generic key-value entity that provides information to a model when its standard specification is not available on AMALTHEA. It can be generalized to a specific type, such as the following ones, which are employed by the solution to perform model annotations:

- **Map:** Key-Value pairs container of custom properties
- **String:** Alphanumeric value
- **Reference:** Reference to other entities
- **Double:** Floating numeric value
- **List:** List of values

Ticks:

Ticks can be considered as a simple timing unit typically used to specify the amount of computation needed to execute anything on a processor. Since different cores may present different execution times to perform the same operation, in AMALTHEA, ticks are usually adopted instead of an alternative timing measurement unit.

4.1.2 Software Model Entities

The AMALTHEA Software Model aims the specification of software components and their dependencies, defining the behaviour of the system with different abstraction levels (Eclipse APP4MC, 2020). The software entities that are relevant for the solution are represented in blue in the domain model.

Labels:

Labels specify data elements located in a memory area. They are used to store data and share resources between software elements. A label may belong to primary memories, or secondary memories.

Tasks:

Tasks are considered as system processes. Although AMALTHEA allows the definition of a task with all the operations described in the activity graph. Within the scope of this work, inspired by the AUTOSAR specification and the use cases to which the solution was applied. The unique operation that is performed by a task is to trigger the execution of a runnable instance. In this way, the analysis is performed at the runnable level, so, other task operations are ignored, such as label accesses.

Runnables:

The runnable is the atomic executable entity in AMALTHEA. The difference between a runnable and a task is that the task is executed in the context of the OS, while the runnables are triggered by a software entity.

ActivityGraph:

The activity graph is a container of the operations performed by each runnable or task of the system. One activity graph is composed of a set of activity graph items.

ActivityGraphItems:

The activity graph items can be used to provide a detailed description of the tasks and runnables execution. The elements considered in the solution are the following:

- **LabelAccess:** Access to a memory label
- **Ticks:** Generic specification of the required execution time expressed in ticks

- **RunnableCall:** Runnable activation Trigger

CustomEntities:

Custom entities are a generic approach offered by APP4MC to specify software entities or information in case they are not covered by the framework. A custom entity only allows the definition of a single string attribute and a set of custom properties, being used by the solution as an annotation container.

4.1.3 Hardware Model Entities

The AMALTHEA system hardware is provided by the Hardware Model (Eclipse APP4MC, 2020). This model is organized through a hierarchical structure, and its elements are represented in the domain model in red.

MemoryDefinition:

The memory definition specifies a type of memory which can be instantiated, defining attributes such as its type and size.

ProcessingUnitDefinition:

Like the memory definition, this class defines an instantiable processing unit, such as a core or a GPU hardware component.

HWStructure:

Hardware structures are hierarchical elements composed by other HW Structures and HW elements, such as memories and processing units.

Memory:

The memory is the instantiation of a memory definition, which can be mapped to memory labels, providing access to information about the hardware resources of the system.

ProcessingUnit:

The processing unit consists of a processing unit definition instantiation, where the system runnables and tasks execute. Like the Memory element, the processing unit specifies information about the configurations of the hardware platform.

4.1.4 Operating System Model Entities

The Operating System Model, specifies the OS (Eclipse APP4MC, 2020). One model may instantiate several operating systems and several types of operating systems may also be

specified. In the solution's domain model, there is only one element from the OS model represented, in yellow:

Scheduler:

One of the responsibilities of the OS is software scheduling. APP4MC offers different types of scheduling algorithms, defining how software tasks are managed. Within the scope of the solution, the scheduler provides a bridge between the Software Model and Hardware Model.

4.1.5 Mapping Model Entities

The Mapping Model is responsible for the definition of the OS and software elements allocation into the hardware platform (Eclipse APP4MC, 2020). It is composed of the following entities, which are represented in the domain model in green:

SchedulerAllocation:

The scheduler allocation specifies the set of processing units of the hardware infrastructure that are managed by a certain scheduler.

TaskAllocation:

This class can be used to define which scheduler is responsible for managing a certain software task.

MemoryMapping:

The Memory Mapping element allows the explicit allocation of labels on existing physical or virtual memory pieces, which can be performed defining specific memory sectors.

4.2 AMALTHEA Trace Database model

To obtain the energy properties that are further annotated in AMALTHEA models, the approach relies in the analysis of system traces. Consequently, since AMALTHEA/APP4MC already provides a standard Trace Model. This model, along with the AMALTHEA System Model, are considered as the main data sources of the solution. However, it is important to mention that the framework presents a high level of extensibility, which allows the adoption of other trace formats and mechanisms.

Unlike the system model, the AMALTHEA Trace Model is defined as a relational database, being also referred as AMALTHEA Trace Database (ATDB) (Eclipse APP4MC, 2020). It describes traces through different events and entity states. In addition, the ATDB offers automatic mechanisms for the calculation of several metrics related to the data persisted in the database.

In this way, the AMALTHEA Trace Model consists of an SQLite Database (SQLite Consortium, 2021). The ATDB's structure is based in the Best Trace Format (BTF), a trace format for the support of system-level analysis targeting embedded real-time multicore systems (Vector Informatik GmbH, 2020). BTF consists of a CSV-based format that can be used within several simulation tools to describe system events in a chronologic way, along with the evolution of the software entities' states. Like AMALTHEA, this trace format is compatible with automotive standards such as AUTOSAR and OSEK (International Organization for Standardization, 2005), an automotive Operating System.

The APP4MC toolchain supports the automatic transformation of BTF traces into ATDB databases, also allowing the conversion of an ATDB instance back to a system model, providing a bidirectional relationship between them (Eclipse APP4MC, 2021).

Finally, the structure of the AMALTHEA Trace Database is illustrated in the relational model represented in Figure 18:

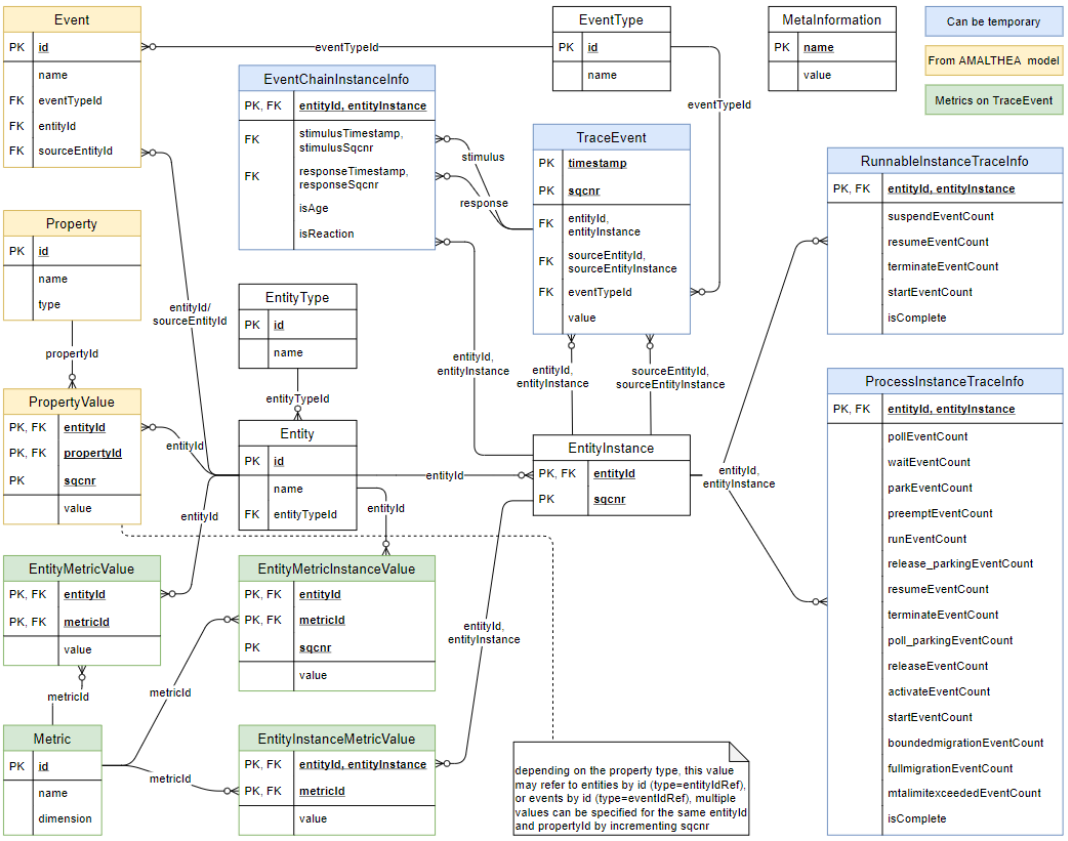

Figure 18 AMALTHEA Trace Database, retrieved from (Eclipse APP4MC, 2020)

4.2.1 Core Tables

The core tables are the foundation of ATDB (Eclipse APP4MC, 2020). Without them the trace would not have any meaning:

- **Metainformation:** Simple key-value pair table used only to identify the trace, for example, specifying the name, or the input model.
- **Entity:** Identifies the entities of the system. Every runtime event is always associated with entities.
- **Entity Type:** Simple table used only to classify an entity by its type, for example, as a scheduler, or a task.
- **Entity Instance:** Identifies the instance of an entity that may execute multiple times, such a periodic process or a runnable.
- **Event Type:** Since ATDB traces are based on events, event types must be defined and referenced by other tables, identifying their occurrence, and allowing the monitorization of the entity's state during its execution time.

4.2.2 Auxiliary Data Tables

Auxiliary tables are provided to specify information related to AMALTHEA models (Eclipse APP4MC, 2020), via the following tables:

- **Property:** This table specifies property types that may be relevant to describe the system's entities, such as the frequency of a CPU.
- **Property Value:** Property values consist of the association of the property to an actual entity, for example, a frequency of 200MHz related to a specific CPU.
- **Event:** Since the AMALTHEA System Model allows event specification, the ATDB also allows their storage. An event must have a specific type, an entity that emits the event, and a target entity.

4.2.3 Metric Tables

This set of tables specify metrics calculated through the results provided by the traced events (Eclipse APP4MC, 2020):

- **Metric:** Specifies the type of a metric, based on its name and dimension.
- **Entity Metric Value:** Provides a metric value for a given identity.
- **Entity Instance Metric Value:** Equally to the entity metric value it specifies the value of a metric. However, it is associated with a specific entity instance, referencing the entity instance table.
- **Entity Metric Instance Value:** This table has the same purpose as the entity instance metric value. However, it refers to instances that are not registered on the entity instance table, identified by the **sqnr**, a sequential integer that serves as an automatic counter of entity instances.

4.2.4 Optional Tables

Optional tables can exist temporarily on AMALTHEA Trace Databases (Eclipse APP4MC, 2020). These tables provide valuable information for metric calculation, representing the trace events raw data. Consequently, metrics calculated using information from optional tables do not reference them, allowing the further elimination of such data and reducing the size of the database:

- **Trace Event:** Stores events from a BTF trace provided by simulation or execution mechanisms.
- **Runnable Instance Trace Info:** Counts the number of events of a runnable instance per event type, in order to identify if the instance's lifecycle is completely covered on the trace.
- **Process Instance Trace Info:** Similar to the runnable instance trace info. However, it is applied to task instances instead of runnables.
- **Event Chain Instance Info:** Summarizes information about the event chain of a certain entity instance.

4.3 Use Case Definition

Use cases (UC) can be considered as the specification of system functionalities from the user perspective, being commonly used to identify software requirements. Typically, UCs are described in a simple way, defining the required system behaviour, and the interactions with external entities, in order to fulfil the demands of the stakeholders (Larman, 2002).

This project aims the support of energy management in AMALTHEA models through energy properties annotations, provided by the analysis of simulation or execution traces. To achieve this goal, the system must support several functionalities. However, most of its operations do not require any sort of manual or external interaction. Therefore, only one use case was identified:

With a trace collected from the execution or simulation of a system specified in an AMALTHEA Model. An external entity must be able to trigger the solution to annotate the energy properties of the system's execution on the respective AMALTHEA specification. An analysis of system traces must be delivered, gathering input hardware resource consumption metrics to perform the energy consumption estimation of the system under study. Finally, the estimated energy properties must be annotated in the AMALTHEA System Model, supporting further energy management in a self-contained way.

After the external trigger activates the system, its internal tasks must operate automatically. Besides, the model annotations should present not only the estimated energy values, but also some basic statistics, the measurement unit, and a reference to the targeted software element.

Primary Actors:

A user who wants to manually trigger the system, or an external mechanism in case an automatic activation is needed.

Preconditions:

The AMALTHEA Model of the system must be previously specified. A simulation or execution trace must be provided in case an offline analysis is desired. Otherwise, a mechanism must be collecting and supplying the system with execution traces, while the analysis is performed.

Post Conditions:

The AMALTHEA model under analysis must become energy aware through the energy properties annotations provided by the solution, supporting further analysis and management.

Basic Flow

1. The Actor triggers the Properties Annotation Service
2. The System analyses the trace and calculates the hardware resource usage input metrics
3. The System estimates the energy consumption
4. The System annotates the estimated properties in the AMALTHEA model
5. The System returns the annotated model

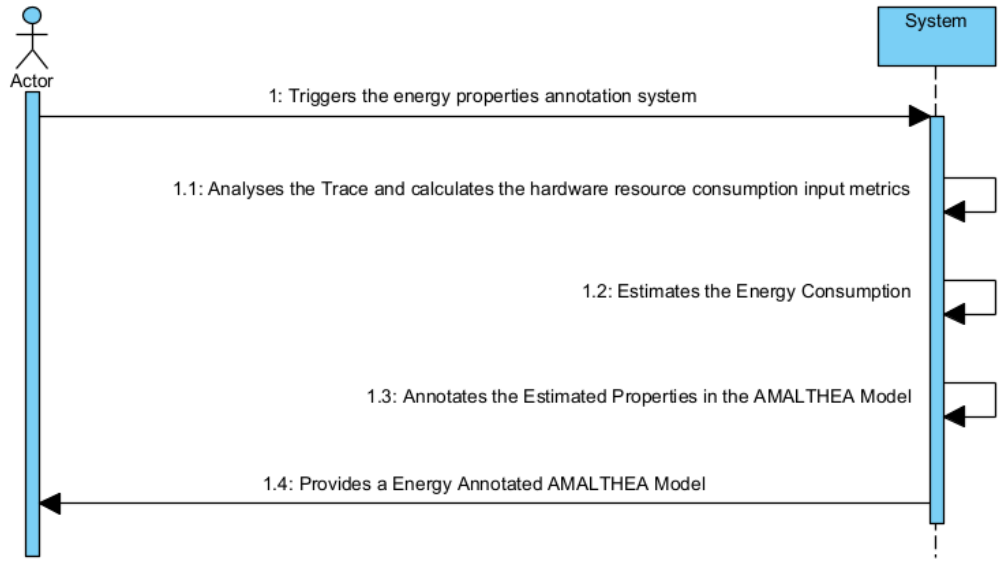

Figure 19 System Sequence Diagram of the Energy properties Estimation and Annotation Use Case

4.4 Requirement Analysis

When the Requirement Analysis is carried out, a set of functional and non-functional requirements is defined, influencing the next steps of the development process, and allowing the design and implementation stages to provide a solution that presents the expected features.

To describe and gather the solution's requirements, the FURPS+ model was adopted. It consists of a software requirements model created by Robert Grady, dividing the requirements into a set of different quality factors, **Functionality**, **Usability**, **Reliability**, **Performance**, **Supportability**, and the **"+"**, which was included later to comprise different types of attributes (Marinho & Ferreira Resende, 2012). Categories that are not comprised by the solution were not addressed in this section.

Finally, the solution requirements are based in the use case described before, along with the state of the art study. It considers the main goals, requirements, and concerns of the automotive industry, along with the aspirations and core aspects of the AMALTHEA framework.

4.4.1 Functionality

The functionality component specifies the product's intended behaviour, considering aspects such as Auditing, Licencing, Security, System Management, and the Workflow (Eeles, 2001), which define system-wide quality attributes that are relevant to the architecture (Larman, 2002).

REQ1 –Trace and System Models Analysis

The software's energy consumption is estimated through a set of input properties, such as the hardware resource usage information of the system (Singh, et al., 2013; Colmant, et al., 2019). To allow energy metrics estimation, the solution must perform an analysis of AMALTHEA Traces and System Models, in order to infer the necessary energy estimation input metrics.

REQ2 – Energy Estimation Mechanism

The system must be able to perform the estimation of the energy consumption of the software under analysis at a runnable granularity.

REQ3 – Energy Properties data Analysis

To simplify energy analysis and management, the energy metrics produced by the system must be treated in order to provide some basic statistical information, easing their interpretation.

REQ4 – Model Annotations of the obtained information

To persist the obtained energy properties, allowing further analysis and management of the system specified under the AMALTHEA framework. This solution must provide a model annotation mechanism to store the information defined in **REQ3**.

REQ5 – Automatic Execution of the system

During the whole process, from the system and trace analysis to the model annotations, there is no need to perform manual operations. On the other hand, this work proposes a solution compatible with dynamic and self-adaptive systems using the AMALTHEA framework, which aims a seamless software development lifecycle. Consequently, the automatic execution of the system tasks is mandatory.

4.4.2 Reliability

Reliability is the system's ability to operate in a trustworthy manner. It is reflected in attributes such as availability, the accuracy of calculation and estimation mechanisms, and failure recovery (Eeles, 2001).

REQ6 – Power Estimation Mechanism Accuracy

The adopted energy estimation mechanism must present an accuracy equal or higher than 90%.

4.4.3 Performance

Performance requirements describe how the system executes, considering features such as the response time, recovery time, start-up time and shutdown time (Eeles, 2001).

REQ7 – System's Response Time

The execution of the entire process, from the trace analysis to the model annotations, must take less time than the duration of the trace under analysis. This solution aims not only to perform offline analysis, but also to be applicable as an online analysis mechanism. Thus, it is important to deliver a solution presenting acceptable response times.

4.4.4 Supportability

Supportability is a quality factor related to the ability of managing the system over time, taking into account aspects such as adaptability, maintainability, configurability, and scalability (Eeles, 2001).

REQ8 – The solution must be extensible

The AMALTHEA framework is highly extensible, providing flexibility to the development team. Although it usually provides standard mechanisms, there is always the possibility of tailoring the platform by integrating it with external tools. Therefore, since there are multiple approaches and formats to ensure the system traceability, along with different techniques to assess energy metrics. Such mechanisms must be easily extensible and replaceable, affecting as less as possible the other components of the system.

REQ9 – The solution must provide deployment freedom

Automotive software is usually composed of distributed computing units, establishing communication between each other or external entities. In addition, one of the main goals of AMALTHEA is to provide flexibility. In this way, this approach must be flexible and compatible with modern solutions, providing deployment freedom and allowing the distribution of its components.

REQ10 – Reusability of the Annotation Mechanism

The annotation mechanism consists of a considerably generic feature. Energy metrics consist only of numeric values associated with a unit, something that is also verified in many other metrics from various domains. Consequently, the annotation mechanism must be potentially extended and used to annotate various types of metrics simultaneously.

4.4.5 Other Requirements (+)

The "+" requirement category comprises quality attributes that are not included in the other FURPS categories. It considers design and implementation constraints, interface requirements that specify the interaction with external entities, and physical requirements related to the adopted hardware platform (Eeles, 2001).

REQ11 – Adoption of the SQLite relational database management system

Accesses to AMALTHEA Trace Databases must consider it as an SQLite Database.

REQ12 – Adoption of Java 8 to perform model manipulations

AMALTHEA System Model manipulations using the Java programming language require the adoption of, at least, the version 8. Although the adoption of Java is not mandatory, it simplifies model manipulations since it is specified with EMF, which provides a set of java classes to manipulate the models.

REQ13 – The model annotations must be performed in AMALHTEA System Models

Since AMALTHEA is a flexible platform, the use of the ATDB is not mandatory. Other trace formats may be easily adopted according to the development team needs. However, the use of the AMALTHEA System Model is essential. So, the energy information provided by the

solution must be annotated in AMALTHA System Models, providing a broader and self-contained solution.

5 Architecture and Design

With the analysis concluded and the requirements gathered, the features that must be delivered by the solution are defined.

This chapter focuses on the next development stage of the project, describing the adopted architecture and exploring possible alternatives. Afterwards, the specification of the system design and architecture is portrayed according to the "4+1" view model (Kruchten, 1995), which is composed of five different elements. The **Logical View**, the **Implementation View**, the **Process View**, the **Deployment View**, and the **Scenarios View**, which was excluded since it provides information related to the use cases, already specified in the Use Case Definition of the solution analysis.

5.1 Architecture

The term software architecture is usually associated with a most abstract definition of the system, considering coarse-grained and top-level decisions. It provides a transition between the analysis and further stages, defining how the solution can satisfy its requirements. In this way, it helps the development team to understand, analyse and implement the system, while promoting its management, reuse, and evolution (Garlan, 2000).

5.1.1 Adopted Architecture

The Adaptive AUTOSAR architectural approach is based in a Service Oriented Architecture (SOA) (AUTOSAR, 2020b). It consists of an architectural style where the system is composed of a set of services, each one responsible for a certain task, and consumed by other services or external entities through a specific communication mechanism. SOA is highly suitable with distributed systems, promoting scalability, flexibility in resource allocation, and parallel processing.

In this manner, considering the Adaptive platform, the AMALTHEA framework, and the solution requirements, the adopted architecture presents several similarities to SOA, being based in the microservices architectural style (IBM Cloud Team, 2014). The Service Oriented Architecture differs from the microservices architecture since it presents an enterprise scope, while the other presents an application scope. Accordingly, the microservices architecture focuses on decomposing one application responsible for a business function into various services. SOA focuses on the coexistence of various business functions (Outsystems, 2021).

Microservice applications are composed of a set of small and loosely coupled services responsible for a specific process, and just like in SOA, communicating between each other

through well-defined communication interfaces. Each service can be implemented with different technologies, and independently deployed (Microsoft, 2019).

As a result, a system that follows this architecture allows the management and extension of a given service without affecting the other components, also promoting simplified testing, reusability, and fault isolation.

Since the solution is based in this architecture, it represents each major capability decomposed into a single service, mainly to allow the adoption of different technologies, distributed deployments, reusability, and improved extensibility. To perform the integration between the system services, the choreography pattern was adopted (Microsoft, 2020), following a similar logic to the one illustrated in Figure 20.

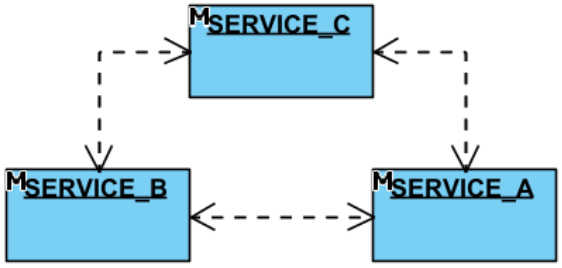

Figure 20 Service Choreography

With service choreography, the system operates properly without relying on a centralized mechanism to control the workflow and establish the interaction between services. Instead, when a service performs a task, it decides how the rest of the operation is processed, transmitting information directly to other services, usually through asynchronous messaging.

This pattern promotes the solution's extensibility, allowing changes to a service without highly affecting others (Microsoft, 2020). Therefore, choreography was adopted since one of the main objectives of AMALTHEA is to be extensible and flexible, allowing heterogenous development teams to use the solution according to their needs, and being able to adapt it easily.

5.1.2 Alternative Architectures

Two main architectural alternatives where identified. In the first place, the system could be developed according to a similar architecture, yet, using centralized service integration mechanisms rather than the choreography pattern. On another way, instead of modularizing the system into a set of services, a monolithic approach could also be considered.

Service Integration through Orchestration Mechanisms

Adopting a similar approach to the one described before, where the system is decomposed into a set of independent services. The integration of each component can be performed in a

centralized way. Thus, a service orchestrator would be responsible for receiving the inputs of the system, decide which services must be executed, and in which order they should perform their operations (Microsoft, 2020). An example of the adoption of a service orchestrator is illustrated in Figure 21.

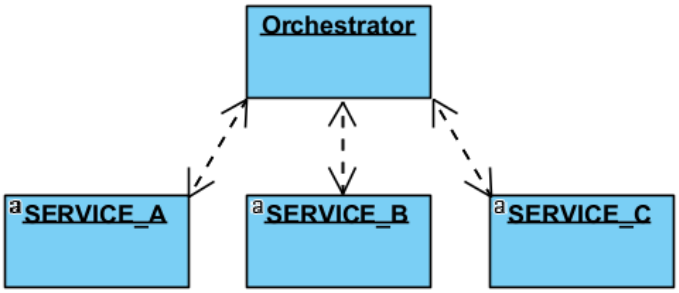

Figure 21 Service Orchestration

Service orchestration is highly suitable for the management of large systems composed by several services and numerous transactions, providing a simplified and resilient workflow management mechanism. However, this pattern increases system coupling, since the orchestrator must be aware of the business logic and the services it controls. Besides, when compared to a choreography, it affects the system performance negatively, also representing a single point of failure (Microsoft, 2020).

Since the solution aims to provide high extensibility, updates and changes may be performed. Yet, it is unlikely to include new workflows, excluding easily manageable transactions, or the replication of already existing services instantiated through a distributed network. Likewise, the already existing workflow described in the Use Case Definition is simple, not justifying the efforts and performance loss related to the implementation of a service orchestrator.

Monolithic Application

The last alternative to the solution's architecture, shown in Figure 22, proposes a drastic change to the system. It consists of adopting a monolithic approach, where instead of implementing a system as a set of services, it consists of a single one containing all the desired functionalities.

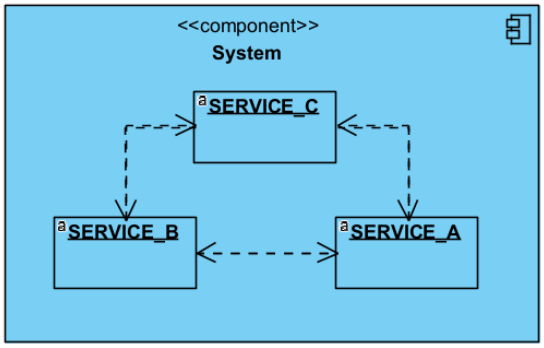

Figure 22 Monolithic Approach

Developing a system as a monolith simplifies the implementation of the transactions and the interaction between entities, since it consists of a single application (Outsystems, 2021b) and avoids remote communications.

However, considering the requirement analysis, this type of approach would not be the most suitable one. It is important to provide a flexible and extensible system, with deployment freedom, and allowing the reusability of the implemented services. Updates to a monolithic system tend to be difficult, due to huge code complexity and deployment times. Additionally, since the system is contained inside the same application, this approach lacks from reusability, the ability of performing distributed and independent deployments (Outsystems, 2021b), and the potential adoption of heterogeneous technologies.

5.2 Implementation View

The implementation view represents the system decomposition, focusing on its modular structure, and considering its components and relationships (Kruchten, 1995). In this project, this view was the first element of the 4+1 architectural model to be analysed since it reflects the main features of the adopted architecture.

As illustrated in Figure 23, to represent the implementation view of the solution, a UML component diagram was adopted.

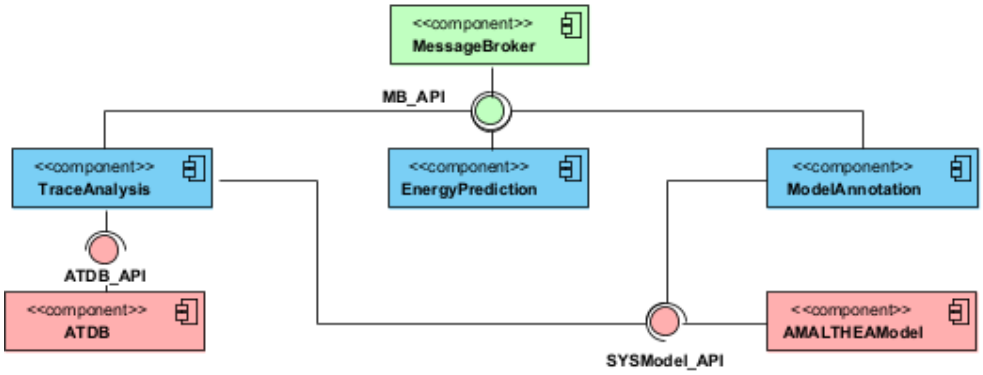

Figure 23 Component Diagram of the Solution

The data sources of the system are represented in pink, being composed of the AMALTHEA Trace Database and the AMALTHEA System Model. The ATDB stores traces with the information about the system execution or simulation, which is used to calculate the hardware resource usage metrics that serve as an input to perform the energy metrics estimation. The AMALTHEA System model contains relevant details about the system specification. Additionally, the energy properties estimated by the solution are annotated in this model.

These data sources are accessed by part of the components represented in blue, the **TraceAnalysis** service, the **EnergyPrediction** service, and the **ModelAnnotation** service.

The **TraceAnalysis** is responsible for the calculation of the input metrics used to perform the energy estimation. This component analyses AMALTHEA traces with information about the system's execution or simulation. On another way, it also examines relevant information about the system specification contained in AMALTHEA system models, especially in simulation scenarios.

Although the ATDB is considered the standard trace database in AMALTHEA, various trace formats and mechanisms can be used. Thus, it becomes important to deliver a modular solution, isolating this functionality as a single component, and allowing changes without unnecessarily influencing other components.

The **EnergyPrediction** service is responsible for the energy metrics estimation process. Several mechanisms can be used to perform this task, and the adoption of specific methods and technologies may be necessary, such as suitable programming languages, libraries, or machine learning algorithms. Therefore, to avoid direct interactions with AMALTHEA models, which are usually manipulated with Java classes, a single component is proposed to perform this task. Additionally, since the adopted architecture provides deployment freedom, the isolation of this service allows it to be instantiated only once and interact with several **ModelAnnotation** and **TraceAnalysis** instances simultaneously.

The **ModelAnnotation** service is responsible for annotating the estimated energy metrics in AMALTHEA models, after performing a simple statistical analysis targeting the received data. The statistical analysis was included in this service since it influences the annotation structure. On the other hand, it consists of a simple function, which does not justify the implementation of an isolated component to fulfil that requirement.

Usually, the ATDB is the adopted mechanism to store information about software traces and executions. It would be acceptable to store the energy metrics in the trace model, instead of performing system model annotations. However, since the use of the ATDB is not mandatory, the approach proposes metric annotations in system models to promote a self-contained and flexible solution.

Finally, the **ModelAnnotation** component is considerably generic. It can ignore the nature of the annotated properties since energy metrics are represented by a numeric value and a

measurement unit. In this way, the representation of the model annotations as an isolated service is beneficial. It allows the simultaneous interaction of a single instance with different services responsible for the calculation of other metrics.

To establish the communication between the components of the solution, the adoption of a message broker is proposed, operating in a Publish/Subscribe manner, and represented in green in the diagram. Similar paradigms were already implemented within the scope of Adaptive AUTOSAR, and it is suitable with the approach and the adopted architecture. The message broker handles the communication between different services, reducing the dependencies between them. Besides that, it allows asynchronous communication, without the need of having the publishers and the subscribers available at the same time (Eugster, et al., 2003).

5.3 Logical View

The Logical View focuses on the object-oriented decomposition of the system, defining which elements must be provided by the solution to support its functional requirements. A class diagram was adopted to represent this view, defining the implemented classes and their logical relationships (Kruchten, 1995), as illustrated in Figure 24.

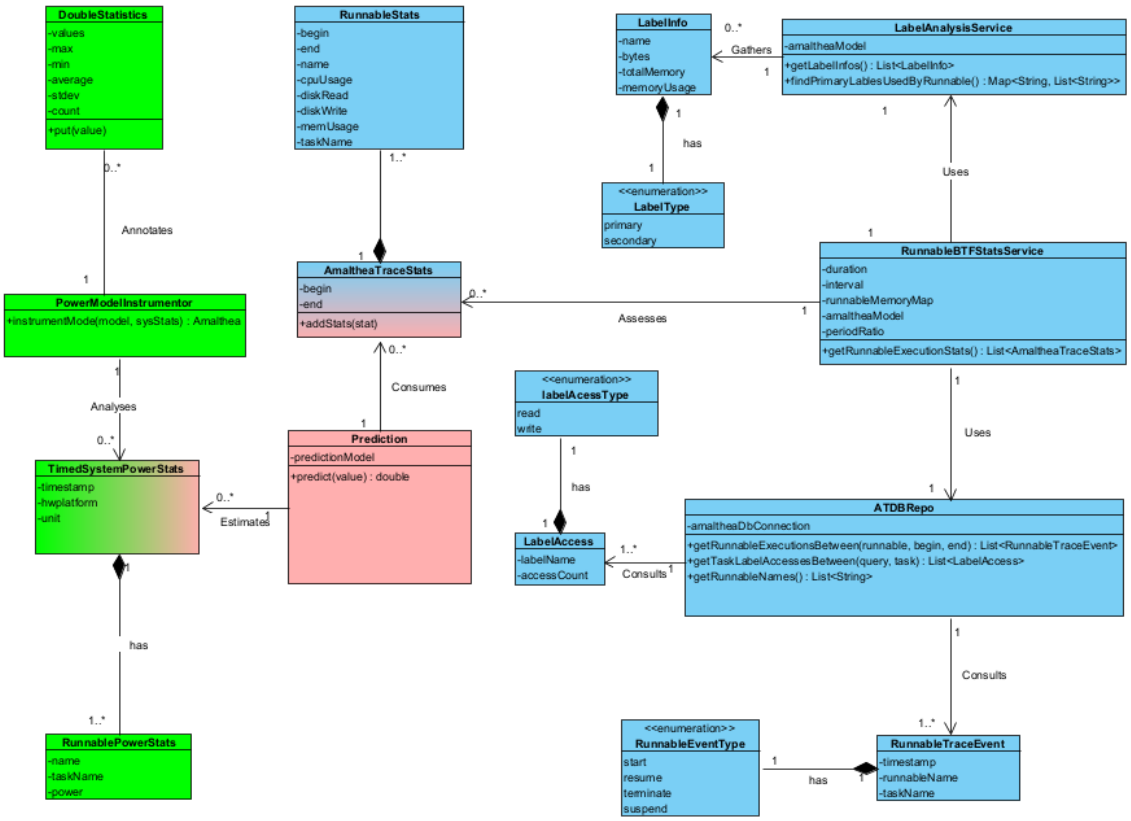

Figure 24 Class Diagram of the Solution

This approach proposes a system composed of three independent services. The classes belonging to the **TraceAnalysis** service are represented in blue, classes from the **EnergyPrediction** service are represented in pink, and the classes from the **ModelAnnotation** service are represented in green.

RunnableBTFStatsService:

This class is responsible for the assessment of the hardware resource usage metrics for each runnable of the system. These metrics are calculated with information from the BTF traces stored in the ATDB, which is gathered using the **ATDBRepo** class. Additionally, it uses the **LabelAnalysisService** to consult memory information stored in AMALTHEA system models.

ATDBRepo:

The **ATDBRepo** can be used to query the AMALTHEA Trace Database. This class retrieves the BTF events that are stored in the ATDB, which consist of **RunnableTraceEvent** or **LabelAccess** objects. Besides, it can be used to consult the runnable entities that were executed in the trace under analysis.

LabelAnalysisService:

This class is responsible for consulting relevant information stored in AMALTHEA models. As the name suggests, the analysed information is related to the AMALTHEA labels, returning **LabelInfo** objects, and the primary type labels that are used by each runnable of the system.

LabelInfo:

The **LabelInfo** presents relevant information about AMALTHEA Labels that is used by the solution to calculate memory-related metrics.

Since a Label can be associated with a primary or a secondary memory, the **LabelType** enumeration is used to define it.

LabelAccess:

This class represents BTF trace events that specify operations related to AMALTHEA Labels. In this way, it becomes possible to know when a certain entity performs an access to a memory label, supporting the hardware resource usage metrics calculation.

A **LabelAccess** event can consist of a read or write operation, which is identified by the **LabelAccessType** enumeration.

RunnableTraceEvent:

Resembling the **LabelAccess**, this class represents BTF events, yet, related to the execution of runnable entities. This event provides timestamped information about their state, recording when their execution starts, suspends, resumes, or finishes, which can be identified through the **RunnableEventType** enumeration.

RunnableStats:

The **RunnableStats** class stores the metrics calculated by the **TraceAnalysis** service for a particular runnable of the system. Each **RunnableStats** object is associated with a specific period of the trace under analysis, along with runnable's parent task.

AmaltheaTraceStats:

The **AmaltheaTraceStats** has the purpose of encapsulating a set of **RunnableStats** for a given timestamped period.

Prediction:

The **Prediction** class belongs to the **EnergyPrediction** service. It is responsible for the energy estimation process, which is performed using **RunnableStats** entities as input, and retrieving **RunnablePowerStats** entities as output.

PowerModelInstrumentor:

This class is responsible for the interpretation of the data contained in the **RunnablePowerStats** entities, provided by the **EnergyPrediction** service. It treats the received data, and performs a basic statistical analysis, creating **DoubleStatistics** entities and annotating the obtained information in AMALTHEA models.

RunnablePowerStats:

The **RunnablePowerStats** consists of the estimated power consumption of a Runnable for a given trace period.

TimedSystemPowerStats:

This class encapsulates a set of **RunnablePowerStats** instances belonging to the same trace period.

DoubleStatistics:

The **DoubleStatistcs** class is responsible for providing the results of the statistical analysis performed by the **ModelAnnotation** service. It contains a list of double values, each one representing the power consumption of a runnable entity for a particular period. Every time a new element is added to the list, this class independently ensures the calculation of the minimum and maximum values, the average, the standard deviation, and the number of entries.

5.4 Process View

The Process View of the 4+1 model not only reflects the functional requirements of the system but also certain non-functional properties, considering aspects such as distribution,

concurrency, synchronization, and the execution of the Logical View as a set of operations (Kruchten, 1995), considering their behaviour and the message flow.

In this project, the UML sequence diagram from Figure 25 was adopted to represent the Process View. In the first place, a coarse-grained perspective of the solution's use case is specified, dividing it into a set of major tasks and considering the components illustrated in the Implementation View. Finally, a component level analysis of the Process View is provided, also using sequence diagrams, and considering the elements from the Logical View.

For simplification purposes, some operations were omitted or illustrated in a simplified way, such as the interactions with AMALTHEA models, which are already supported by APP4MC and not implemented by the project. In this way, only the most relevant methods and elements are described.

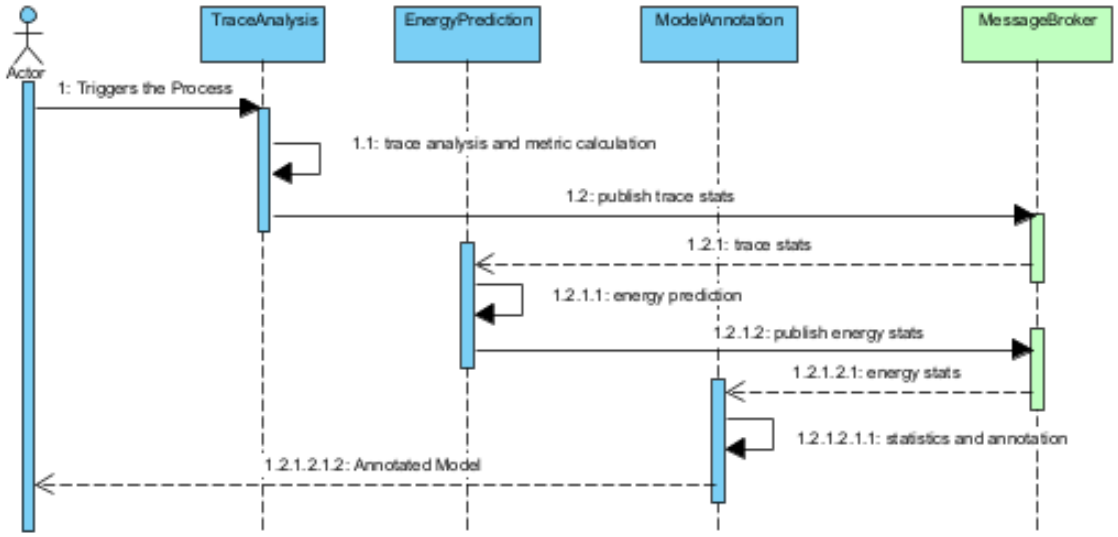

Figure 25 Coarse-Grained Sequence Diagram of the Solution's Use Case

Initially, when the system is triggered, the **TraceAnalysis** service performs the first operation, calculating the hardware resource usage metrics of the system's runnables through the analysis of execution or simulation traces. Meanwhile, when the metric calculation is finished, this service publishes the results in the message broker.

Afterwards, the **EnergyPrediction** service subscribes the message queue with the information that results from the trace analysis. In this way, it consumes the received data and performs the energy metrics estimation, publishing the results in the message broker again.

Finally, the predicted properties that are stored in the message broker are subscribed by the **ModelAnnotation** service. When this service receives the energy data, it performs the statistical analysis and annotates the energy information in AMALTHEA models.

5.4.1 Trace Analysis

The Trace Analysis process can be considered has the one relying on more computational efforts, due to the fact that it performs an analysis of each runnable executed by the system numerous times, considering different perspectives, their life-cycle, and state changes.

The operations provided by the **TraceAnalysis** service are represented in the sequence diagram illustrated in Figure 26.

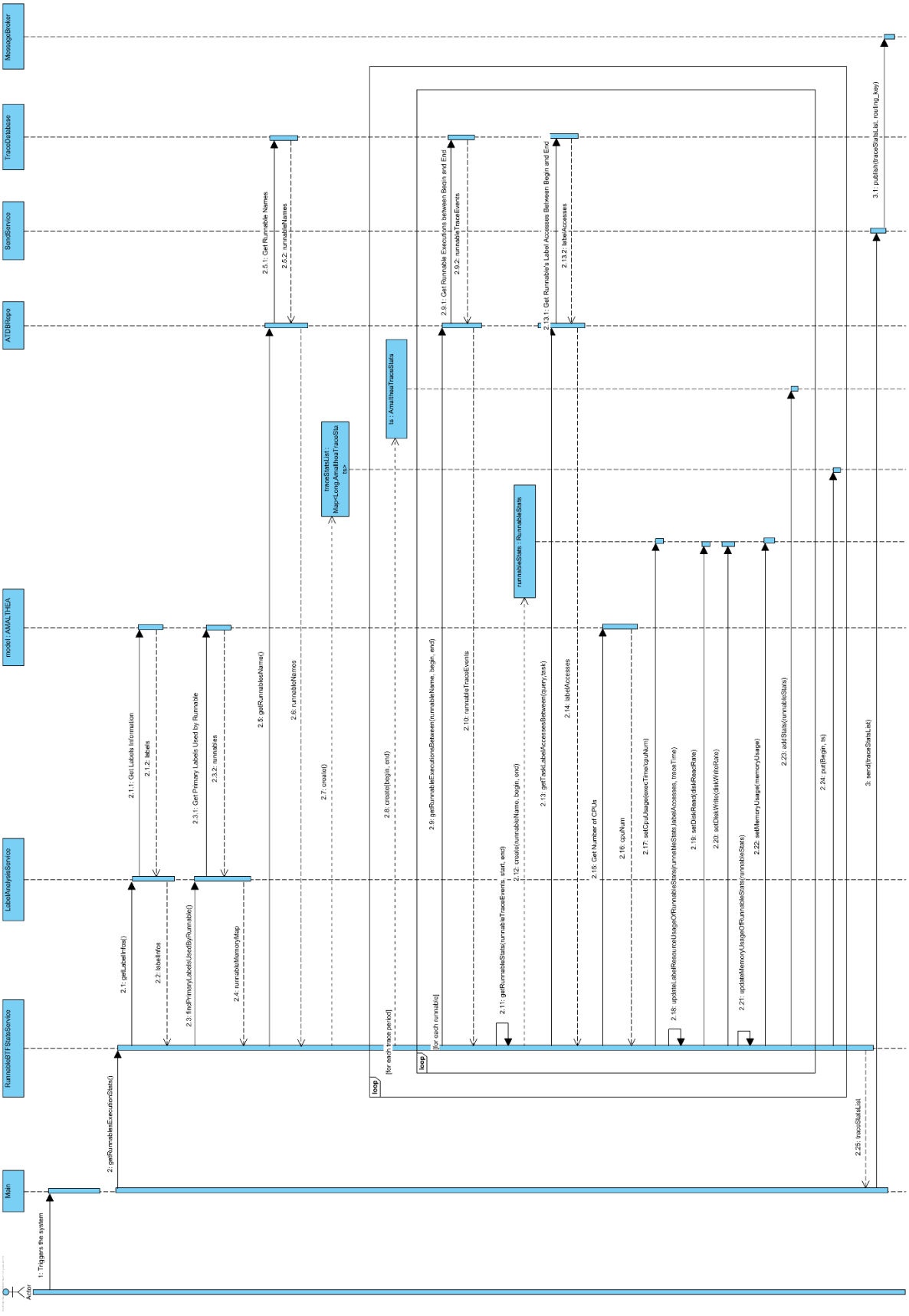

Figure 26 Sequence Diagram of the Trace Analysis operation

When the system is triggered, the **TraceAnalysis** service starts the analysis of the collected trace by calling the **getRunnableExecutionStats()** method from the **RunnableBTFStatsService** class.

In the first place, this method uses the **LabelAnalysisService** to collect the **LabelInfo** entities from the AMALTHEA model under analysis, and a map associating each runnable entity with the primary type labels that it uses during its execution. Some computational power is necessary to obtain such information. Therefore, since it is used multiple times throughout the entire process, it becomes beneficial to eagerly perform these operations.

With the labels information collected, the **RunnableBTFStatsService** uses the **ATDBRepo** class to get the names of the runnables executed by the system. Then, the analysis of the AMALTHEA trace begins, dividing it into a configurable set of time spans, and creating an **AmaltheaTraceStats** entity for every one of them.

For each trace interval, the metric calculation is performed for each executed runnable. Initially, the **RunnableTraceEvent** entities are collected using the **ATDBRepo** class. Then, for the obtained runnable trace events, and the period under analysis, the **getRunnableStats()** method its executed. First, a **RunnableStats** entity is created for the period under analysis. After that, the **LabelAccess** entities referring to the runnable under analysis are collected. Therefore, it becomes possible to know which events targeting a certain runnable occurred, along with the memory labels that it accessed.

Later, using the events gathered from the trace database, the CPU usage of the runnable entity is verified through the collected **RunnableTraceEvent** objects, which provide means to calculate the execution time of a runnable through its state changes. Additionally, the **LabelAccess** events and the memory information gathered before, using the **LabelAnalysisService**, provides the information needed to perform the hardware resource usage calculation regarding the primary memory and the disk metrics employed for the estimation of energy metrics.

Finally, the **RunnableStats** entities resulting from the analysis of the runnable execution are added to the corresponding **AmaltheaTraceStats** entity, and the process is repeated until all the trace time intervals are fully analysed. In the end, the system provides a set of **AmaltheaTraceStats** entities containing the information needed to estimate the energy consumption of the system's execution at the runnable level. This information is published in the message broker, in order to allow the other components of the solution to subscribe it and proceed with their operations.

5.4.2 Energy Prediction

After the input hardware resource usage metrics are calculated, the **EnergyPrediction** service is responsible for the energy estimation process. This task is relatively simple, and it is represented in the following diagram.

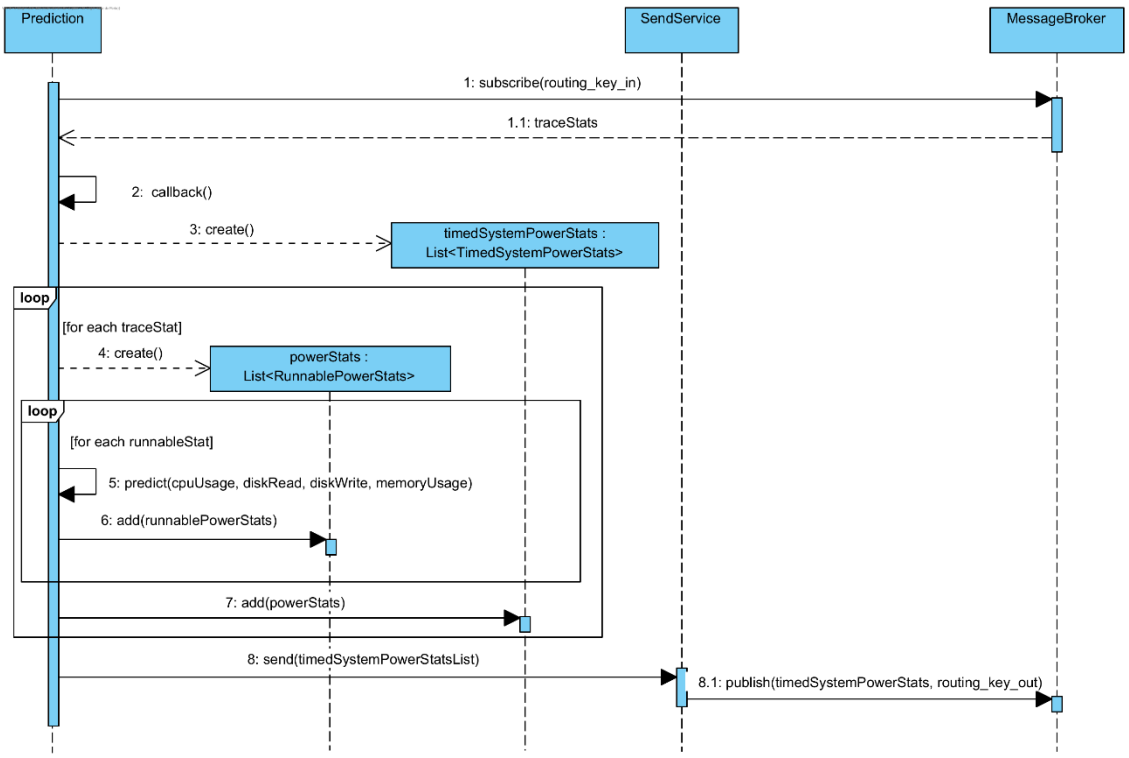

Figure 27 Energy Prediction Sequence Diagram

The **EnergyPrediction** service subscribes the message broker to receive messages from the **TraceAnalysis**. When data arrives to the queue, it comes in the form of an **AmaltheaTraceStats** list, and each one of its elements is individually analysed. In this way, the assessment of the energy consumption is performed for every runnable executed by the system, and for each analysed trace period, through the information provided by the **RunnableStats** entities contained on each **AmaltheaTraceStats**.

Finally, the predicted energy metrics are stored in a set of **TimedSystemPowerStats** objects, each one of them composed of a set of **RunnablePowerStats** entities referring to the analysed runnables, in a similar way to the received input. Once again, the obtained data is published in the message broker to ensure the remaining operations.

5.4.3 Model Annotations

The final operation of the process is performed by the **ModelAnnotation** service, represented in the sequence diagram from Figure 28.

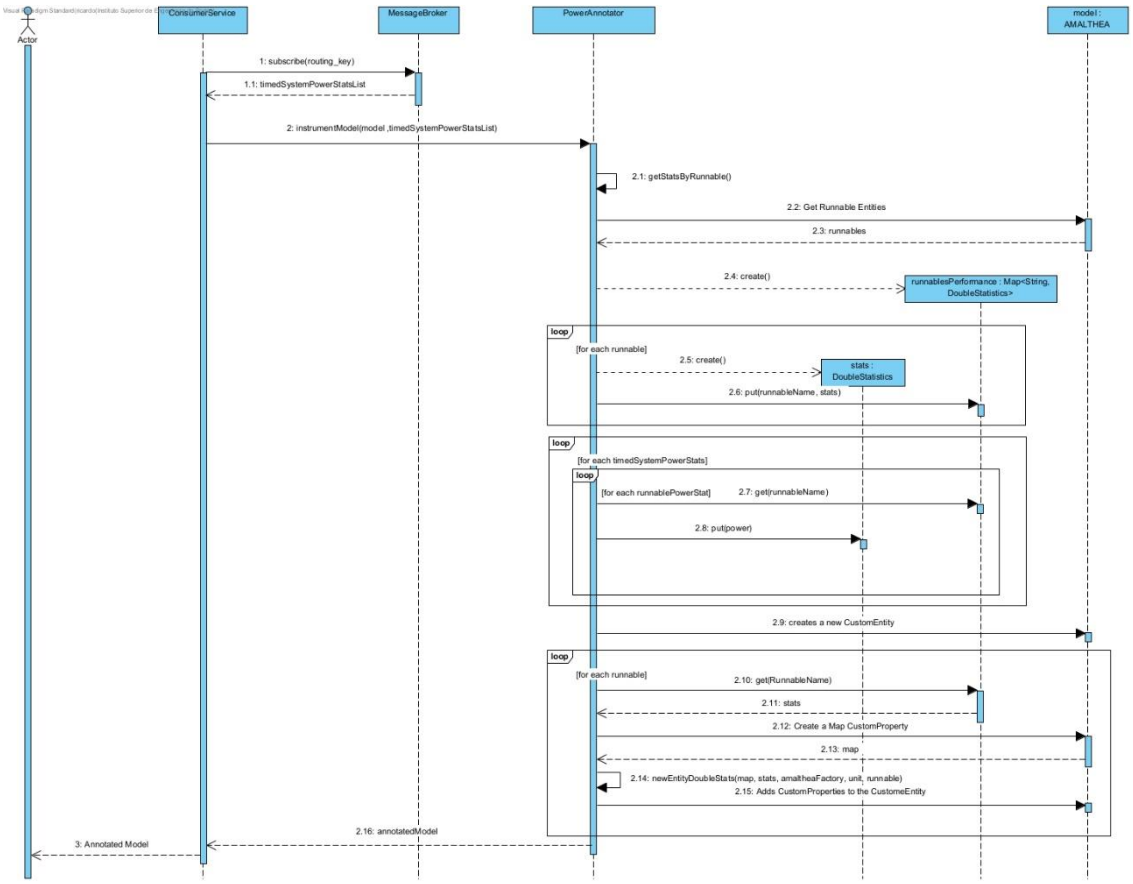

Figure 28 Sequence Diagram of the model annotation process

When energy properties are received from the **EnergyPrediction** service through the message broker, the **ModelAnnotation** executes the **instrumentModel()** function, which is responsible for providing the model annotations.

In the first place, since the trace information is grouped by a timestamp, the system consults the runnable names and creates the **runnablesPerformace** map, using the runnable's name as key and a **DoubleStatistics** object as value. This object stores the statistical information of the runnable's energy consumption, and every time a value is added, it handles the statistical analysis in an autonomous way.

Finally, for each **RunnablePowerStats** entity of every **TimedSystemPowerStats** element, the power consumption information per runnable is appended and analysed on its corresponding **DoubleStatistics** object of the map.

Once all the information is structured in the map, it is annotated in the form of CustomProperties in the AMALTHEA model of the system under analysis, presenting a similar structure to the **DoubleStatistics** class, but also including a reference to the target runnable, and the metric's measurement unit.

5.5 Deployment View

According to the 4+1 model, the Deployment View represents the physical configuration of the system, considering the allocation of software into the hardware infrastructure. To represent the deployment configuration of the system, an UML Deployment Diagram was used, as illustrated in Figure 29.

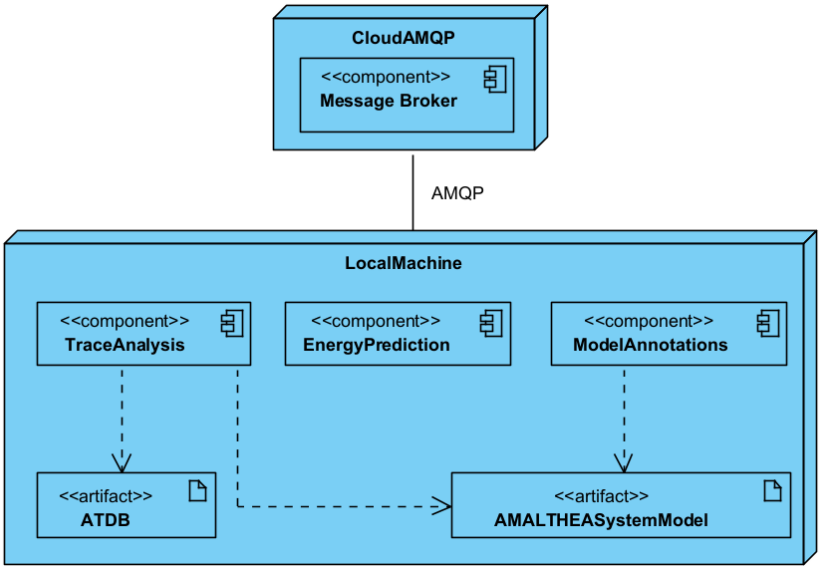

Figure 29 Deployment Diagram of the Solution

Since the implementation of the approach, in this project, is only applied to simulation traces, instead of the execution of a system. There is no need of providing any kind of distribution. Every component was deployed locally in the same hardware device, except for the message broker, which, for simplification proposes, was installed in the cloud using the CloudAMQP platform, in order to avoid unnecessary setup and configuration efforts (CloudAMQP, 2021).

6 Experimental Implementation

After the conclusion of the analysis and design stages, it becomes possible to conduct the development of a solution to the problem at hands.

This chapter describes an experimental implementation of the approach focusing on the development environment, the adopted communication mechanisms, the interaction with the elements from the AMALTHEA framework, and the most important aspects related to the implementation of the trace analysis, energy prediction, and model annotation processes. Construction decisions are explained, evidencing the instantiation of the system according to the adopted design and architecture, fulfilling the demanded requirements.

6.1 Development Environment and Conventions

This work is part of a greater project that relies on the implementation of different solutions and components to achieve various goals, sharing artifacts between them and being developed by more than one person. Consequently, a GitLab repository (GitLab, 2021) was set up to publish the solution, supporting the version control of the project, and making it easily accessible among the team members and stakeholders.

Since the development team and each component of the project present a reduced dimension, concurrent changes to the same file or functionality are rare. The application of branching techniques was only considered necessary to create different versions or extensions of a component.

Finally, for this experiment, the Nvidia Jetson AGX Xavier (Nvidia, 2021b) was the adopted hardware platform. It is composed of an 8-core ARM v8.2 64-bit CPU, a 32GB 256-bit LPDDR4x primary memory, and 32GB of disk capacity, executing under the Ubuntu operating system. Any hardware dependent feature of the solution, such as the energy estimation mechanism and the adopted AMALTHEA models, was configured considering this machine and its characteristics as the hardware platform of the system.

6.2 Communication Mechanism

RabbitMQ was adopted to establish the communication between the components of the solution (RabbitMQ, 2021). It consists of an open-source message broker that delivers low latency and high throughput messaging (Dobbelaere & Esmaili, 2011), supporting multiple protocols and programming languages. Whitin the scope of the project, RabbitMQ was configured to operate according to the Advanced Message Queue Protocol (AQMP), as illustrated in Figure 30.

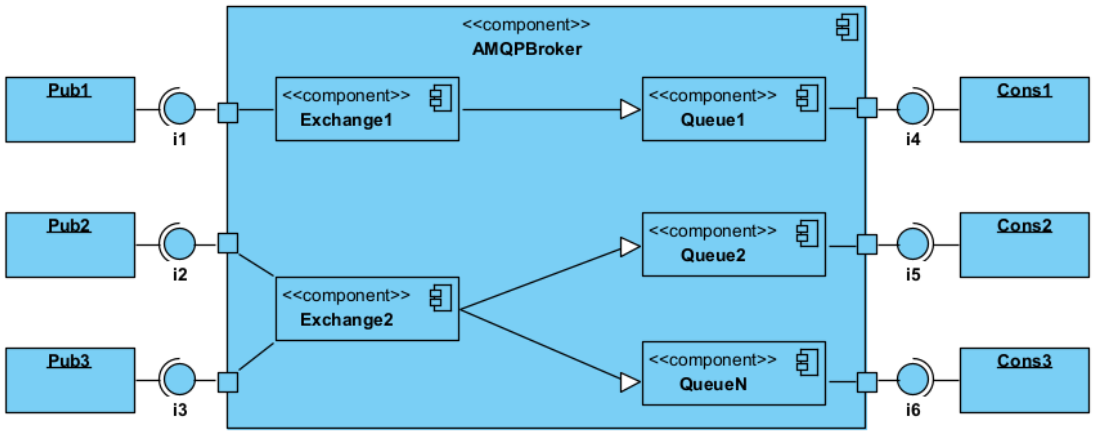

Figure 30 Example of an AMQP Scheme

Data that arrives to the message broker is stored in an exchange. Subsequently, the exchange copies the received messages and forwards them into one or more queues, each one associated with a single consumer.

The exchange is considered as a routing mechanism that can be configured in various ways, allowing the adoption of customizable routing approaches. In this way, the transmission of messages to a particular set of queues can be delineated by a specific behaviour, defined by the exchange type:

- **Direct Exchange:** Distributes messages using a routing key, associating them with queues that are bound to it.
- **Fanout Exchange:** Connected to a set of message queues, broadcasting the received messages to all of the connected consumers.
- **Topic Exchange:** Routes messages dynamically to one or more queues by matching the routing key to a pattern that associates the queue to the exchange.
- **Headers Exchange:** Operates according to the direct exchange. However, it provides a routing mechanism through header attributes instead of routing keys.

Finally, when AMQP message brokers are implemented, the definition of the rules that an exchange must follow to route messages to a set of queues is performed through the specification of bindings. A binding defines the exchanges that are responsible for routing messages to a certain queue, along with the routing keys that it consumes (RabbitMQ, 2021b), depending on the exchange type.

For this project, RabbitMQ was exclusively employed using the direct exchange type, since it provides a simple mechanism for the selective transmission of messages, allowing the consumer to subscribe only the subset of messages with valuable information to it.

According to the design of the solution, the information flow of the system starts with the **TraceAnalysis** service publishing hardware resource usage metrics to the message broker. Then, the **EnergyPrediction** service subscribes this information and estimates the energy

consumption of the system under analysis. In the end, the obtained data is also published in the broker and consumed by the **ModelAnnotation** service.

As a result, one exchange was created, named **power_queue**. The **TraceAnalysis** service was connected to it, publishing messages with the **power_in** routing key. The **EnergyPrediction** service was connected not only as a consumer of the **power_in** key messages, but also as a publisher, sending messages with a routing key named **power_out**. To consume the messages associated with this routing key, the **ModelAnnotation** service was connected to a message queue and bound to that same exchange, yet only consuming messages from the **EnergyPrediction** service.

The code snippet presented below, illustrates the implementation of the connection between the **EnergyPrediction** service and the message broker:

```
##########################
# Subscriber Connection #
##########################
def setup_connection(params,exchange_name,routing_key):
  connection = pika.BlockingConnection(parameters)
  channel = connection.channel()
  channel.exchange_declare(exchange=exchange_name, exchange_type='direct')
  result = channel.queue_declare(queue='', exclusive=True)
  queue_name = result.method.queue
  channel.queue_bind(exchange_name, queue=queue_name,routing_key=routing_key)

########################
# Publisher Connection #
########################
class sender:
  def __init__(self,params,exchange_name):
    connection = pika.BlockingConnection(parameters)
    channel = connection.channel()
    channel.exchange_declare(exchange=exchange_name,exchange_type='direct')

  # Send Messages
  def send_message(self,exchange_name,send_routing_key,message):
    self.channel.basic_publish(
    exchange=exchange_name,
    routing_key=send_routing_key,
    body=json.dumps(message))

#####################################
# PREDICTION SERVICE IMPLEMENTATION #
#####################################
exchange_name = 'power_queue'
routing_key = 'power_in'
send_routing_key = 'power_out'
params = pika.URLParameters('URL')
send_service = Sender(params,exchange_name)
(channel,queue_name) = setup_connection(params,exchange_name,routing_key)

def callback(ch, method, properties, body):
  # Handles the Received Message
  # Estimates the Consumption (…)
  # Sends message to the broker
  send_service.send_message(exchange_name,send_routing_key,message)
```

```
channel.basic_consume(queue=queue_name, on_message_callback=callback,
auto_ack=True)
channel.start_consuming()
```

Code 1 RabbitMQ Routing Publish/Subscribe implementation in Python

6.3 TraceAnalysis Service

This experiment considers the power consumption, in Watts, as the energy metric under evaluation. To estimate it, the following hardware resource usage metrics were used as input:

- **CPU Load:** Percentage of time executing in the CPU
- **Disk Write Data Rate:** Bytes written per second in the disk
- **Disk Read Data Rate:** Bytes read per second in the disk
- **Memory Usage:** Percentage of primary memory allocated

The **TraceAnalysis** service is responsible for the computation of these metrics by analysing the software traces of systems specified in AMALTHEA models. This service is triggered by an external entity and the analysis is performed by dividing the trace into several parts with identical time periods, which can be configured according to the user's needs.

In the first place, BTF traces are gathered with a simulation or monitorization tool, and the collected information is stored in an AMALTHEA Trace Database, which is analysed by the service. However, it is not always possible to collect all the necessary information for the metric calculation without also consulting AMALTHEA system models. The system model is particularly necessary when simulation traces are used, providing Information such as the hardware features and specific configurations of the system.

At this point, the development of the **TraceAnalysis** service is explained, considering the adopted technologies, and the mechanisms that were applied to access AMALTHEA trace databases and system models. Finally, the description of the employed techniques to perform the computation of the input hardware resource usage metrics is also provided.

6.3.1 Adopted Technologies and Interactions with AMALTHEA System Models

In APP4MC, the AMALTHEA system model was specified using the Eclipse Modelling Framework. Therefore, the **TraceAnalysis** service was implemented using the Java programming language, since EMF supports the interaction and manipulation of every element from a model through a set of automatically generated Java classes.

Consequently, when information from the system model is needed, the elements from AMALTHEA can be imported to a Java project, and the system becomes able to interact with them in a programmatic way.

The code represented in the following snippet exemplifies the adopted techniques to interact with AMALTHEA system models through java classes:

```
import org.eclipse.app4mc.amalthea.model.*;

class LabelAnalysisService {

Amalthea model;

LabelAnalysisService(Amalthea model) {
  this.model = model;
}

Map<String, List<String>> findPrimaryLabelsUsedByRunnable() {

  EList<Runnable> runs = this.model.getSwModel().getRunnables();
  Map<String, List<String>> rmMap = new HashMap<>();

  for (Runnable r : runs) {
    List<String> labels = new ArrayList<>();
    for (ActivityGraphItem it : r.getActivityGraph().getItems()) {
         if (it instanceof LabelAccess && (!((LabelAccess) it).getData()
         .getMappings().get(0).getMemory().getDefinition()
         .getMemoryType().equals(MemoryType.FLASH))){
            labels.add(((LabelAccess) it).getData().getName());
         }
    }
    rmMap.put(r.getName(), labels);
  }
  return rmMap;
 }
```

Code 2 Access to AMALTHEA Models using Java Classes

This example refers to a method specified in the **TraceAnalysis** service. It belongs to the **LabelAnalysisService** class, which is responsible for performing a set of operations related to the analysis of memory label elements in AMALTHEA models.

The **findPrimaryLabelsUsedByRunnable()** method finds the primary memory labels that are used by each runnable of the system, returning a map where the key is a runnable name, and the value is the list of primary label names that it accesses.

To perform this operation, the access to various attributes from the AMALTHEA model is necessary. First, the runnable list from the Software Model must be consulted and iterated. Then, for each label accessed by a runnable, the hardware element to which it is mapped must also be consulted, in order to verify if it belongs to a primary memory or to the disk. If the label belongs to a primary memory, it is added to the list, else, it is discarded.

6.3.2 Accesses to AMALTHEA Trace Databases

Since the ATDB consists of an SQLite database, the Java Database Connectivity (JDBC) API was used to manage the accesses to it. JDBC consists of an industry standard mechanism that allows the management of SQL databases and tabular data sources through Java (ORACLE,

2021). To configure it, the SQLite JDBC Driver was used, consisting of a JAR archive that can be imported to Java projects (Xerial, 2020).

In the **TraceAnalysis** service, the connections to the ATBD are managed through the **ATDBRepo** class, composed of several methods to query information from the database, as exemplified in the following code snippet:

```
import java.sql.*;
(…)
class ATDBRepo {
  Connection amaltheaDBCon;

 Map<String,List<RunnableTraceEvent>> getRunnableExecutionsBetween(String runnab
 leName, Long begin, Long end) throws SQLException {

   Map<String,List<RunnableTraceEvent>> evR = new HashMap<>();
   String qMetricByEnt = "SELECT * FROM vTraceEvent Where entityName = '"
     +runnableName + "' AND timestamp >= "+begin+" AND timestamp <= "+ end
     + " AND (eventType = 'start' OR eventType = 'terminate' OR"
     + " eventType = 'suspend' OR eventType = 'resume')";

   try (Statement stmt = amaltheaDbConnection.createStatement();
     ResultSet rs = stmt.executeQuery(qMetricByEnt))
     while(rs.next()){
     String taskName = rs.getString("sourceEntityName");
     if(evR.get(taskName) == null) {
       evR.put(taskName, new ArrayList<>());
     }
     evR.get(taskName).add(new RunnableTraceEvent(
     rs.getLong("timestamp"),rs.getString("entityName"),
     rs.getString("eventType"),taskName));
   }
   rs.close();
 } catch (SQLException e) {
   System.out.println("ERROR: "+e.getMessage());
 }
 return evR;
 }
```

Code 3 Fragment of the ATDBRepository Class

The fragment of the repository class demonstrates that it is composed of one attribute, the connection between the system and the trace database, which consists of an **.atdb** file that is passed through the class constructor. Additionally, the **getRunnablesExecutionsBeween()** method is represented, exemplifying how queries are performed to AMLATHEA trace models.

This method's purpose is to collect the execution events of a runnable between two timestamps, supporting the **TraceAnalysis** service during the calculation of the hardware resource usage metrics. Therefore, the runnable's name and two timestamps are passed as parameters, and the query that is defined in the function collects all the execution events targeting the runnable, between the provided timestamps.

Finally, this method returns a key-value map with a list of **RunnableTraceEvent** objects grouped by task. This map is created since every runnable instance belongs to a task, and due to the fact that BTF traces only associate label accesses with tasks, instead of runnables.

Additionally, although it was not experienced in this project, within the scope of the AMALTHEA framework, different tasks may execute instances of the same runnable in parallel. Consequently, the distinction between parent tasks is necessary to guarantee a correct metric calculation process.

6.3.3 CPU Usage Calculation

One of the hardware components comprised by the solution to perform the power consumption estimation of a runnable entity is the CPU. From the BTF perspective, a runnable instance can present 3 different states, as illustrated in Figure 31.

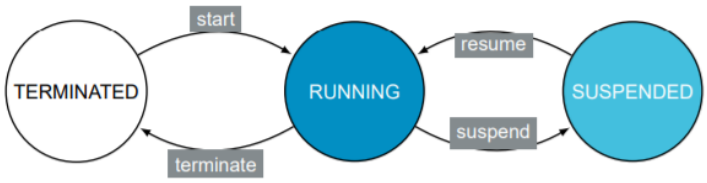

Figure 31 Runnable State Chart, retrieved from (Vector Informatik GmbH, 2020)

The **Running** state represents the runnable execution on a core. The **Suspended** state refers to when a runnable stops its execution without finishing it. Finally, the **Terminated** state means that the runnable has already finished its execution, or that it has not started it yet (Vector Informatik GmbH, 2020).

Like this, the CPU usage of a runnable execution can be calculated through the percentage of time that it spends in the **Running** state, considering a period under analysis. However, BTF traces do not directly represent the states of a runnable instance. Instead, they only provide a set of timestamped events which define the transitions between them:

- **Start:** when a runnable start its execution, from the **Terminated** state to the **Running** state
- **Resume:** when a runnable resumes its execution, from the **Suspend** state to the **Running** state
- **Suspend:** when a runnable suspends its execution, from the **Running** state to the **Suspended** state
- **Terminate:** when a runnable finishes its execution, from the **Running** state to the **Terminated** state

Taking this into account, to verify the execution time of a runnable, the sum of the differences between the timestamps where a consecutive **Start** or **Resume** event and a **Terminate** or **Suspend** event occur must be computed.

In this way, the first step of the CPU usage calculation process, for a given period, is to query the runnable trace events from the ATDB, with the function represented before, in Code 3.

Afterwards, it becomes possible to calculate the CPU load, as demonstrated in the following code fragment:

```
RunnableStats getRunnableStats
(List<RunnableTraceEvent> runEvents, Long begin, Long end) {

  Long execTime = 0L;
  Long traceTime = end - begin;
  RunnableStats rs = new RunnableStats(begin, end
  , runEvents.get(0).getTaskName());
  rs.setName(runEvents.get(0).getRunnableName());

  // (...)
  for (int i = 0; i < runEvents.size(); i += 2) {

        execTime += runEvents.get(i + 1).getTimeStamp() -
        runEvents.get(i).getTimeStamp();
      // (...)
  }
  // (...)
  double cpuNum = AmaltheaModelUtils.numOfCpusOfTask(this.model
    ,runEvents.get(0).getTaskName());
  double cpuUsage= execTime.doubleValue() * 100.0 / traceTime.doubleValue();
  rs.setCpuUsage(cpuUsage/cpuNum);
  // (...)
  return rs;
}
```

Code 4 CPU Usage Calculation in the TraceAnalysis Service

Here, the **getRunnableStats()** method from the **RunnableBTFStatsService** class is partially represented. It is responsible for calculating the hardware resource usage metrics for a set of **RunnableTraceEvent** objects occurred between the timestamps passed as parameters of the function.

In the first place, it creates a **RunnableStats** entity to store the metrics related to the runnable execution during the period under analysis. Then, it iterates the received **RunnableTraceEvent** entities two by two. For each set of two events, it subtracts the timestamp of the first event from the second one, since the first event of a runnable instance is always the **Start** event, and it is impossible for consecutive events to keep the runnable in the same state. Finally, the differences between timestamps are all summed up, and the CPU usage is assessed through the division of the obtained value by the by the length of the period.

Additionally, depending on the number of cores belonging to the hardware platform, the CPU usage, when calculated with this technique, can reach values above 100%. For example, if the platform has 2 cores, two tasks may execute in parallel, and the execution time of the system would be the double of the elapsed time, presenting a CPU usage of 200%. Therefore, since it was defined that, for this experiment, the CPU Load metric must be limited between 0% and 100%, before completing the metric calculation process, the number of cores of the hardware platform is consulted in the system model, and the CPU usage is divided by the obtained value.

6.3.4 Disk Data Rates Calculation

In AMALTHEA, an access to the disk is represented as an access to a memory label that is mapped to a secondary memory. To identify if a memory definition in AMALTHEA is primary or secondary, the memory type must be consulted. Then, the disk data rate can be calculated, consisting of the amount of data that is written or read by a software entity in bytes per second.

In BTF (Vector Informatik GmbH, 2020), the **Signal Events** specify an access from a task to a memory address, or a label when applied to AMALTHEA. There are two different **Signal Events**, the read event, which is used to calculate the disk read data rate, and the write event, used to calculate the disk write data rate.

As a result, the first step to calculate the disk data rate metrics is to collect all the **Signal Events** referring to a certain runnable. Still, since **Signal Events** are bound to tasks instead of runnables, it becomes necessary, for the runnable's execution time, to collect all the **Signal Events** targeting the task to which the runnable entity is related.

Like the CPU usage calculation, this operation is performed inside the **getRunnableStats()** function:

```
RunnableStats getRunnableStats
(List<RunnableTraceEvent> runEvents, Long begin, Long end) {
 // (...)
 String queryLabel = "(";
 for (int i = 0; i < runEvents.size(); i += 2) {
  // (...)
  if (queryLabel.equals("(")) {
    queryLabel += "(timestamp > " + runEvents.get(i).getTimeStamp()
    +"AND timestamp <= "+runEvents.get(i + 1).getTimeStamp() + ")";
  } else {
    queryLabel += "OR (timestamp > " + runEvents.get(i).getTimeStamp()
   +"AND timestamp <= "+runEvents.get(i + 1).getTimeStamp() + ")";
 }}
// (...)
```

Code 5 Collecting the Executions Periods of one Runnable

When the execution time of a runnable is being calculated, through the difference between the timestamps of two consecutive events, the **queryLabel** variable stores the execution intervals of the runnable, as exemplified in the following statement.

```
(timestamp > 0 AND timestamp <= 1) OR (timestamp > 3 AND timestamp <= 4)
```

Code 6 Example of the Variable with the Runnable Execution Intervals

In this way, the labels accessed by a runnable can be fetched from the ATDB by consulting the labels accessed by the runnable's task between the intervals represented in the variable created before, which can be integrated in an SQL query, as exemplified in the following code fragment.

```
SELECT entityName,eventType,count(*) FROM vTraceEvent
WHERE eventType= 'read' AND entityType = 'SIG' AND sourceEntityName = 'TASK_X'
AND ((timestamp > 0 AND timestamp <= 1) OR (timestamp > 3 AND timestamp <= 4))
GROUP BY entityName
UNION
SELECT entityName,eventType,count(*) FROM vTraceEvent
WHERE eventType= 'write' AND entityType = 'SIG' AND sourceEntityName = 'TASK_X'
AND ((timestamp > 0 AND timestamp <= 1) OR (timestamp > 3 AND timestamp <= 4))
GROUP BY entityName
```

Code 7 Example of an SQL Query to find the Label Accesses of a Runnable

Finally, all the Label accesses are gathered into a set of **LabelAccess** entities, composed of the access type, the label name, and the number of accesses. Each **LabelAccess** is analysed, and if the label belongs to the secondary memory, it is used to calculate the disk data rate.

Read operations are included in the runnable's disk read data rate, and write operations are included in the disk write data rate metric. To obtain the amount of bytes associated with each label, the **LabelInfo** entities that were created in the beginning of the process are used, as specified in the following code snippet:

```
void updateLabelResourceUsageOfRunnableStats
(RunnableStats rs, List<LabelAccessDTO> labAccesses, Long traceTime) {

  for(LabelAccessDTO labA : labAccesses) {
     LabelInfo labelInfo = findLabelInfo(labA.labelName,this.labelInfos);
     if(labelInfo != null
     && labelInfo.getType().equals(LabelType.SECONDARY)) {
       Long bytesL = (long) (1000000000L *
       labA.accessCount * labelInfo.getBytesCapacity());
       if(labA.accessType.equals(LabelAccessTypeDTO.read)){
          rs.setDiskRead(rs.getDiskRead()+(bytesL/traceTime));
       }else {
          rs.setDiskWrite(rs.getDiskWrite()+(bytesL/traceTime));
}}}}
```

Code 8 Method to Calculate the Disk Data Rate of a Runnable

6.3.5 Memory Usage Calculation

The memory usage of a runnable entity is the fourth and last input metric used for the power consumption estimation. The Proportional Set Size (PSS) memory was the adopted metric, consisting of the amount of memory allocated to a process, evenly dividing it between the number of entities to which it is assigned (Psutil, 2020).

Unlike the other metrics, which are dynamic and rely in the information of a trace, the memory usage is static, consisting of an allocation. Consequently, it does not depend on the behaviour of the system, being calculated by the analysis of AMALTHEA System Models.

In this manner since this metric is consulted multiple times. To avoid the unnecessary repetition of operations, in the beginning of the **TraceAnalysis** execution, the

findPrimaryLabelsUsedByRunnable() method is called, retrieving a key-value map with the list of primary labels used by each runnable, as represented before, in Code 2.

Afterwards, during the end of the **getRunnableStats()** execution, the memory usage of a runnable entity is calculated through the following method:

```
Map<String,List<String>> runMemMap;
// (...)
private void updateMemoryUsageOfRunnableStats(RunnableStats rs) {
  String name = rs.getName();
  List<String> labelNames= this.runMemMap.get(name);
  for(String ln : labelNames) {
    LabelInfo li = findLabelInfo(ln,this.labelInfos);
    if( li != null){
      rs.setMemUsage(rs.getMemUsage() + li.getMemoryUsage());
}}}
```

Code 9 Method to Calculate the Memory Usage of a Runnable Instance

For a **RunnableStats** entity provided as input, the list of primary labels that it uses is retrieved by the map. Then, for each allocated label, the corresponding **LabelInfo** is consulted and the memory usage of the **RunnableStats** object is increased.

The memory usage of a single primary label is calculated in the beginning of the **TraceAnalysis** execution, like the creation of the key-value map of the primary labels used by the runnable entities. This information is gathered during the creation of the list of **LabelInfo** entities, as illustrated in the following code snippet:

```
double findLabelUsage(Label l, long numberBytes, Long memTot) {

  double c = 0;
  EList<Runnable> runs = model.getSwModel().getRunnables();
  for(Runnable r : runs) {
    for(ActivityGraphItem it : r.getActivityGraph().getItems()) {
      if(it instanceof LabelAccess) {
        if(((LabelAccess) it).getData().getName().equals(l.getName()))
                {
            c ++;
            break;
  }}}}
  if(c == 0) {
    c = 1;
  }
  return (((double) numberBytes*100)/c)/((double) memTot);
}
```

Code 10 Method to Calculate de Memory Usage of a Label

In this method, the parameters passed as input are a memory label, its size in bytes, and the total memory of the system in bytes. Then, to compute the memory usage, all the runnables of the AMALTHEA model are iterated, and the number of runnables that uses that label is registered. In the end, the size of the memory label is divided by the obtained value, and the

PSS memory usage is calculated through the division of the result by the total memory of the system.

6.3.6 Limitations

One limitation related to the assessment of the disk data rate metrics was found during the implementation of the **TraceAnalysis** service. Since the analysis of this metric is performed through the ATDB, which is composed of timestamped events, and since the Signal event is associated with a task and not with a runnable, the solution is only capable of ensuring the accurate calculation of the disk data rates if memory access latencies are defined.

Figure 32 represents part of a BTF trace used to supply an ATDB with information. It is composed of 2 runnable events represented with the letter **R,** and two signal events represented with **SIG**.

```
165813,Task_10MS,0,SIG,BrakePedalPosition_type_Label,0,write,
165813,Task_10MS,0,R,CheckPlausability,0,terminate,
165813,Task_10MS,0,R,BrakeActuatorMonitor,0,start,
165815,Task_10MS,0,SIG,BrakeForce_type_Label,0,read,
```

Figure 32 BTF trace with a runnable Terminate event, a Start event, and a Signal event simultaneously

It is possible to verify that when the **CheckPlausability** runnable terminates its execution, at the same time that the access to the **BrakePedalPosition_type_Label** is registered, the beginning of the **BrakeActuatorMonitor** runnable is verified.

If no latencies were defined to memory accesses, the second signal event would also occur at the same time of the other events, and not 2 ns later, as represented in the trace. In this way, it would be impossible to know which memory accesses belong to the runnable that finished its execution, or to the runnable that started it, considering the timestamp of the event.

Therefore, to provide the accurate computation of this metric, memory latencies are mandatory. The system considers that a signal event of a task belongs to one of its runnables that has already started its execution and has not finished it yet. If the event occurs at the same time of the **Start** event, it will be considered as an event referring to other entity.

6.4 EnergyPrediction Service

With the hardware resource usage metrics collected, the **EnergyPrediction** service is responsible for the power consumption estimation process. A machine learning approach was adopted, based in the work described in Section 5.2.5 (Singh, et al., 2013). It proposes a fine-grained power consumption estimation mechanism using a nonlinear Support Vector Regression model and providing more accurate results than a linear alternative to which it was compared.

Although this work proposes the power consumption estimation through the CPU load, the memory usage, the disk write and read data rates, and the network transmit and receive data rates, our approach does not comprise the network component. Limitations were experienced in terms of process-level data collection. In addition, it presented an almost negligible influence of about 0.5% in the model's accuracy, something that was also faced in similar projects (Singh, et al., 2013; Kansal, et al., 2010).

6.4.1 Support Vector Machines

The Support Vector Machines algorithm (Awad & Khanna, 2015) was originally designed for classification problems. It provides a solution to categorize a set of entities, splitting it into different classes divided by a decision boundary, the hyperplane.

The position of the hyperplane is defined by the Support Vectors, a subset of the training data that specifies its margins according to a set of parameters, influencing how the algorithm constructs the model.

Finally, the main goal of SVM is to find a hyperplane with the greatest possible margin in relation to the data that it separates. Like so, it attempts to provide a more generalized solution from the samples that are used during the training process, as represented in Figure 33.

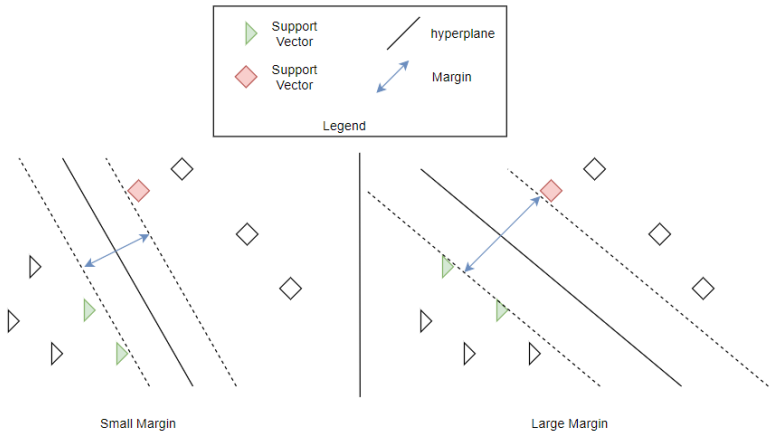

Figure 33 Example of an SVM problem (Gandhi, 2018)

Support Vector Regression consists of a SVM generalization capable of providing the accurate estimation of continuous values. Unlike the classification algorithm, SVR aims to provide a hyperplane comprising as much training data as possible within the shortest radius around it, as represented in Figure 34. Like so, it minimizes the error of the model, which is represented as the distance between the predicted and the expected values.

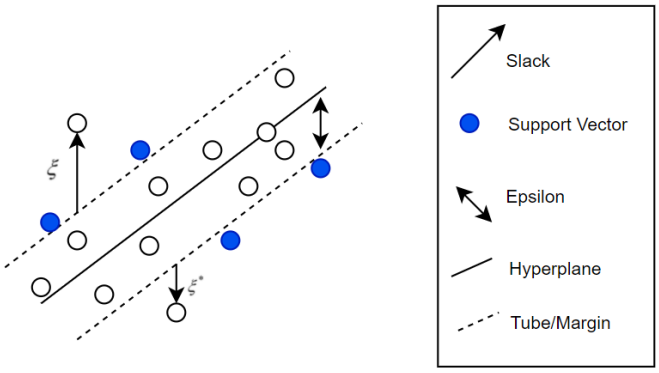

Figure 34 SVR model example

The radius of the hyperplane margin is defined by the **epsilon** parameter ($\boldsymbol{\varepsilon}$), acting as an error threshold. Values that do not attend the margin are considered as outliers, represented through the **slack** variables (ξ, ξ^*), which specify the deviation of the training data points in relation to the **epsilon** (Awad & Khanna, 2015).

In this way, the adoption of different **epsilon** values influences how the algorithm constructs the estimation model. Larger values are reflected in a greater margin and more data values inside of it. Otherwise, with smaller values, a strict radius is defined, resulting on a larger training error rate and more data values comprised outside the margins (Awad & Khanna, 2015).

Figure 35 exemplifies the manipulation of the **epsilon** parameter for the same training data set.

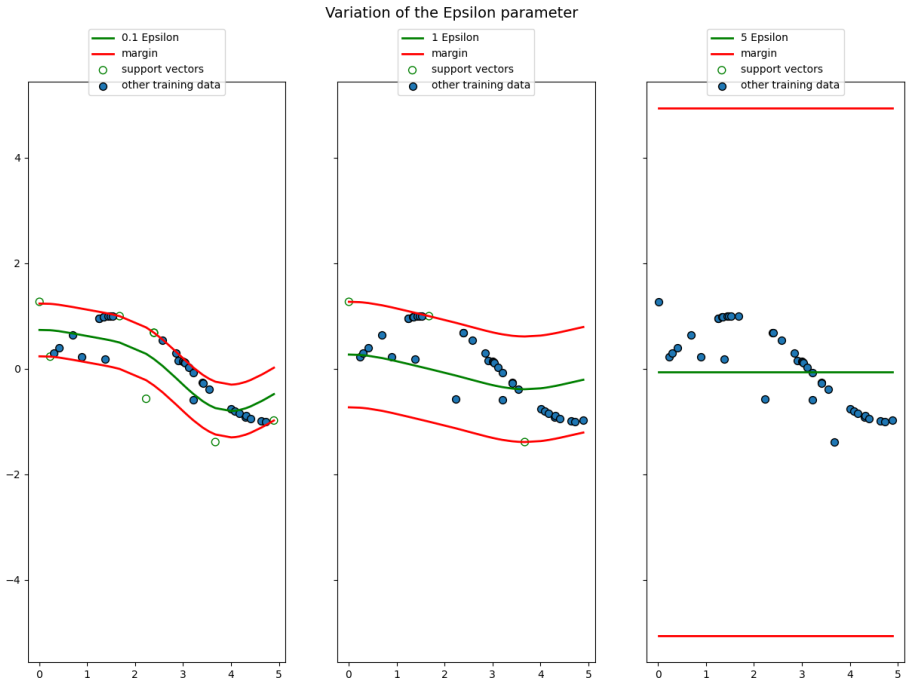

Figure 35 Variation of the Epsilon Parameter

The first graphic shows a model with a lower **epsilon** value. It is possible to verify that since the width of the radius around the hyperplane is shorter, a greater error rate is obtained, with a larger amount of data comprised beyond the defined boundaries.

The second graphic presents an **epsilon** value 10 times greater than the first one. In this case, the radius is considerably greater, including more data inside the tube, which is reflected in a substantially different hyperplane, furthest from the training data values.

Finally, the third graphic illustrates an even more increased **epsilon**. The model retrieved by the algorithm includes the entire data set within the defined boundaries. It presents a horizontal line as output, influenced by a wider set of values which, when compared to the boundaries, is significantly close to the hyperplane, and away from the margins.

Beyond the **epsilon**, more parameters affect the performance of SVR and were comprised for the calibration of the model implemented in the solution. Therefore, in this section, these parameters are also explained.

Kernel

One of the main features of Support Vector Machines is the fact that it can solve both linear and nonlinear problems. When a nonlinear problem is faced, SVR maps data into higher dimensions through the adoption of a **kernel** function, achieving a more accurate estimation model (Awad & Khanna, 2015).

SVM algorithms offer the possibility to adopt four different kernel types (scikit learn, 2020):

- Linear;
- Polynomial;
- Radial Basis Function (RBF);
- Sigmoid.

As the name suggests, the Linear kernel is intended to solve linear problems. On the other hand, the other three functions can be applied to nonlinear scenarios. Figure 36 exemplifies the differences between the adoption of linear and nonlinear kernels applied to the same data set:

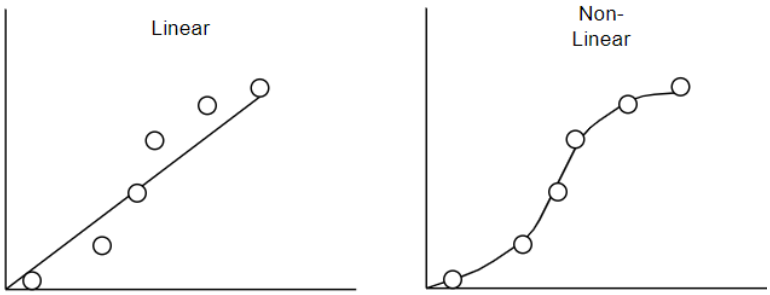

Figure 36 Linear Model Vs. Nonlinear Model

C Regularization Parameter

The regularization parameter outlines the error costs, defining the influence of the outliers in the model produced by the algorithm, considering the slack variables:

$$C \sum_{i=1}^{n} (\xi_i + \xi_i^*). \quad (12)$$

When **C** presents smaller values, the accuracy of the training data set's prediction decreases, presenting a greater error tolerance. On the other hand, with greater values, there is an effort to correctly estimate the training values (scikit learn, 2020), as represented in the equation described before.

Figure 37 illustrates the influence of the **C** parameter manipulation for the same data set.

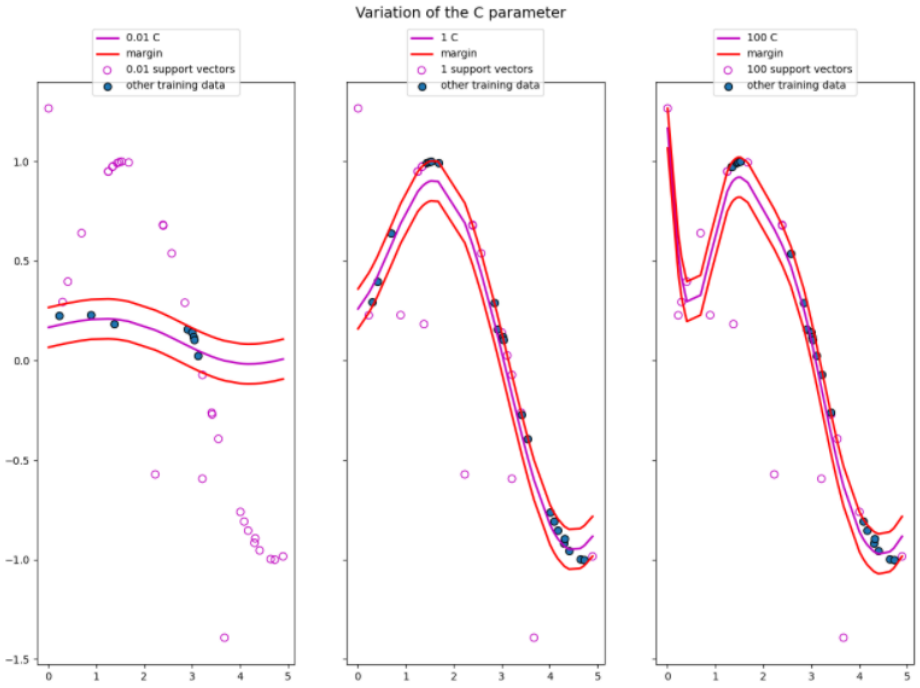

Figure 37 Variation of the C parameter

Three graphics are represented with an increasing **C** value from the left to the right. The manipulation of this parameter does not affect the radius of the hyperplane margin. However, since greater **C** values will tolerate less error rates, aiming to accurately predict the training set, the hyperplane will cover most of the data values, which may lead to overfitting and the inclusion of outliers in the produced model.

On the other hand, when the value is reduced, since higher errors are tolerated, the produced model will ignore and be less influenced by outlying data values. However, it may not be able to correctly capture the distribution of the data points.

Gamma

Gamma is a parameter from the **RBF** kernel that defines the influence reach of the training data values. When the **Gamma** is low, the influence of a value presents a longer area. For a higher **Gamma**, the data values influence the model from a shorter distance (scikit learn, 2020).

In this figure, three solutions for the same data set are presented, each one with a different **Gamma**, which increases from the left to the right.

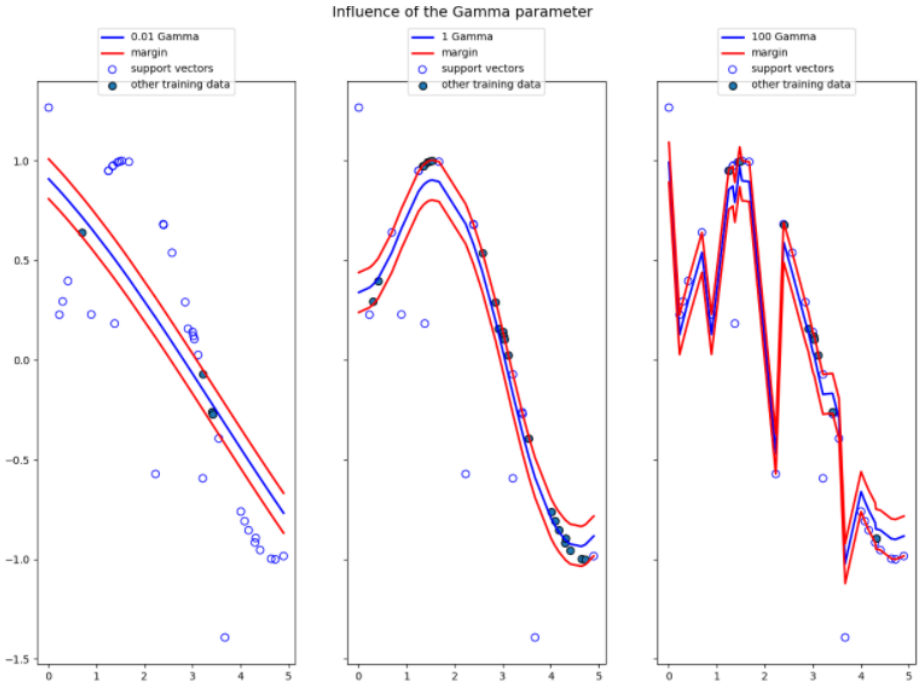

Figure 38 Variation of the Gamma Parameter

When **Gamma** presents smaller values, since the influence area of each data value is greater, it will cover more space, and the model will not be able to verify their distribution. Consequently, it will tend to behave similarly to a linear model. If the **Gamma** is set to a greater value, since the influence area of each data point will be smaller, the model will tend to be overfitted to the training set (scikit learn, 2020).

6.4.2 Model Calibration

The implementation of the **EnergyPrediction** service relies on a data collection process that supports the construction of the estimation model. This process was conducted considering the Nvidia Jetson AGX Xavier as the adopted hardware platform, and the training data collection was performed in the following manner:

- The input hardware resource usage metrics were collected using **psutil**, a cross-platform monitoring library for the Python programming language (Psutil, 2020).
- The power consumption was measured using Tegrastats, through the power monitors provided by the hardware platform (Nvidia, 2021).

Since there are no mechanisms to provide the measurement of the power consumption at the process-level, the model's training data was collected at the system-level. Later, process-level data was also collected to validate the mechanism through an indirect method, which is explained in the next chapter.

During the data collection process, **stress-ng** was used to provide variations to the system workload. It consists of a load testing tool that stresses several hardware components of Linux systems in a configurable way. **Stress-ng** allows the specification of the number of processes used to stress the system and the type of stress tests to be performed (Ubuntu Manuals, 2017). Thus, the model can be trained to estimate the power consumptions through a broader training data set, positively affecting its quality.

A total of between 0 and 10 processes were simultaneously executed using **stress-ng** to provide various execution environments, from an almost idle state to a system where all the processing units execute at the same time, making an intensive usage of the disk, the memory, and the network hardware components. The adoption of this tool was reflected in a variation of the power consumption of the system ranging from approximately 2.6 Watts to 8.3 Watts.

The hardware resource usage metrics were periodically measured, along with the power consumption in 1 second intervals, and a total of 6399 samples was collected. The obtained data entries were randomly separated into two different sets, one with approximately 70% of the collected data, used to train the model. Another composed of the rest of the samples, for further validation. In this way, it became possible to proceed with calibration of the regression model using the training data set.

To implement the model, **scikit-learn** was used, consisting of a Python machine learning library. In the first place, the collected data was treated with a Standard Scaler, a mechanism that standardizes data by removing the mean from each sample and dividing it by the standard deviation of the entire data set, in order to present an approximately normal distribution. This procedure is commonly used for machine learning estimation mechanisms such as SVR, increasing the accuracy of the regression models (scikit learn, 2020).

Afterwards, to conclude the procedure, it was necessary to define the parameters of the SVR model. This task was performed using the **GridSeachCV**, a technique offered by **scikit-learn**. It consists of an exhaustive search method which generates various estimator alternatives from a grid of model parameters. In the end, Grid Search evaluates each alternative against the training data, in order to provide the set of parameters that define the most reliable model (scikit learn, 2020).

The obtained model presented an **RBF** kernel, which consists of a nonlinear function. It was compared with a **linear** kernel, and according to the Grid Search mechanism, **RBF** proved to be the best alternative to solve the problem under study. Additionally, the other parameters suggested by the Grid Search were the **scikit-learn** SVR default values, **C** value as 1, **Epsilon** as 0.1, and **Gamma** as scale, corresponding to $1/(samplesLength * inputVariance)$.

6.4.3 Service Implementation

With the estimation mechanism defined, the development of the **EnergyPrediction** service can be concluded. This service was implemented using the Python programming language, since it was also adopted to calibrate the model.

Most of the efforts related to the implementation of this component are associated with the model calibration process. Once the model was constructed, it was persisted using the **Joblib** python library (Joblil, 2021). In this way, the repetition of the training process, which may be associated with considerable time costs, can be avoided, allowing the system to load the already calibrated estimator in a lightweight manner, every time it is deployed.

Subsequently, the SVR model was integrated into the **Prediction** class, which basically consists of a RabbitMQ consumer. It was configured to estimate the power consumption of the system at the runnable-level, every time data is received from the **TraceAnalysis** service, as defined in the following code snippet:

```
## idle_pwr = power consumption of the system when it is not executing
## regressor = SVR model
## sc_S.transform – standard scaler
## sc_t.transform – transforms the consumption back to the original scale
def callback(ch, method, properties, body):
  trace_list = json.loads(body)
  send_m = []
  for elem in trace_list:
    powers = []
    for stat in elem['runStats']:
      stand_data = sc_S.transform([[stat['cpuUsage'],stat['diskRead'],
      stat['diskWrite'],stat['memUsage']]])
      p1 = sc_t.inverse_transform(regressor.predict(stand_data))[0] – idle_pwr
      powers.append({
        "name" : stat['name'],"taskName" : stat['taskName'],"power": p1
      })
    send_m.append({
       "timestamp":elem['timestamp'],
       "hwPlatform": hw_plat,
       "stats": powers,
       "powerUnit":"W"
    })

  send_service.send_message(exchange_name,send_routing_key,send_m)
```

Code 11 Implementation of the Prediction service RabbitMQ consumer callback function

In the first place, the **EnergyPrediction** service receives a set of **AmaltheaTraceStats** entities. Since these entities are only destined to transport data, and not to perform any operation. The **EnergyPrediction** service does not rely on any class to represent it. Instead, **dictionaries** were used, storing key-value pairs in a similar structure to json, the adopted messaging format.

Every time data arrives from the message queue, the received **AmaltheaTraceStats** entities are iterated, along with the **RunnableStats** entities referred to each one of them. Thus, the service collects all the hardware resource usage metrics and performs the estimation of the power consumption at the runnable-level.

Finally, for each analysed runnable execution, it creates a **RunnablePowerStats** entity containing its power consumption, once again, using **dictionaries**. Then, when the process is concluded, the estimated data is transmitted to the message broker as a list of **TimedSystemPowerStats** entities, one per timestamp, composed of a set of **RunnablePowerStats** entities and defining the measurement unit of the estimated metric, which in this case is Watts. Additionally, the **TimedSystemPowerStats** entities specify the target hardware platform since the regression model was calibrated for a specific machine.

6.4.4 Limitations

Some limitations were faced during the implementation of the **EnergyPrediction** service, most of them related with the definition of the SVR model, during the calibration process.

In the first place, as mentioned before, there were constraints related to the network metrics assessment at the process-level. The collection process was highly influenced by the project on which our approach is based (Singh, et al., 2013), using information that can be retrieved by **procfs**[1] to collect metrics, a pseudo-filesystem that allows the simplified access to kernel data structures with information about the system and the executing processes.

That approach proposes the network metrics calculation by gathering information from the */proc/net/tcp, /proc/net/tcp6, /proc/net/raw,* and */proc/net/udp* files. According to the authors, each one of them presents the tx_queue_{inode} and the rx_queue_{inode} attributes, respectively representing the amount of data transmitted and received by an *inode*.

These values can be associated with a process, which is mapped to a set of *inodes* according to the file descriptors from the */proc/<PID>/fd* directory. However, during this experiment, this technique was tested without success. Usually, the information offered by **procfs** is stored in cumulative variables. In this case, the files specified under the */proc/net* directory only provide instantaneous values, and the collected metrics presented nearly 0 bytes received and transmitted by the system processes almost every time the information was consulted.

Consequently, the network component was discarded. Without access to this data, the validation of the estimation mechanism at the process-level is impossible, considering the network component. Additionally, the influence verified by the network metrics at the system-level was almost negligible and no other suitable measurement techniques were found.

Other problems were also faced during the process-level data collection process. While validating and comparing the process-level and system-level disk data rates values, it was verified that the data from these two granularities was out of phase.

[1] https://www.kernel.org/doc/Documentation/filesystems/proc.txt

This synchronicity issue exists since the process-level metrics represent the I/O operation between processes and kernel threads or the kernel block device subsystem. On the other hand, the system-level metrics represent the disk I/O operations between the kernel threads or the kernel block device subsystem and the actual disk (Wise, 2009).

However, despite this problem, the evaluation of the estimation mechanism was performed. The disk metrics were included, and since the obtained results were positive, this issue was ignored, and the model was accepted comprising the disk component.

Finally, since the implementation of machine learning algorithms is based on training samples with real-life data, there is always an inescapable limitation. Although the training data set was collected from a system under the influence of various stressing mechanisms, presenting several execution environments, the model is always more or less limited to the training data set.

The data collected to train the model presents a hardware resource usage comprised between the values represented in the following table:

Table 10 Limits of the SVR model

Metric	**Minimum**	**Maximum**
CPU Usage (%)	0	99,98
Disk Read Data Rate (B/s)	0	36700160
Disk Write Data Rate (B/s)	0	122683392
Memory Usage (%)	3,239	67,554
Power Consumption (W)	2,642	8,314

Therefore, the model is limited to this range of values. Data entries that stray too far from these boundaries may compromise the model's ability to return a reliable power consumption estimation.

6.5 ModelAnnotation Service

After the power consumption values are estimated, the **ModelAnnotation** is the last service to operate. Since this component directly interact with AMALTHEA models, it was implemented in Java, consulting information, and manipulating the models throughout the same techniques that were adopted in the **TraceAnalysis** service.

In the first place, it takes the data that is published by the **EnergyPrediction** service and groups it by runnable entity, performing a simple statistical analysis. Finally, it annotates the AMALTHEA System Model that specifies the system under analysis with the obtained information about the energy consumption of the system elements, supporting further energetic management and analysis.

6.5.1 Statistical Analysis

The statistical analysis of the estimated power consumptions is handled with the **DoubleStatistics** entity. The information that is computed for each runnable of the system is the following:

- Maximum Value.
- Minimum Value.
- Mean Consumption
- Standard Deviation.
- Number of Samples.
- List of Values.

This entity is initialized with an empty list of values and provides the **put()** operation, which is responsible not only for including new values in the list, but also for the statistical metrics calculation, as specified in the following code fragment:

```
class DoubleStatistics {

  List<Double> values;
  double max;
  double min;
  double average;
  double std;
  double stdev;
  int count;
  //(...)
  void put(double value) {
    if (value > this.max) {
       this.max = value;
    }
    if (value < this.min) {
      this.min = value;
    }
    this.count++;
    this.values.add(value);
    this.average = (this.average * (this.count - 1))+ value;
    this.average = this.average/this.count;
    double sumStd = 0;
    for (double d : this.values) {
      sumStd += Math.pow(d - this.average, 2);
    }
    this.stdev = sumStd / this.count;
    this.stdev = Math.sqrt(this.stdev );
  }}}
```

Code 12 Implementation of the DoubleStatistics Class

It starts by verifying if the introduced value is the minimum or the maximum value of the list, updating these attributes if necessary. Then, it increments the count attribute and includes the new value in the list. Finally, the average and the standard deviation are calculated.

For a certain trace, a set of **DoubleStatistics** entities is created, one for each runnable, in the **getStatsByRunnable()** function of the **PowerModelInstrumentor** class. To identify which **DoubleStatistics** belongs to each runnable, an HashMap was used with the runnable name as key and the statistics as value:

```
Map<String,DoubleStatistics> getStatsByRunnable(Amalthea model,
List<TimedSystemPowerStats> sysStats ){

  List<String> runNames = AmaltheaModelUtils.getRunnableNames(model);
  Map<String,DoubleStatistics> runMap = new HashMap<>();
  for(String name : runNames) {
         runMap.put(name, new DoubleStatistics());
  }
  for (TimedSystemPowerStats tsp : sysStats) {
         for(RunnablePowerStats rps : tsp.getStats()) {
                runMap.get(rps.getName()).put(rps.getPower());
         }
  }
  return runMap;

}
```

Code 13 Treatment of the Data received from the EnergyPrediction Service

In this way, the process of analysing and treating the data received from the **EnergyPrediction** service is finalized, grouping the information by runnable entity, and making it available to proceed with the model annotations.

6.5.2 Model Annotation of Energy Properties

According to Section 4.1, APP4MC provides the **CustomEntity** class to extend the model with information that is not included by default. Accordingly, each time the solution is applied to a trace, the obtained information is annotated in a **CustomEntity** named after the hardware platform where the system executes. This information is provided by the **EnergyPrediction** service, and it was the adopted nomenclature to allow the analysis of the same model using different hardware platforms, since the energy estimation depends on it.

For each runnable of the system, a **CustomProperty** is created and added to the **CustomEntity**. To provide a clear and well-defined structure for the annotation, this property is created using the **Map** type, allowing it to store a set of key-value pairs, each one with an attribute that must be annotated for each runnable, defined with other **CustomProperty** objects.

In this way, the **Map CustomProperty** of each runnable is filled with the necessary information. In the first place, a **Reference CustomEntity** is created to associate the runnable-level information with the analysed runnable. Then, a **String CustomEntity** is also defined, specifying the measurement unit of the annotated properties, which in this case is Watts.

Finally, the rest of the annotated properties are related with the **DoubleStatistics objects**. A **List CustomEntity** is created and composed of **Double CustomEntity** elements, one for each estimated consumption. In addition, five **Double CustomEntity** objects are created to store

the number of samples, the minimum value, the maximum value, the average value, and the standard deviation.

The following code fragment represents the model annotation process implementation, represented in the **instrumentModel()** function. Additionally, it illustrates the creation of the **CustomEntity** composed of the information about the system runnables, provided by the **newEntityDoubleStats()** function from the **ModelAnnotation** service:

```
Amalthea instrumentModel(
Amalthea model, List<TimedSystemPowerStats> sysStats
) {

      final AmaltheaFactory fac = AmaltheaFactory.eINSTANCE;
      //(...)
      Map<String,DoubleStatistics> runnablesConsumptionStats;
      runnablesConsumptionStats= getStatsByRunnable(model, sysStats);
      CustomEntity powerStats = createCustomEntity(swModel,sysStats, factory);
      swModel.getCustomEntities().add(powerStats);
      for (Runnable run : runs) {
         String runName = runnable.getName();
         if (runnablesConsumptionStats.containsKey(runName)) {
          DoubleStatistics powerStats=runnablesConsumptionStats.get(runName);

          MapObject mapR = fac.createMapObject();
          powerStats.getCustomProperties().put(runName, mapR);
          newEntityDoubleStats(mapR.getEntries(), powerStats, fac,siUnit, run);
 }}}

//(...)

void newEntityDoubleStats(
EMap<String, Value> customProps, DoubleStatistics stats, AmaltheaFactory fac,
String powerUnit, Runnable runnable) {

  ReferenceObject refRun = fac.createReferenceObject();
  refRun.setValue(runnable);
  customProps.put("runnable", refRun);

  StringObject unit = fac.createStringObject();
  unit.setValue(powerUnit);
  customProps.put("unit", unit);

  DoubleObject max = fac.createDoubleObject();
  max.setValue(stats.getMax());
  customProps.put("max", max);

  DoubleObject min = fac.createDoubleObject();
  min.setValue(stats.getMin());
  customProps.put("min", min);

  DoubleObject avg = fac.createDoubleObject();
  avg.setValue(stats.getAverage());
  customProps.put("avg", avg);

  DoubleObject stdDev = fac.createDoubleObject();
  stdDev.setValue(stats.getStdev());
  customProps.put("stddev", stdDev);

  DoubleObject samples = fac.createDoubleObject();
  samples.setValue(stats.getCount());
```

```
    customProps.put("samples", samples);

    ListObject values = fac.createListObject();
    customProps.put("values", values);

    EList<Value> valuesList = values.getValues();
    stats.getValues().forEach(v -> {
      DoubleObject lValue = fac.createDoubleObject();
      lValue.setValue(v);
      valuesList.add(lValue);
    });
  }
```

Code 14 Annotation Process Implementation

7 Experimentation and Evaluation

After the implementation of the approach, the evaluation process must be conducted. This chapter describes the experimentation and evaluation of the project, supporting a quantitative and systematic approach to verify the objectives and requirements fulfilment.

First, the project's research hypothesis is formulated. Then, we define quality indicators and information sources, along with the respective evaluation methodologies. Finally, the conducted experiments are described, and the obtained results analysed.

7.1 Research Hypothesis

The Research Hypothesis (Lavrakas, 2008) defines the expected results of a project, improving the researcher's ability to successfully conduct it. It must be unambiguously defined and testable in a quantitative way, considering the problem statement and the research questions, along with evaluation metrics and methodologies to verify the obtained results.

The research hypothesis of the project is the following one:

It is possible to support energy management on automotive software systems by providing energy properties extraction mechanisms, along with their integration on automatic and dynamic environments, through the analysis of AMALTHEA models and traces, and property model annotation mechanisms.

7.2 Evaluation Indicators

To perform the evaluation of the solution, verifying the research hypothesis, a set of indicators must be specified, considering the problem at hands and the requirements definition. This section describes the adopted evaluation indicators and justifies their relevancy to the project.

7.2.1 Accuracy

The Accuracy consists of an evaluation indicator that can be applied to formulas or estimation mechanisms. This metric is represented by the difference between the expected values and the estimated ones (Lavrakas, 2008).

In this project, to evaluate the accuracy, the inverse of the mean relative error was the adopted metric:

$$Accuracy = 100 - \frac{\sum_{i=1}^{nSamples} \frac{|ObservedValue_i - RealValue_i|}{\max(\epsilon,\ RealValue_i)}}{nSamples} 100 \quad (13)$$

Using this equation, the difference between each observed and real value is computed and divided by the real value or, in case it is 0, by the variable ϵ, consisting of a small positive number close to 0 to avoid undefined divisions. Afterwards, the mean relative error is obtained through the sum of each relative error divided by the number of samples. Finally, this value is converted to a percentage and subtracted from 100, representing the accuracy of the evaluated mechanism (scikit learn, 2020).

Within the scope of the approach, this metric is employed to evaluate the quality of the implemented energy estimation mechanisms, since it is essential to provide energy metrics as close as possible to the real consumptions of the system.

7.2.2 Effectiveness

It is expected for the solution to present a specific behaviour, performing a set of well-defined functionalities, according to the requirements specification of the project.

In this way, the Effectiveness measures the system's capability to fulfil the set of operations required by the solution. This indicator is calculated through the percentage of tasks correctly delivered by the system, regarding the expected ones:

$$Effectiveness = \frac{length_{deliveredTasks}}{length_{expectedTasks}} \times 100 \quad (14)$$

7.2.3 Degree of Automation

This project aims to provide a solution suitable for dynamic and automatic execution environments, relying as less as possible on manual tasks for the whole process, from the interpretation of the system traces to the model annotations.

Therefore, the Degree of Automation must be evaluated. This indicator is expressed as the percentage of tasks automatically executed by the system, considering the total amount of tasks:

$$Automation = \frac{length_{automaticTasks}}{length_{allTasks}} \times 100 \quad (15)$$

7.2.4 Response Time

The Response Time refers to the time required by the system to perform a certain operation. Since the solution is meant to be applied to execution traces and online analysis environments, this indicator becomes increasingly important.

To calculate de Response Time of the system, the difference between the beginning and the end of its execution is calculated:

$$ResponseTime = endTime - startTime \quad (16)$$

7.3 Evaluation Methodology

The adopted evaluation methodology is defined according to the metrics described in the previous section. The most appropriate measurement and analysis techniques must be carefully selected for each of the indicators and requirements specified before. Hence, the research hypothesis is verified if the observed values for the set of indicators are within the acceptable expected values. Otherwise, the approach cannot be completely considered as the solution to the problem at hands, as it does not fulfil all the demanded requirements.

7.3.1 Metric Estimation Mechanism Experimental Study

Considering the adopted and similar energy metrics extraction approaches, the evaluation of the estimation mechanism can be conducted through an experimental study (Singh, et al., 2013). As described in Chapter 6, to calibrate the regression model, a set of coarse-grained hardware resource usage metrics were periodically measured, along with the system power consumption.

Information from execution traces, simulation traces, or probing mechanisms can be used to compute the hardware resource usage information of the system at different granularities. Additionally, there are solutions to perform energy measurements at the system or hardware component granularity (Singh, et al., 2013), with tools such as watt-meters or other hardware-specific mechanisms like Tegrastats.

As a result, to evaluate the ability of the implemented mechanism to perform the assessment of system-level metrics, the system-level energy consumption can be predicted and compared with the values obtained during the data collection process described before, or a similar one. However, in this project, the estimation mechanism was implemented with the aim of performing runnable-level metric extraction. Therefore, since there are no methods to perform energy measurements with such granularity, the evaluation of the model's performance applied to fine-grained metrics can only be carried out through an indirect approach (Singh, et al., 2013).

Instead of using system-level input metrics, a set of processes is monitored, along with the rest of the system as a whole. Simultaneously, the power consumption values are measured at the system-level.

Afterwards, the consumption is estimated individually for each one of the monitored processes and the rest of the system. Finally, the system-level energy consumption can be

calculated through the sum of the values obtained before, which can be then compared with the expected ones, evaluating the estimation mechanism at a fine-grained granularity, namely at the process-level.

This method is employed to verify the **accuracy** of the energy estimation mechanism. The obtained mean accuracy of the prediction model is computed and compared with the expected accuracy of the solution. If the obtained value is greater or equal than the expected, the adopted approach meets the demanded requirements. Otherwise, it is not a suitable solution for the problem at hands.

7.3.2 Software Unit Testing

This project followed an iterative approach, where tests were applied to the developed functionalities according to the defined requirements, to allow the continuous validation and refinement of the solution.

In this way, Software Unit Testing is used to verify the behaviour of each isolated function of the system. The obtained results, serve as an input for the evaluation of the solution's **effectiveness**, providing information about which functionalities are correctly implemented.

It is expected from the project to provide a solution that correctly implements 100% of the proposed functionalities.

7.3.3 Experimental Use Case Application

To verify the applicability and correctness of the proposed approach, it can be applied to an AMALTHEA use case based on real-life automotive software scenarios.

This procedure ensures that the implemented functionalities present the expected behaviour, verifying to the solution's **effectiveness** from an integrated point of view. On the other hand, it becomes possible to evaluate the **automation degree**, which is an important feature for dynamic execution environments.

With the definition of the functional requirements, it is expected for the solution to execute 100% of its tasks automatically. In terms of effectiveness, it must also provide 100% of the expected functionalities. Therefore, after the execution of the system, the AMALTHEA model under analysis must become automatically annotated with the energy properties computed by the estimation mechanism through the input metrics provided by the analysis of the system execution or simulation.

7.3.4 Instrumentation Testing

To evaluate the **response time** of the solution, verifying its ability to perform the online analysis of executing software, an instrumentation testing procedure can be carried out.

Software instrumentation consists of a set of techniques that can be used to trace, debug, and analyse the performance of software systems, through the injection of additional code in certain parts of its implementation. As a result, the instrumented code may be used to collect events related the program's execution (Microsoft, 2017).

In this project, code instrumentation is performed in order to measure the timestamps of the beginning and the end of solution's execution, collecting its response times, and allowing the evaluation of the obtained results.

According to the requirement specification, if the **response times** presented by the system are less than the duration of the trace under analysis, the solution presents the ability to perform the online analysis of executing software.

7.4 Experiments and Results

With the implementation process concluded and the evaluation methodology planed, it becomes necessary to conduct the experiments defined above. In this way, this section describes the instantiation of the evaluation process, verifying the assessment of the evaluation indicators and providing an analysis of the obtained results.

7.4.1 Metric Estimation Mechanism Experimental Study

During the calibration of the estimation model, after conducting the training data collection process, about 30% of the obtained data set was held in reserve for validation purposes, corresponding to 1950 samples. Additionally, to provide a more comprehensive evaluation of the implemented SVR model, it was decided that an additional testing data set should also be collected.

The second collection process was like the first one. The hardware resource usage metrics and the power consumption were measured at the system-level in 1 second intervals. At the same time, **stress-ng** was used to provide different execution environments to the system. However, during this procedure, efforts were made to provide not only part of the workloads induced in the initial data collection process, but also some variations that were not used to train the model. Consequently, if the estimation model is overfitted to the training set, its performance would significantly decrease when different values are applied to it.

In the end, a testing set composed of 2494 elements was collected, and it was merged with the validation data set obtained during the initial collection process. Afterwards, the implemented mechanism was used to perform the estimation of the power consumption for each one of the collected samples.

To verify the accuracy of the estimation, the model evaluation utilities from **scikit-learn** were used, providing means to calculate quality indicators for regression models (scikit learn, 2020).

The obtained accuracy values are represented in the following table:

Table 11 Accuracy of the Estimation Model with System-level Metrics

Data Set	Linear Kernel	RBF Kernel
Training Data	95,00%	96,19%
Test Data	95,20%	96,23%

Along with calibration of the nonlinear RBF SVR model, a linear SVR model was also trained for comparison purposes. Although both models presented significantly positive results, it was verified that the nonlinear kernel provides a higher accuracy than the linear approach.

At this point, the obtained results suggest that the nonlinear SVR model is the most suitable way to estimate the power consumption of software at a course-grained level, considering the environment of the experiment. However, since the solution aims to support the power estimation at a fine-grained level, it becomes necessary to proceed with the indirect approach for the evaluation of process-level estimations described before.

To do such, a third and last collection process was conducted. In this case, the input metrics were collected at the process-level, for a selected set of processes belonging to the **stress-ng** tool. Additionally, the system-level metrics were collected, along with the power consumption. A total of 1000 samples was obtained, each one corresponding to a period of one second and containing the individual hardware resource usage of between 4 and 10 processes, the system-level input hardware resource usage, and the respective power consumption, ranging from 3.616 to 9.876 Watts.

The data set was then applied to the implemented models and evaluated in a comparable way to the system-level dataset described before:

Table 12 Accuracy of the Implemented Models at the Process-Level

Model	Linear Kernel	RBF Kernel
Accuracy	94.43%	96.12%

As in the previous experiment, we observed positive results. Once again, the non-linear SVR model outperformed the linear model, proving to be the most suitable solution for the problem in hands, considering the environment where the experiment was conducted, and the collected data.

Finally, according to the requirements definition of the project, the estimation mechanism must provide an **accuracy** equal or greater than 90%. Consequently, since the adopted model provides an accuracy of 96.12%, this requirement was successfully satisfied.

7.4.2 Software Unit Testing

The unit tests of the solution were only applied to the **ModelAnnotation** service and the **TraceAnalysis** service. Tests were not employed to the **EnergyPrediction** service due to the component's simplicity. It basically consists of the integration of the estimation model into a RabbitMQ consuming mechanism. In addition, the estimation model was already tested in the experiment described before.

Since both **ModelAnnotation** and **TraceAnalysis** services were implemented in Java, the unit tests were developed using the **Junit**[2] testing framework. Additionally, the **EclEmma**[3] Eclipse plugin was adopted to verify the code coverage of the test suite, as illustrated in Figure 39.

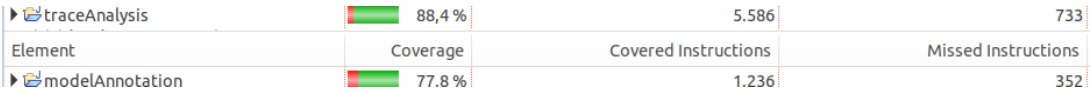

Figure 39 Unit Tests Coverage of the Solution

Although the coverage cannot be used to assess the quality of the test cases, it provides information about the amount of instruction that were analysed, being important to confirm if an appropriate number of tests was developed regarding the implemented code.

In this way, a total of 46 test cases was developed. Applying the **EclEmma** plugin to the projects, a coverage of 77.8% was obtained for the **ModelAnnotation** service and 88,4% for the **TraceAnalysis**. These values were accepted since there are parts of the system that were intentionally not tested, such as certain classes or simple methods like getters, setters, to string, main classes, and RabbitMQ operations.

The results of the software unit tests execution is represented in Figure 40.

[2] https://junit.org/junit5/
[3] https://www.eclemma.org/

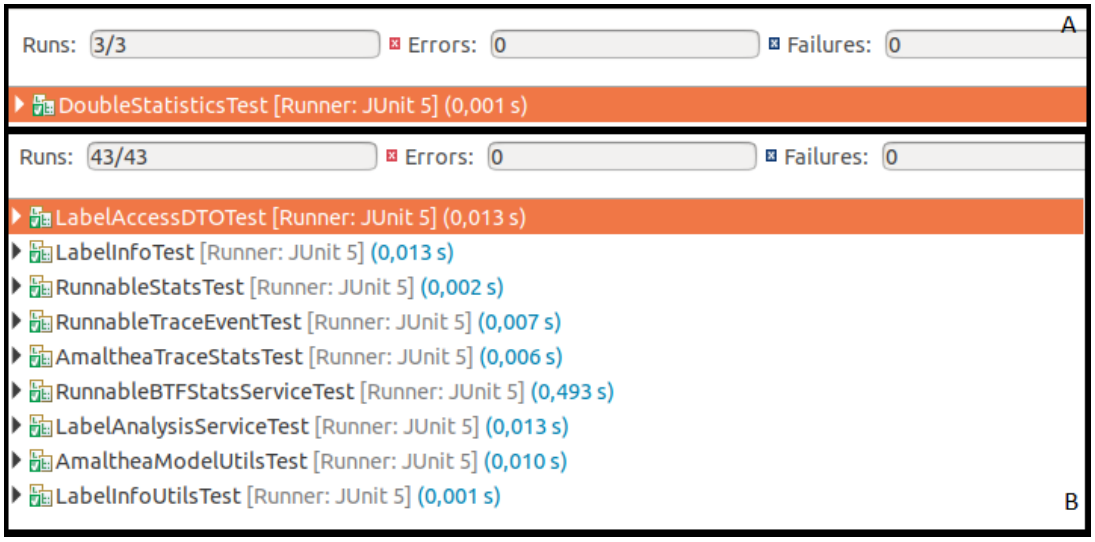

Figure 40 Unit Test Results. Image A represents the ModelAnnotation service, Image B represents the TraceAnalysis service

With the execution of the test suite, it was verified that all the tests passed without any error or failure. Since a percentage of 100% was expected in terms of the solution's **effectiveness**, this requirement was successfully fulfilled.

7.4.3 Experimental Use Case Application

The experiments described above were conducted to evaluate the quality of the solution by targeting its components in an isolated way. Additionally, the use case application evaluates the solution's effectiveness from an integrated perspective, being also essential to verify its applicability to real-life automotive use cases.

Therefore, the solution was applied to a use case specified in AMALTHEA based in an engine control application, the ETAS DemoCar Project (Frey, 2011). It consists of a model specifying an AUTOSAR based system composed of 43 runnable entities. Since the approach considers an Nvidia Jetson AGX Xavier board has the hardware platform of the systems under analysis, the DemoCar Project system model was adapted to present a similar hardware platform. Additionally, since the original model does not make any use of secondary memories, part of the memory labels was randomly mapped to a secondary memory instance.

In the first place, due to the absence of an implementation of this use case, simulation mechanisms were employed to collect the trace data of the system's execution. The simulation was performed for a total execution time of 10 seconds, gathered in a BTF trace, and automatically converted into an ATDB instance. Then, the solution was configured to analyse the trace database in a cyclic way, dividing it into 1 second intervals, which was the time period used to calibrate the estimation model.

Finally, the system was triggered and the properties resulting from the solution's execution were annotated in the AMALTHEA System Model, presenting (1) the list of power consumptions, (2) the adopted measurement metric, and (3) the results of the statistical analysis for each runnable of the system under analysis. In this way, all the expected tasks were properly executed, proving the solution's **effectiveness.**

In terms of the **automation degree**, it was verified that 100% of the functionalities offered by the solution executed independently and automatically, which suggests its applicability for dynamic and self-adaptive systems, also demonstrating that it meets the demanded requirements.

An extract of the output of the solution applied to the DemoCar Model is illustrated in Figure 41. It presents the model annotations provided by the solution, referring to a subset of the runnables specified in the DemoCar use case, the **BrakeForceArbiter**, the **BrakeForceCalculation**, and the **BrakePedalSensorDiagnosis**. For each one of the monitored runnables, a Map Custom Property object was created, composed of 8 different entries:

- **runnable**: Reference to the analysed runnable entity.
- **unit**: Measurement unit of the metric under analysis, such as Watts.
- **max**: Maximum obtained value.
- **min**: Minimum obtained value.
- **avg**: Average value.
- **stddev**: Standard deviation.
- **sample**: Number of entries.
- **values**: List of the estimated raw power consumptions.

Figure 41 Power Consumption Model Annotation

7.4.4 Instrumentation Testing

To conduct the Instrumentation testing experiment, in the first place, the implementation code was adapted to measure a timestamp at the beginning of the process, before analysing the input traces, and also in its conclusion, after the model annotations.

To perform this experiment, the system was deployed on an ASUS VivoBook, equipped with an Intel Core i7-1065G7 CPU, and 16 GB DDR4 of primary memory.

As in the previous experiment, the instrumented solution was applied to the DemoCar Model adapted to the Nvidia Jetson AGX Xavier board, along with a 10 second simulation trace. This system presents a considerable number of runnables, which implies substantial computation efforts to perform the analysis of its execution.

The solution was executed 30 times for the practical experimental set up, under the same environment and configurations, to avoid outlying results. The obtained response times for the sequence of 30 executions are presented in Figure 42.

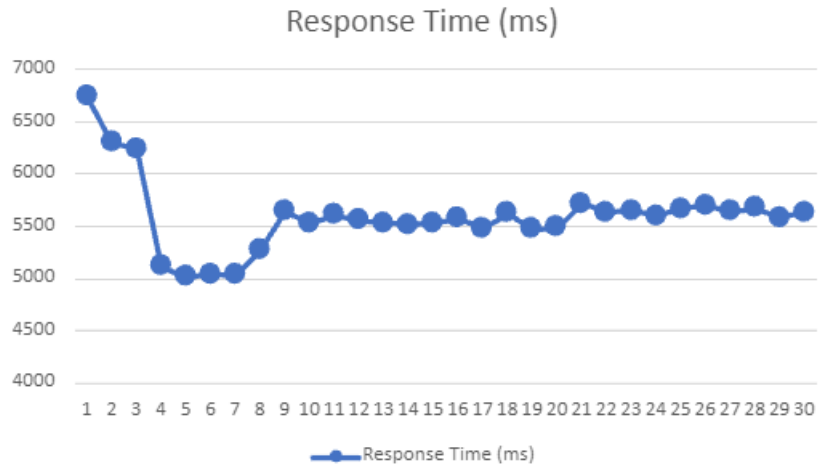

Figure 42 Response Time of the Solution

According to the results, the response times presented by the system were comprised between 5010 ms and 6743 ms, with a mean response time of 5592.233 ms, corresponding to 559.2233 ms per second.

As a result, our proof of concept solution is able to perform the online estimation of executing software, since, according to the requirements analysis, the system must present a **response time** inferior to the duration of the trace under analysis.

However, it is important to consider the adopted hardware platform. This experiment was conducted using a hardware platform with considerable computation power. If the solution were applied to other types of platforms, such as ECUs or other embedded systems, the response times presented by the system could be potentially different.

However, the application of our solution to such limited devices should occur in a scenario of online monitorization and analysis of executing software. This scenario excludes the processing of simulation traces, which has a potential positive impact on the response time of the system.

Therefore, the reduced computational power of the adopted hardware platform could be potentially balanced with an optimization of the trace analysis operation. When simulation traces are used, namely with the BTF format, the metric calculation process consists of the analysis of events referring to the execution and state of the software elements specified in AMALTHEA models, which can be associated with heavy operations. Alternatively, when execution traces are used, or the execution of the system is monitored, there are already certain tools that provide more refined metrics, such as psutil, or the Linux's procfs. Consequently, probing mechanisms could be employed and implemented in a more lightweight manner, counteracting the lack of computational capacity of the adopted platform.

8 Conclusions

The automotive industry has been experiencing a significant increase of software features. Despite several advantages, such as the improved safety and comfort of the vehicle, this evolution is also marked by a constantly growing complexity, and the rise of challenging requirements.

Consequently, efforts were made to provide improved standards and development methodologies, leading to the establishment of frameworks such as AMALTHEA/APP4MC. Additionally, due to the general electrification of automotive systems, it becomes necessary to ensure their energy efficiency. However, the traceability solutions offered by AMALTHEA are mostly focused on the timing properties of the system, and there is a reduced amount of work that addresses the energy domain.

Therefore, this project proposes a flexible and self-contained approach to support energy analysis and management mechanisms in automotive software systems specified under AMALTHEA. Through the analysis of AMALTHEA models and execution or simulation traces, energy estimation techniques were employed to compute the energy consumption of the system, supporting further analysis by annotating the obtained information in the models.

This solution consists of a modular system, providing extensible and reusable components which can be independently deployed, communicating by means of an asynchronous message broker. In this way, the approach was designed to be applicable to dynamic and self-adaptive systems, considering the concepts addressed by the Adaptive AUTOSAR standard.

In this chapter, the conclusions and discussion of the project results are presented. In the first place, the main objectives are evaluated, followed by a description of the overall obtained results. In the end, considering the limitations that were faced during the development process, and the main contributions of the project, a proposal for future work is provided.

8.1 Objective Fulfilment and Contributions

During the earlier stages of the project, the addressed problem was defined and contextualized. Therefore, we outlined an approach based on a set of four main objectives:

- Study and comprehension of automotive software systems and development methodologies.
- Study of software energy metric extraction mechanisms and management techniques applied to the automotive software domain.
- Development of an energy metric extraction mechanism relying on data that can be obtained through the analysis of AMALTHEA simulation and execution traces.

- Design and Implementation of a solution applied to the AMALTHEA framework for the analysis of execution or simulation traces, and the annotation of energy properties in AMALTHEA system models, obtained through the adopted metric extraction mechanism.

The first objective was accomplished through a study of the current state of the art. It became possible to contextualize and realize the environment where the solution is placed. On the other hand, the concepts addressed here provided useful information for the clarification of the adopted solution, namely, during the requirements gathering process and the selection of a software architecture.

The main features of automotive systems were described, along with some of the most important challenges and requirements. In addition, automotive software development standards and methodologies were studied, with a special focus on the AUTOSAR reference architecture. In the end, a contextualization of Model-Based Engineering techniques applied to this domain was also provided, along with the comprehension of the core concepts referring to the AMALTHEA framework, and a critical analysis of related technologies.

The second objective was also achieved through a study of the state of the art. In this case, the aim of the study was to explore approaches related with the analysis and management of energy requirements in the automotive software domain, defining the problems that are mostly faced in the industry, and techniques that promote the energy efficiency of automotive systems. To support the solution proposed in the project, energy measurement and estimation techniques applied to software systems were also studied and critically analysed.

In this way, an energy metric extraction mechanism was defined, supporting the third objective. To decide which energy estimation mechanism was going to be adopted, during the value analysis of the project, the AHP decision support method was used, taking advantage of the information provided by the state of the art study. Then, the mechanism was developed during the implementation stage of the project, providing the power consumption of the execution of a software entity through a set of input metrics that can be collected through AMALTHEA models and software traces.

Finally, the fourth objective was also successfully fulfilled. In the first place, with the information provided by the state of the art studies, the solution analysis was conducted. The domain of the problem was studied considering the most relevant concepts of the AMALTHEA/APP4MC platform technical specification. Moreover, the solution's requirements were also gathered, and the design of the approach defined, describing the solution from several perspectives. In this end, an experimental implementation and evaluation of the system became possible and was accordingly carried out.

8.2 Overall Results

Considering the solution analysis, a set of functional and non-functional requirements was specified. Thus, to validate the project's compliance with the requirements definition, the implemented system was submitted to an evaluation procedure composed of several experimentations.

In the first place, each component of the solution was individually evaluated, starting with the estimation mechanism. Since it consists of a nonlinear SVR model, it was compared with a linear alternative and evaluated regarding the estimator's accuracy. Two evaluation processes were conducted, one targeting system-level data, and the other focusing on process-level data.

In both procedures, the adopted model presented higher accuracy than the linear model. Besides, the obtained values were also acceptable considering the requirements definition. The expected accuracy of the estimation model was at least 90%, and the obtained accuracy was 96% for both process-level and system-level scenarios. Therefore, this mechanism was accepted and adopted for the solution.

To evaluate the model annotation and trace analysis operations, unit tests were employed to the solution, taking into consideration the functional requirements. A test suite composed of 46 tests was developed, covering over 70% of the target components implementation. The test cases were successfully executed, without presenting any failure. As a result, from an isolated perspective, each component of the solution presented the expected behaviour.

Finally, an AMALTHEA automotive use case was applied to the solution, verifying the compliance with major non-functional requirements and the verification of the system's behaviour, considering the integration of the implemented components. Once again, the system presented the expected behaviour, performing each task in an automated way, with the annotation of the energy properties obtained through the analysis of system traces in the AMALHTEA model under analysis, suggesting its applicability to dynamic systems, and the compliance with energy aware software development processes.

Furthermore, this use case was also applied to the solution in order to evaluate its response times. It was expected for the application to present a faster execution than the length of the trace under analysis. Therefore, to measure the response time of the system, instrumentation tests were performed, recording the beginning and the ending of the system's execution in timestamps. The execution of the solution was repeated 30 times, in order to collect a more comprehensive data set. Finally, the obtained results indicated that the system required an average of 5.592233 seconds to operate, almost half of the duration presented by the 10 second trace under analysis. Since the requirements definition specified that the solution must present lower response times than the trace under analysis, it was concluded that it presents an acceptable response time, being applicable to the online analysis of executing software.

8.3 Limitations and Future Work

Despite the accomplishment of the outlined objectives and the optimistic results provided by the solution, some limitations were faced throughout the project. Consequently, there is still room for improvement. On the other hand, the obtained results allow the extensibility of the solution, along with the possibility of exploring new approaches to the analysis and management of energy requirements under the scope of the AMALTHEA framework.

In the first place, considering the analysis of system traces, there is the impossibility of guaranteeing a reliable metric calculation process if the corresponding system model is specified without memory latencies. Since, in BTF, label accesses are not directly mapped to a runnable instance, but to its parent task. According to the AMALTHEA simulation mechanisms, two consecutive runnables belonging to the same task and executing in the same core would be able to simultaneously access memory labels. One runnable could access a label when its execution terminates, while the other would be able to perform an access when its execution starts, while the last runnable finishes its execution. For that reason, the solution would not be able to distinguish which label was accessed by which runnable entity. However, with label access latencies defined, a runnable becomes unable to perform a memory access at the exact same time that its execution starts, hence avoiding this issue.

Regarding the implementation of the estimation mechanism, it was not possible to collect fine-grained network metrics. Although this component did not present a noteworthy influence on the accuracy of the adopted mechanism, when other estimation techniques or hardware platforms are used, the network component may present a more considerable impact in the energy consumption of the system. Therefore, it is important to solve this issue, allowing the inclusion of network metrics in fine-grained level estimation mechanisms.

On the other hand, limitations related to the inclusion of the disk component were also faced. The values collected at the process-level were not synchronized with the corresponding system-level data. Still, since the accuracy provided by the estimation mechanism remained significantly high, it was deemed that this issue does not compromise the ability of the estimator to include disk input metrics.

In terms of future work, there are also some considerations related to the input data used by the estimation mechanism. One of the greatest advantages of regression models is the fact that they can estimate a metric through its relationship with a potentially open and unlimited set of other metrics. Therefore, it becomes important to study the assessment and inclusion of other relevant input data and hardware components, such as the Hardware Performance Counters and GPU information (Colmant, et al., 2019), with the aim of improving the quality of the energy estimation mechanism.

On another way, since the approach was designed to be flexible and extensible, it would be interesting to conduct experiments targeting different trace formats, analysis techniques, and estimation mechanisms, such as alternative probing and tracing mechanisms, regression methods, and machine learning techniques. Additionally, since the evaluation was only

performed over simulation traces, the application of the solution to perform the online analysis of executing software becomes essential, along with its application to other hardware platforms, such as ECUs.

Finally, this approach aims the support of energy analysis and management techniques in systems specified in AMLATHEA. Therefore, this project leads the way to conduct experiments related to the implementation and evaluation of such mechanisms using our solution to analyse the energy properties of the monitored systems.

References

Acar, H., Alptekin, G., Gelas, J. & Ghodous, P., 2017. The Impact of Source Code in Software on Power Consumption. *International Journal of Electronic Business Management, Electronic Business Management Society,* Volume 14, pp. 42-52.

Amalfitano, D., Simone, V. D., Fasolino, A. R. & Scala, S., 2017. *Improving Traceability Management through Tool Integration: An Experience in the Automotive Domain.* Paris, Association for Computing Machinery.

AMALTHEA, 2011. *Deliverable D1.1- State of the art of Design flow and verification methods and tools,* s.l.: s.n.

Ambrosio, J. & Soremekun, G., 2017. *Systems Enineering Challenges and MBSE Opportunities for Automotive System Design.* Banff, Canada, IEEE.

Antinyan, V., 2020. *Revealing the Complexity of Automotive Software.* New York, NY, USA, Association for Computing Machinery, p. 1525–1528.

Armstrong, K., Das, S. & Cresko, J., 2020. The energy footpring of automotive eletronic sensors. *Sustainable Materials and Technologies,* Volume 25, pp. 1-8.

Automotive SIG, 2021. *Automotive SPICE®.* [Online]
Available at: http://www.automotivespice.com/
[Accessed 26 01 2021].

AUTOSAR, 2008. *Technical Overview,* s.l.: s.n.

AUTOSAR, 2017. *Layered Software Architecture.* s.l.:s.n.

AUTOSAR, 2019. *Specification of the AUTOSAR Network Management Protocol,* s.l.: s.n.

AUTOSAR, 2020a. *Classic Platform Release Overview.* s.l.:s.n.

AUTOSAR, 2020b. *Explanations of Adaptive Platform Design,* s.l.: s.n.

AUTOSAR, 2020c. *Adaptive Platform Release Overview,* s.l.: s.n.

AUTOSAR, 2021. *AUTOSAR.* [Online]
Available at: https://www.autosar.org
[Accessed 14 01 2021].

Awad, M. & Khanna, R., 2015. *Efficient Learning Machines - Theories, Concepts, and Applications for Engineers and System Designers.* 1 ed. New York: Apress Media, LLC.

Barthels, A. et al., 2012. PREcup-1: An embedded system platform for prototyping ECU power management. *2012 IEEE Vehicle Power and Propulsion Conference*, pp. 1535-1540.

Bilic, D. et al., 2018. Model-Based Product Line Engineering in an Industrial Automotive Context: An Exploratory Case Study. *22nd Internation Systems and Software Product Line Conference*, pp. 56-63.

Bo, H., Dong, H., Dafang, W. & Guifan, Z., 2010. Basic Concepts on AUTOSAR Development. *2010 International Conference on Intelligent Computation Technology and Automation,* Volume 1, pp. 871-873.

Bologa, O., Breaz, R., Racs, S. & Crenganis, 2016. Using the Analytic Hierarchy Process (AHP) in evaluating the decision of moving to a manufacturing process based upon continuous 5 axes CNC machine-tools. *Procedia Computer Science ,* Volume 91, pp. 683-689.

Borza, J., 2011. FAST Diagrams- The Foundation for Creating Effectuve Function Models. *General Dynamics Land Systems*, 28 11.

Bourdon, A., Noureddine, A., Rouvoy, R. & Seinturier, L., 2011. *Linux: Understanding Process-Level Power Consumption.* Lisbon, s.n.

Broy, M., 2006. *Challenges in Automotive Engineering.* New York, USA, Association for Computing Machinery.

Broy, M. et al., 2010. Seamless Model-based Development: from Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE,* 98(4), pp. 526-545.

Chiara, F. & Canova, M., 2012. A review of energy consumption, management, and recovery in automotive systems, with considerations of future trends. *Journal of Automobile Engineering,* Volume 227, pp. 914-936.

CloudAMQP, 2021. *RabbitMQ as a Service.* [Online]
Available at: https://www.cloudamqp.com/
[Accessed 13 08 2021].

Colmant, M. et al., 2019. The Next 700 CPU Power Models. *Journal of Systems and Software,* Volume 144, pp. 382-396.

CORDIS, 2017. *Cost-Efficient Methods and Processes for Safety Relevant Embedded Systems.* [Online]
Available at: https://cordis.europa.eu/project/id/100016
[Accessed 20 06 2021].

CORDIS, 2017. *Model-Based Generation of Tests for Dependable Embedded Systems.* [Online]
Available at: https://cordis.europa.eu/project/id/216679
[Accessed 12 03 2021].

CRYSTAL, 2021. *CRYSTAL - CRITICAL SYSTEM ENGINEERING ACCELERATION.* [Online]
Available at: https://www.crystal-artemis.eu/
[Accessed 11 02 2021].

Culshaw, C. & Winter, A., 2007. *Optimizing Power Consumption of Automotive Systems Requiring Periodic Wake Up.* s.l.:s.n.

Dajsuren, Y. & Van den Brand, M., 2019. Automotive Software Engineering: Past, Present, and Future. In: Y. Dajsuren & M. Van den Brand, eds. *Automotive Systems and Software Engineering: State of The Art and Future Trends.* Cham, Switzerland: Springer International Publishing, pp. 3-8.

Dobbelaere, P. & Esmaili, K. S., 2011. *Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper.* Barcelona, Spain, Association for Computing Machinery.

Do, T., Rawshdeh, S. & Shi, W., 2009. A Process-level Power Profiling Tool. *Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, 10.

Doughty-White, P. & Quick, M., 2015. *Infromation is Beautiful.* [Online]
Available at: https://www.informationisbeautiful.net/visualizations/million-lines-of-code/
[Accessed 07 12 2020].

Dowdeswell, B., Sinha, R. & Haemmerle, E., 2016. Torus: Tracing Complex Requirements for Large Cyber-Physical Systems. *International Conference on Engineering of Complex Computer Systems,* Volume 21, pp. 23-32.

Ebert, C. & Favaro, J., 2017. Automotive Software. *IEEE Software,* Volume 34, pp. 33-39.

Eclipse APP4MC, 2020. *APP4MC 1.0.0 Online Help.* [Online]
Available at: https://www.eclipse.org/app4mc/help/app4mc-1.0.0/index.html
[Accessed 15 12 2020].

Eclipse APP4MC, 2021. *Eclipse APP4MC.* [Online]
Available at: https://projects.eclipse.org/projects/technology.app4mc
[Accessed 08 02 2021].

Eclipse Foundation, 2021. *Eclipse.* [Online]
Available at: https://www.eclipse.org
[Accessed 05 02 2021].

Eclipse Modeling Framework, 2021. *Eclipse Modeling Framework (EMF).* [Online]
Available at: https://www.eclipse.org/modeling/emf/
[Accessed 09 02 2021].

Economous, D., Rivoire, S., Kozyrakis, C. & Ranganathan, P., 2006. Full-System Power Analysis and Modeling for Server Environments. *Proceedings of Workshop on Modeling, Benchmarking, and Simulation*, pp. 70-77.

Eeles, P., 2001. *Capturing Architectural Requirements,* s.l.: s.n.

Eggert, A. & Ulaga, W., 2002. Customer Percieved Value: a substitute for Satisfation in business markets?. *Journal of Business & Industrial Marketing,* Volume 17, pp. 107-118.

Embitel GmbH, 2020. *Adaptive AUTOSAR vs Classic AUTOSAR: Which Way is the Automotive Industry Leaning?.* [Online]
Available at: https://www.embitel.com/blog/embedded-blog/adaptive-autosar-vs-classic-autosar
[Accessed 18 01 2021].

Eugster, P. T., Felber, P. A., Guirraoui, R. & Kemarrec, A.-M., 2003. The many faces of publish/subscribe. *ACM Computing Surveys,* 35(2), pp. 114-131.

European Commission, 2007. *The 2020 Climate and Energy Package.* [Online]
Available at: https://ec.europa.eu/clima/policies/strategies/2020_en
[Accessed 29 01 2021].

Fairley, P., 2018. *IEEE Spectrum - Exposing the Power Vampires in Self-Driving Cars.* [Online]
Available at: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/exposing-the-power-vampires-in-self-driving-cars
[Accessed 01 02 2021].

Fennel, H. et al., 2006. *Achievements and exploitation of the AUTOSAR development partnership.* s.l., Fennel 2006 Achievements AE.

Franz, E. et al., 2016. Requirements and tasks for active energy management systems in automotive industry. *14th Global Conference on Sustainable Manufacturing*, 10, pp. 175-182.

Fraunhofer Fokus, 2014. *ModelBus User Guide,* s.l.: s.n.

Fraunhofer FOKUS, 2020. *MODELBUS.* [Online]
Available at: https://www.modelbus.org/
[Accessed 12 02 2021].

Freitas Junior, V., Ceci, F., Woszezenki, C. R. & Gonçalves, C. L., 2017. Design Science Research Methodology. *Revista Espacios,* 38(6), pp. 25-34.

Frey, P., 2011. *A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems,* s.l.: Open Access Repositorium der Universität Ulm und Technischen Hochschule Ulm.

FTSRG, 2021. *Model-based Generation of Tests for Dependable Embedded Systems.* [Online]
Available at: https://inf.mit.bme.hu/en/research/projects/model-based-generation-tests-dependable-embedded-systems-mogentes
[Accessed 12 02 2021].

Fuerst, S., 2015. *AUTOSAR the Next Generation - the Adaptive Platform.* Paris: s.n.

Fuerst, S. & Bechter, I. M., 2016. AUTOSAR for Connected and Autonomous: Vehicles The AUTOSAR Adaptive Platform. *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops,* Volume 46, pp. 215-217.

Gandhi, R., 2018. *Support Vector Machine — Introduction to Machine Learning Algorithms.* [Online]
Available at: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
[Accessed 25 08 2021].

Garlan, D., 2000. Software architecture: a roadmap. *Conference on The Future of Software Engineering*, 05, pp. 91-101.

GitLab, 2021. *GitLab.* [Online]
Available at: https://gitlab.com/
[Accessed 17 08 2021].

Griessnig, G. et al., 2011. *Improving automotive embedded systems at European level.* s.l., Springer Verlag.

Guerel, E. & Merba, T., 2017. SWOT Analysis: A Theoretical Review. *The Journal of International Social Research,* 10(51), pp. 994-1006.

Guissouma, H., Klare, H. & Burger, E., 2018. An Empirical Study on the Current and Future Challanges of Automotive Software Release and Configuration Management. *44th Euromicro Conference on Software Engineering and Advanced Applications,* pp. 298-305.

Haghighatkhah, A. et al., 2017. Automotive software engineering: A systematic mapping study. *Journal of Systems and Software,* 128(0164-1212), pp. 25-55.

Hein, C., Ritter, T. & Wagner, M., 2013. Implementation of the RTP. In: A. Rajan & T. Wahl, eds. *CESAR- Cost-efficient Methods and Processes for Safety-relevant Embedded Systems.* Oxford: Springer, pp. 237-262.

Herstatt, C. & Verwoen, B., 2001. *The "Fuzzy Front End" of innnovation.* Hamburg: s.n.

Holtmann, J., Meyer, J. & Meyer, M., 2011. A Seamless Model-Based Development Process For Automotive Systems. *Software Engineering 2011 - Workshopband,* February, P-184(GI-Edition Lecture Notes in Informatics (LNI)), pp. 79-88.

Hunsley, J., 2019. The ECU diet: heterogeneous computing could save power, weight and space in the vehicle of the future. *M:bility Nagazine,* Volume Q1, pp. 48-51.

IBM Cloud Team, 2014. *SOA vs. Microservices: What's the Difference?.* [Online]
Available at: https://www.ibm.com/cloud/blog/soa-vs-microservices
[Accessed 11 08 2021].

IBM, 2021. *IBM.* [Online]
Available at: https://www.ibm.com/products
[Accessed 11 02 2021].

IEEE, 2004. *IEEE1003.13.* s.l.:s.n.

International Organization for Standardization, 2005. *ISO 17356-3:2005: Road vehicles – Open Interface for Embedded Automotive Applications – Part 3: OSEK/VDX Operating System (OS),* Geneva: s.n.

Iqbal, D. et al., 2020. Requirement Validation for Embedded Systems in Automotive Industry Through Modeling. *IEEE Access,* Volume 8, pp. 8697-8719.

ISO, 2011. *ISO 26262.* [Online]
Available at: https://www.iso.org/standard/43464.html
[Accessed 09 12 2020].

Joblil, 2021. *Joblib 1.0.1.* [Online]
Available at: https://joblib.readthedocs.io/en/latest/
[Accessed 23 08 2021].

Kansal, A. et al., 2010. Virtual machine power metering and provisioning. *ACM symposium on Cloud computing,* Volume 1, pp. 39-50.

Koen, P. et al., 2002. Fuzzy Front End : Effective Methods , Tools , and Techniques.

Krawczyk, L., Wolff, C. & D, F., 2015. Automated Distribution of Software to Multi-Core Hardware in Model Based Embedded Systems Development. In: G. Dregvaite & R. Damasevicius, eds. *Information and Software Technologies.* Cham: Springer International Publishing, pp. 320-329.

Krejčí, L. & Novák, J., 2017. Model-Based Testing of Automotive Distributed. *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 09, pp. 668-673.

Kruchten, P., 1995. The 4+1 View Model of Architecture. *IEEE Software,* 11, 12(6), pp. 42-50.

Lapierre, J., 2000. Customer-Perceived Value in Industrial Contexts. *Journal Of Business & Industrial Marketing,* Volume 15, pp. 122-145.

Larman, C., 2002. *Applying UML and Patterns.* 2 ed. s.l.:Prentice Hall.

Lavrakas, P. J., 2008. *Encyclopedia of Survey Research Methods.* Thousand Oaks, CA: Sage Publications.

Lazic, V., Djokic, M., Velikic, G. & Kovacevic, B., 2018. Adaptive AUTOSAR for vision based algorithms. *International Conference on Consumer Electronics* , pp. 1-4.

Lee, E. A., 2007. Computing foundations and practice for cyberphysical systems: A preliminary report.

March, S. T. & Smith, G. F., 1995. Design and natural science research in Information Technology. *Decision Support Systems,* Volume 15, pp. 251-266.

Marinho, E. & Ferreira Resende, R., 2012. Quality Factors in Development Best Practices for Mobile Applications. *international conference on Computational Science and Its Applications,* IV(12).

MaRS, 2012. *Crafting Your Value Proposition.* s.l.:Mars Discovery District.

Martínez-Fernandez, S., Ayala, C. P., Franch, X. & Nakagawa, E. Y., 2015. A Survey on the Benefits and Drawbacks of AUTOSAR. *S. {Martínez-Fernandez} and C. P. {Ayala} and X. {Franch} and E. Y. {Nakagawa},* pp. 19-26.

MBEDDR, 2013. *mbeddr - Engineering the Future of Embedded Software.* [Online]
Available at: http://mbeddr.com/
[Accessed 10 09 2021].

Mellor, S., Clark, A. & Futagami, T., 2003. Model-Driven Development. *IEEE Software,* Volume 20, pp. 14-18.

Microsoft, 2017. *Tracing and Instrumenting Applications.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/tracing-and-instrumenting-applications
[Accessed 11 10 2021].

Microsoft, 2019. *Microservices Architectural Style.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
[Accessed 11 08 2021].

Microsoft, 2020. *Choreography pattern.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/choreography
[Accessed 10 08 2021].

Murphy, S. & Kumar, V., 1997. The front end of new product development: a Canadian survey. *R&D Management,* Volume 27, pp. 5-16.

Neap, H. & Celik, T., 1999. Value of a Product: A Definition. *International Journal of Value-Based Management,* Volume 12, pp. 181-191.

Nikolic, M., Krunic, M., Milar, B. & Cetic, N., 2019. Utilization of pattern generators in Adaptive AUTOSAR Platform. *Telecommunications forum TELFOR,* Volume 27, pp. 1-4.

Noureddine, A., Rouvoy, R. & Seinturier, L., 2013. A Review of Energy Measurment Approaches. *ACM SIGOPS Operating Systems Review,* 47(3), pp. 42-49.

Nvidia, 2021b. *Nvidia, Jetson AGX Xavier.* [Online]
Available at: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[Accessed 23 08 2021].

Nvidia, 2021. *Nvidia.* [Online]
Available at: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/AppendixTegraStats.html
[Accessed 02 02 2021].

ORACLE, 2021. *Java SE Technologies - Database.* [Online]
Available at: https://www.oracle.com/java/technologies/javase/javase-tech-database.html
[Accessed 20 08 2021].

Outsystems, 2021b. *Monolithic vs. Microservices Architecture: When to Use What.* [Online]
Available at: https://www.outsystems.com/blog/posts/monoliths-or-microservices-make-both-your-domain/
[Accessed 11 08 2021].

Outsystems, 2021. *SOA vs. Microservices: What's the difference.* [Online]
Available at: https://www.outsystems.com/blog/posts/soa-vs-microservices/
[Accessed 12 08 2021].

Panorama, 2021. *Panorama.* [Online]
Available at: https://www.panorama-research.org/
[Accessed 05 02 2021].

Peffers, K., Tuunanen, T., Rothenberg, M. A. & Chatterjee, S., 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems,* 24(3), pp. 45-77.

Petrovic, O. & Kittl, C., 2003. Capturing the Value Proposition of a Product or Service. *International Workshop on Business Models*, 04 10, pp. 1-5.

Pisner, A. & Schnyer, M., 2020. Support Vector Machine. *Machine Learning: Methods and Applications to Brain Disorders,* pp. 101-121.

Polgár, B., Ráth, I. & Majzik, I., 2011. Model-based Integration Framework for. In: E. Schnieder & G. Tarnai, eds. *Forms/Format.* Berlin: Springer, pp. 227-235.

Polzer, A. et al., 2012. Managing complexity and variability of a model-based embedded software product line. *Innovations in Systems and Software Engineering,* Volume 8, pp. 35-49.

Pretschner, A., Broy, M., Kruger, I. H. & Stauner, T., 2007. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering (FOSE '07)*, pp. 55-71.

Psutil, 2020. *psutil documentation.* [Online]
Available at: https://psutil.readthedocs.io/en/latest
[Accessed 20 08 2021].

RabbitMQ, 2021b. *AMQP 0-9-1 Model Explained.* [Online]
Available at: https://www.rabbitmq.com/tutorials/amqp-concepts.html
[Accessed 18 08 2021].

RabbitMQ, 2021. *RabbitMQ.* [Online]
Available at: https://www.rabbitmq.com/
[Accessed 17 08 2021].

Rich, N. & Holweg, M., 2000. *Value Analysis- Value Engineering,* Cardiff: Lean Enterprise Research Centre.

Rutten, B. & Cobbenhagen, R., 2019. Future Trends in Electric Vehicles Enabled by Internet Connectivity, Solar, and Battery Technology. In: Y. Dajsuren & M. v. d. Brand, eds. *Automotive Systems and Software Engineering State of the Art and Future Trends.* Cham: Springer Nature Switzerland AG, pp. 323-346.

Saaty, T., 2008. Decision making with the analytic hierarchy process. *Int. J. Services Sciences,* 1(1), pp. 83-98.

Šandor, R., Stepanović, M., Bjelica, M. & Samardžija, D., 2018. Vehicle2X communication proposal for Adaptive AUTOSAR. *International Conference on Consumer Eletronics,* Volume 8, pp. 1-3.

Schmidt, K., 2007. Model Driven Development of Automotive Software: Present State and Future Requirements. *Workshop on Object-oriented Modeling of Embedded Real-Time Systems*, 14 10.

scikit learn, 2020. *User Guide.* [Online]
Available at: https://scikit-learn.org/stable/user_guide.html
[Accessed 23 08 2021].

Selic, B., 2006. Model-Driven Development: Its Essence and Opportunities. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing,* Issue 9, pp. 1-7.

Singh, S., Sheng, Q., Benkhelifa, E. & Lloret, J., 2020. Guest Editorial: Energy Management, Protocols and Security for the Next Generation Networks and Internet of Things. *IEEE Transactions on Industrial Informatics,* 16(5), pp. 3515-3520.

Singh, V., Durra, K. & VanderMeer, D., 2013. Estimating the Energy Consumption of Executing Software Processes. *IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 94-101.

SQLite Consortium, 2021. *SQLite.* [Online]
Available at: https://www.sqlite.org/index.html
[Accessed 09 02 2021].

Staron, M., 2017. *Automotive Software Architectures: An Introduction.* Stuttgart, Germany: Springer, Cham.

Stojanovic., Krunic, M., Cetic, N. & Lukic, N., 2019. Source code generators for ADAS feature deployment in context of ROS and adaptive AUTOSAR applications. *Telecommunications forum TELFOR,* Volume 27, pp. 1-4.

Thomas, J., Dziobek, C. & Hedenetz, a. B., 2011. Variability Management in the Autosar Based Development of Applications for In-Vehicle Systems. *Workshop on Variability Modeling of Software-Intensive Systems,* Volume 5, pp. 137-140.

Torchiano, M. et al., 2012. Benefits from modelling and mdd adoption: Expectations and achievements. *International Workshop on Experiences and Empirical Studies in Software,* Issue 2, pp. 1-6.

Ubuntu Manuals, 2017. *stress-ng - a tool to load and stress a computer system.* [Online]
Available at: https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html
[Accessed 23 08 2021].

Vector Informatik GmbH, 2018. *Vector E-learning.* [Online]
Available at: https://elearning.vector.com/mod/page/view.php?id=445
[Accessed 14 01 2021].

Vector Informatik GmbH, 2020. *Best Trace Format (BTF) Technical Specification - Version 2.2.0,* s.l.: s.n.

Voss, S. & Eder, J., 2018. Handling System Complexity in sCPS: Usable Design Space Exploration. *International Workshop on Software Engineering for Smart Cyber-Physical Systems,* Volume 4, pp. 2-5.

Wise, P., 2009. *iotop - simple top-like I/O monitor.* [Online]
Available at: http://manpages.ubuntu.com/manpages/bionic/man8/iotop.8.html
[Accessed 24 08 2021].

Wolff, C. et al., 2015a. Automotive software development with amalthea. *Project Management Development – Practice and Perspectives*, 04, pp. 432-442.

Wolff, C. et al., 2015b. AMALTHEA – Tailoring Tools to Projects in Automotive Software Development. *The 8th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 24-26 September, pp. 515-520.

Woodall, T., 2003. Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis. *Academy of Market Science Review,* Volume 12.

Xerial, 2020. *SQLite JDBC Driver.* [Online]
Available at: https://github.com/xerial/sqlite-jdbc
[Accessed 21 08 2021].