FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Evaluation and Prototypical Implementation of Machine Learning to Detect ECU Misbehaviour

Pedro Alves de Almeida

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: André Monteiro de Oliveira Restivo

Supervisor: Marc Weber

September 22, 2017

Abstract

The automotive industry had been evolving at a relatively stable pace until recently when a major shift in the technology evolution happened. Vehicles started to receive more and more electronics as time went by, and more recently, the industry began to look into autonomous vehicles. At the same time, with the increase in electronic components, the use of wireless technologies rose in popularity, and the idea of creating a global network started to come into being.

The problem arises when both of these technologies merge and are implemented simultaneously. Suddenly there is no need to have a human driver, but the car can still be remotely accessed. This remote access means that the car can be treated as a complex computer and can be hacked by a third party which, if successful, can gain control of the vehicle.

With this problem in mind, the industry recently started to look for solutions for this problem, and it is in that context that this dissertation is inserted. This dissertation is part of a larger project proposed by *Karlsruhe Institute für Technologie*, and the proposed solution tries to monitor the in-vehicle network as well as the behaviour of all the Electronic Control Units. This dissertation implements a prototype capable of monitoring the Electronic Control Units.

The monitoring is done on a software level with the help of Machine Learning algorithms. These algorithms try to learn the standard behaviour of the Electronic Control Unit's software and then, based on the training, try to detect anomalies that represent a deviation from the normal behaviour, thus, being a misbehaviour of the Electronic Control Unit.

The developed algorithm joins two different algorithms, Decision Trees and Graphs, to learn the correct sequences and subsequences in which the different components of the software run. If a sequence that is detected deviates from the profile learned during the training phase, it will be recorded and later will part of an error log containing all the identified anomalies.

The algorithm was implemented and tested, obtaining an accuracy of 92,4%. It was also demonstrated that, given more training data to the algorithm, this accuracy could improve, becoming closer to 100%. These results are promising because, with the increase in connectivity, it will be possible to obtain large amounts of data that can be used for both training and testing, which results in an improvement of the algorithm, thus, making the vehicle more secure.

In a final step, the previously developed algorithm was improved, by allowing the creation of multiple structures for each task that had Runnable Entities and each Runnable Entities that performed API Calls. Therefore, having an hierarchy of several similar structures that not only allows the algorithm to know sequences and subsequences, but also causal relations between tasks, Runnable Entities and API calls.

Acknowledgements

First and foremost, I would like to thank both Universities, Faculdade de Engenharia da Universidade do Porto and Karlsruhe Institute für Technologie, for providing me with the means and opportunity to work and do my dissertation.

Then, I would like to thank my advisers. Herr M. Eng. Marc Weber, for not only giving me the opportunity to work with him, helping me evolve, both personally and at a professional level, and for all the advises given throughout this thesis. I would also like to thank Professor André Restivo, for all the suggestions and advises given throughout this thesis, for motivating me and helping me see further into the project.

I am also thankful Herr Dr Professor Eric Sax, for facilitating this opportunity and for captivating my interest into the field during his classes.

I would like to thank my colleague Daniel Grimm for the help given and all the clarifying discussions.

A big thank you to both Ricardo Gomes and Vasco Patarata, for not only putting up with me and all my quirkiness during these five years but also for being there for me when I needed.

I am thankful for having spent these five years along with André Coelho, Catarina Simões, Paulo Luís and Vítor Mendes, for all the projects done together, the good moments that I will cherish, and for being there when I needed the most.

I would like to thank my aunt Maria José Almeida for the advises given, the calls, and time spent with me during this work.

I would like to thank my family, my mother, my father, and my sister, for everything, all the help, time, and patience that they had not only during this thesis but during my life.

Finally but not least, I would like to thank my grandfather, José Maria Almeida, for all the good moments that we spent together, for inciting my curiosity and for helping me become who I am today.

"Nous sommes des nains juchés sur les épaules de géants, nous voyons plus qu'eux, et plus loin, non que notre regard soit perçant, ni élevée notre taille, mais nous sommes élevés, exhaussés, par leur stature gigantesque."

Bernard De Chartres

Contents

List of Figures

List of Tables

Abbreviations and symbols

ABS	Anti-Lock Breaking System
ECU	Electronic Control Unit
ML	Machine Learning
CAN	Controller Area Network
SVM	Support Vector Machine
OCSVM	One Class Support Vector Machine
E/E	Electrical/Electronic
AUTOSAR	AUTomotive Open System ARchitecture
OSEK	*Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*
BSW	Basic Software Layer
API	Appli- cation Programming Interface
RTE	Runtime Environment Layer
APL	Application Layer
SW-C	Software-Component
VFB	Virtual Function Bus
RE	Runnable Entity
BswS	Basic Software Scheduler
OS	Operative System
ISR	Interrupt Service Routine
FIFO	First In First Out
DC	Direct Current
NN	Neural Network
DDoS	Denial of Service Attack
SE	Supervised Entity
MC	Managing Core
MDT	Merged Decition Tree

Chapter 1

Introduction

1.1 Context

Currently, two upcoming trends are the primary incentive for the development of the automotive industry, autonomous driving and connectivity. Regarding safety and security, these two areas are contradictory. On the one hand, as the industries evolve towards a fully autonomous vehicle, the electric and electronic components have a bigger influence on the behaviour of the vehicle ending with the full control of all the driving functions. Initially, systems like the Anti-lock Braking System(ABS) or Cruise Control, were only able to assist the driver with the driving [1]. As technology developed these functions got more complex, and today the autonomous car has already reached the market [2]. Albeit it is still possible to drive it, the drivers are advised to keep their hands on the wheel at the whole time. However, this progress also brings some downsides. The drivers might start to unlearn how to drive or get careless when behind the wheel.

On the other hand, connectivity means that all vehicles are starting to create a global network that allows them to communicate with one another [3] and even update their software *over the air*, that is via wireless [4]. As it already happens with computers, this opens doors to hackers because, since if the vehicles are connected to the "outside world" someone might try to get in and hack the system. For this, all it needs to happen is something as small as changing or altering one message in the in-vehicle network as demonstrated by Hoppe *et al.* [5]. Hoppe *et al.* also showed possible consequences of these attacks [5]. Koscher *et al.* [6] also studied how an automobile can be controlled by an outside entity and completely ignore the driver's input. More recently, a group of people also reached the same results with a car from the *Jeep* manufacturer [7]. They not only gained access to the car but they were able to shut down the vehicle completely.

1.2 Motivation

With the rise of the problems mentioned in the previous Section 1.1, there is a need to come up with a solution not only to prevent but also mitigate them. Such a solution would allow a

smooth integration between the two areas, autonomous driving and connectivity. Several possible solutions are being tested and evaluated, and those will be approached in Chapter 3.

Of all the solutions, the one that will be studied in this dissertation revolves around the monitoring of an Electronic Control Unit (ECU) of a vehicle. Currently, a single car has not one but several ECUs that are part of what is called the in-vehicle network [8]. An ECU is the unit responsible for monitoring and controlling a specific part of the vehicle such as the motor, suspension, ABS, and others. Given this fact, the solution proposed in this dissertation opts to monitor one ECU using software that will be connected to it. The software will monitor in real time the behaviour of the ECU and will be looking for any anomalies in the behaviour. It will also create an error log that will allow the manufacturer to study the misbehaviour and fix the cause of such misbehaviour. Regarding the behaviour of the ECU, three possible methods can be used:

1. Compare the actual behaviour of the ECU with a database that contains data describing how it would normally operate;
2. Evaluate the monitored behaviour with a set of previously established rules that together define what a normal operation is;
3. Use Machine Learning (ML) algorithms to learn what is the proper behaviour of the ECU and, autonomously, detect any malfunction.

This dissertation will study the third method, and to facilitate the choice of which algorithm should be implemented, a state-of-the-art study will be made.

The use of an intelligent software presents several advantages over the other two methods.

1. It is independent of the manufacturer of the vehicle. Due to the intelligent characteristics of the software, it will be able to adapt itself to the ECU. It does not depend on a database or the connection to it, and it is also independent of any rules.
2. It uses an "intelligent" algorithm which means it is better suited to defend itself against unforeseen problems.
3. It is based on ML, which means that the software is able to adapt and evolve with new data and then be updated to the vehicle;
4. It has no hardware which means it is easier and cheaper to create redundancy. This redundancy translates to a safer program without any additional costs;

1.3 Objectives

As mentioned in Section 1.2, this dissertation has the primary objective of using and studying Machine Learning algorithms to detect ECU anomalies that represent misbehaviour.

Definition 1.1: *An anomaly is something that deviates from what is standard, normal, or expected.*

As it was also mentioned in Section 1.2, a state-of-the-art study will be made, not only to find what is currently being done but to choose the algorithm that better suits the problem and later implement it.

The algorithm will be implemented, and it will be adapted to work with the communication protocol Controller Area Network (CAN). To help with the integration, a program made by Vector Informatik called CANoe [9] will be used. CANoe allows the simulation of a system composed of one or several ECUs. The program was made with the purpose of developing and testing ECUs which is the main reason why it was chosen.

After the integration, the algorithm will be trained to identify misbehaviours in the simulations. A function responsible for the creation of an error report every time the program detects an anomaly will also be implemented. The report has the purpose of helping the manufacturer in finding the root of the misbehaviour.

In the final phase, the program should be not only able to detect misbehaviour, but also be able to create them. This way, the program would have the possibility of training itself. As in the previous phases, the program should also detect the attack and create an error log.

The objectives for this dissertation are:

- Implement a ML algorithm;
- Simulate an ECU;
- Simulate an in-vehicle network with several ECUs;
- Integrate the algorithm with the simulated in-vehicle network;
- Train the algorithm;
- Add the error log generation function to the algorithm;
- Add the error generation function to the algorithm.

1.4 How to read this dissertation

This dissertation has six more chapters.

The next chapter, Chapter 2, contains the theoretical foundations that support and help understand the following chapters and the whole dissertation.

Chapter 3 contains an overview of the state of the art. The state of the art is a review of what is currently being done and used in the same fields and subjects of this dissertation to solve similar problems.

Chapter 4 presents a more detailed description of the problem that motivated this dissertation. It also describes the structure of the chosen solution and the system that originates from said structure.

Chapter 5 enumerates all the steps taken to achieve the final results and how all the parts of the system were implemented.

Chapter 6 presents the final results, as well as a discussion of the same results.

The final chapter, Chapter 7, concludes this dissertation and mention future steps, not only small improvements that can be done to the work presented in this dissertation but also how the chosen and implemented solution can be used in other future projects.

Each chapter has a similar structure. It has a small introduction describing the structure of said chapter accompanied by a summary of the chapter.

Chapter 2

Foundations

2.1 Introduction

This chapter will introduce the reader to the theoretical concepts used in this dissertation, giving the reader a better understanding of the following chapters.

It is divided into two main sections, the first section, Section 2.2 regarding the software used in today vehicles, and the second section, Section 2.3, exploring the field of Machine Learning (ML). Both sections will start with a brief historical context.

Section 2.2 then goes into more detail of the software present in the Electronic Control Units (ECUs) present in a vehicle. Starting by explaining the structure of the software, then focusing on each of the three main parts of the software. Each part will have an introduction to the working principle and main functions of said part.

After the historical context, Section 2.3 will try to explain what is the field of ML. Then comes an introduction to the family of algorithms known as Decision Trees. Starting with an introduction to the structure and working principle of the Decision Tree and then with a more detailed explanation of one of the most used algorithms responsible for building such Trees. It will then introduce the concept of Graphs, what is a Graph and how does a Graph work, terminating with three different types of Graphs. The section will end with an introduction to Support Vector Machines (SVM) the third and last type of algorithms relevant for this dissertation. Starts with a brief introduction of what is a SVM, followed by their working principle and ending with the introduction of a specific type of SVM, the One Class Support Vector Machine (OCSVM).

2.2 AUTOSAR

2.2.1 History of AUTOSAR

With the evolution of computers, and therefore the increase in computing power, the automotive industry started to make use of such resources leading the automobile industry to evolve alongside the computer. The integration of computing units, ECUs, in a vehicle allowed for a greater control of certain actions and parts within the vehicle and made possible the use of additional functionality

as explored in the Chapter1. But this brought more complexity to the vehicle, and with the recent escalation, a problem arose, how to control a complex Electrical/Electronic (E/E) architecture.

To solve this problem, a group of vehicle manufacturers and Tier 1 suppliers, those who directly supply the vehicle manufacturers, joined in a partnership to create a set of standards and develop software that would be able to control the complexity of the E/E architecture and, at the same time, be adaptable and modular. The motivation behind this partnership was, to handle the increase in the E/E complexity, the flexibility of a product concerning modifications, upgrades and updates, the improved quality and reliability of E/E systems and the scalability of solutions within and across product lines [10]. The goals were to fulfil future vehicle requirements, such as availability and safety, software upgrades/updates and maintainability. Increase the scalability and flexibility to integrate and transfer functions, to have a higher penetration of *Commercial off the Shelf* software and hardware components across product lines, to improve the containment of product and process complexity and risk and to optimise the cost of scalable systems [10].

Thus, in July of 2003, the AUTomotive Open System ARchitecture (AUTOSAR) was created with the purpose of creating such standards and software [11].

However this partnership was not the first of its kind and follows an already previously existing partnership, *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen* or OSEK. AUTOSAR's software Operative System located in the Basic Software layer is based on the Operative System from OSEK.

OSEK was a set of standards and specifications developed by the German partnership, and it also produced an Operative System, with the same name, to help manage the Basic Software Layer of the, at the time, software present in the ECUs [12].

So the software present in an ECU could be represented as in Figure 2.1. To help visualise the differences, on the right, Figure 2.2 is a picture illustrating an ECU with AUTOSAR as software.

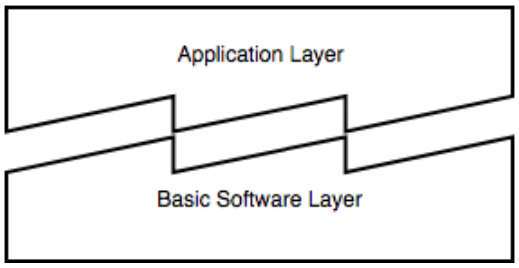

Figure 2.1: Structure of the Software of an ECU without AUTOSAR

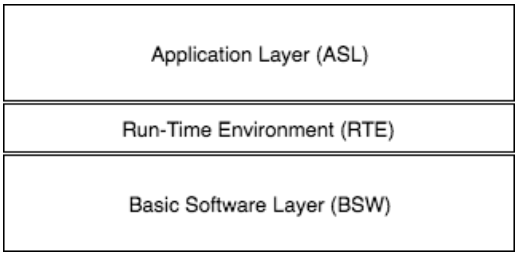

Figure 2.2: Structure of the Software of an ECU with AUTOSAR

Today, AUTOSAR is a widely accepted software, and there are several companies which develop their specific versions of software that can be implemented in an ECU. Companies like ETAS [13], Vector Informatik [14], Elektrobi [15] and Mentor [16] all develop and sell their own versions.

2.2.2 Technical Overview

As mentioned in Section 2.2.1, the software was developed, and the resulting software provides a set of standard interfaces that work in a wide range of vehicles from different brands and which purpose is to, not only control the E/E architecture of such vehicles but to do so with optimal performance.

To control the entire E/E system of a vehicle, the software needs to be able to interact with every individual component (sensors, actuators, ECUs, and so on) and provide means for these components to communicate and interact with one another. For example, to transmit a signal from a sensor, process the obtained information and perform an action based on that information through an actuator.

To achieve this objective, the developed software has three different main layers, therefore is referenced as a layered software, where each layer has its structure and functions.

The layer that is closer to the ECU and interacts almost directly with it is the Basic Software Layer or BSW. This layer provides a set of services and modules, each with its functionality, to the applications, other modules, Application Programming Interfaces (APIs). Therefore it is the layer responsible for the communication and control of the microcontroller that controls the ECU.

Then comes the middleware, the Runtime Environment Layer or RTE which, in simple terms, allows and establishes the communication between the other two layers. The RTE is the main innovation in the AUTOSAR software since it allows the indiscriminate development of the applications in the Application layer, without the concern of the hardware in which they will run. Therefore the RTE acts as a driver for the Application layer and allows the use of the ECU Operative System and its Basic Software Modules. This grants the AUTOSAR software the flexibility that was one of the consortium's main goals.

Finally, the "upper" layer is the Application Software Layer or APL. This layer contains the Software Components (SW-Cs) that command the BSW, by communicating with the RTE which in turn communicates with the BSW and make use of the ECU services. It is also the layer that permits the communication between ECUs.

Due to the layered system, it is possible to define two different views into the system, the Virtual Function Bus, or VFB, view and the System view.

The first view comes from the fact that there is a separation between the ASL and the BSW. This separation is physically represented by the RTE and by the VFB regarding the software.

The VFB is a virtual BUS system that links and allows the virtual communication between SW-Cs and between the SW-Cs and the Basic Software Modules, present in the BSW. These SW-Cs are the basic units of the Application Layer and will be further explored in Section 2.2.2.1. The VFB does not exist physically speaking. Instead, it specifies to the system how and through which tools can the SW-Cs communicate. This allows for a SW-C to be developed disregarding the vehicle and brand in which it will be implemented, complying with one of the goals that were adaptability. It also makes it possible for a SW-C to be used in different projects as long as it follows the standards and norms of AUTOSAR.

The System view is the view point from the Basic Software layer. It sees the operating system existent in an ECU and the services and modules present in the ECU as well as the communication with the RTE and the microcontroller. It is a more technical and low level view and it can vary from ECU to ECU [17] [18] [19] [20] [21] [22].

The overall picture of the software can be seen in Figure 2.3 [20]. Here we see in the upper level the VFB where the connections between the SW-Cs are established and defined. Then we have the combination of these definitions with the ECUs' descriptions which will allow for the generation of the RTE and the mapping of the SW-Cs to the several ECUs. Finally, it is possible to see the physical system, the ECUs and the BUS systems that allow the inter-ECU communication, and their software divided into the three layers.

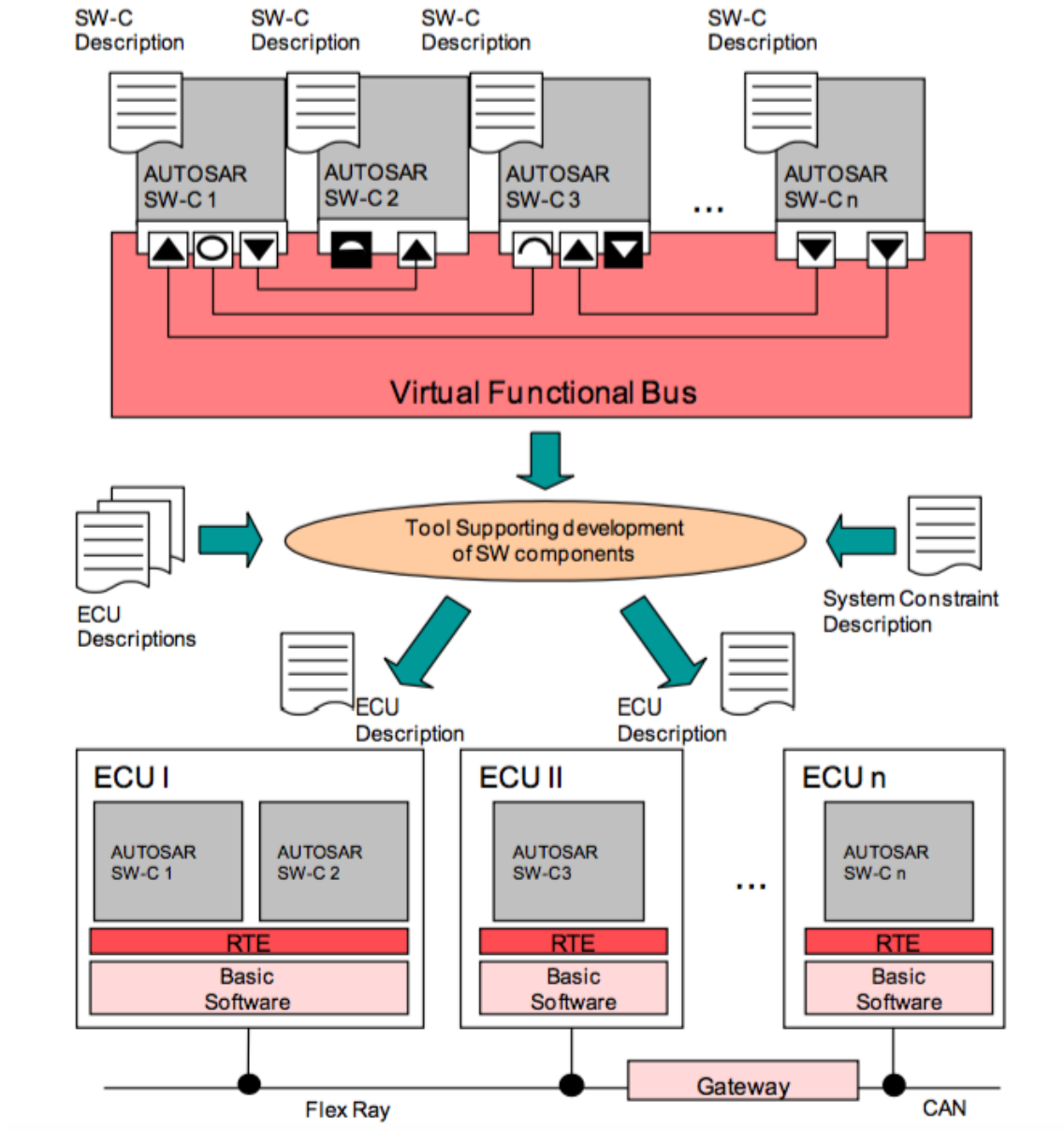

Figure 2.3: AUTOSAR software

To give a better perspective of the layers, Figure 2.4 [23] represents the point of view of a single ECU.

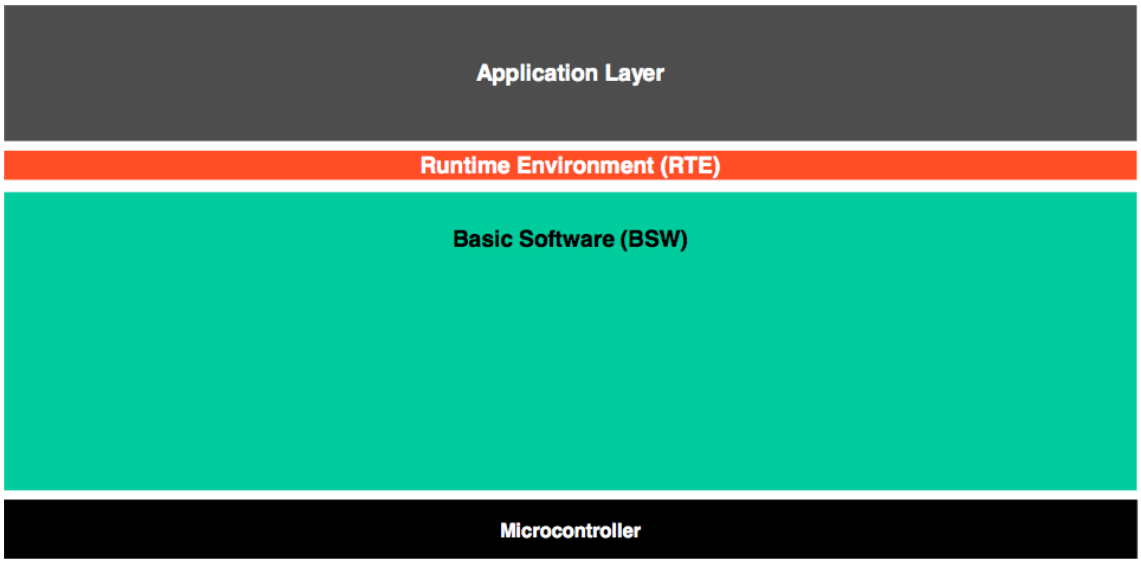

Figure 2.4: AUTOSAR layered system from the view point of an ECU

2.2.2.1 Application Software Layer

As seen in Figure 2.4, the Application Software Layer can be considered to be the upper layer regarding the ECU microcontroller, since it is the most distant at a software level.

As previously mentioned, this is where the applications exist. An application can be perceived as a program, or a partial program, that has a set of functions or instructions, each one with their purpose. Therefore, in the context of this dissertation, we can define an application as:

Definition 2.1: *An Application is a group of Software Components that represent and are responsible for a specific task.*

An example of an application can be the control of the throttle's position in the engine.

As previously mentioned, an application is made of SW-Cs. These are the building blocks of the ASL and therefore are the active part of the application. The fact that the ASL is completely divided into SW-Cs makes it a component based layer. Applications can have in its internal structure a single SW-C or multiple SW-Cs and the ASL of an ECU can, likewise, be made of a single application or have several applications using that ECU.

A SW-C can be described as a stand alone component, which means it can exist by himself, or it can be made of other SW-Cs, with a defined hierarchy in its structure.

This brings us to the two main types of SW-Cs. The first type, when a SW-C cannot be further divided and is already in its most basic form, it is of the type *AtomicSwComponentType*. An Atomic SW-C can also be defined when a SW-C is mapped to a single ECU. This means that the SW-C and its instructions only communicate and use the resources of a single ECU and therefore

it is not possible to further divide the component. But an *AtomicSwComponentType* can be reused, meaning that an *AtomicSwComponentType* can be found in more than one ECU.

Definition 2.2: *An AtomicSwComponentType is a Software Component that can no longer be divided into smaller Software Components.*

On the other hand, when a SW-C is made of atomic SW-Cs, it is of the type *Composition-SwComponentType*. It is important to note that this type of SW-C is only a logical and hierarchical representation of the links between the Atomic SW-C and thus have no special representation when it comes to coding. Unlike the Atomic SW-C, the Composition SW-C can be mapped to several ECUs, since it uses more than one Atomic SW-C and they may be in different ECUs.

Definition 2.3: *A CompositionSwComponentType is a SW-C made of one or more AtomicSwComponentType and is not restricted to a single ECU.*

As mention above, an Atomic SW-C is one of the two main types of SW-C, but there are several sub-types of Atomic SW-Cs, each with its specific functionality. There are Application SW-Cs, which represent hardware-independent software, Memory SW-C which defines which non-volatile data, data that is not lost without power, can be shared to other SW-Cs, Sensor/Actuators SW-Cs, links the software representation of a sensor or actuator to its hardware through the ECU, among others.

To be able to communicate with the RTE or other SW-Cs (through the RTE), each SW-C has a set of well-predefined ports, each one with its own and specific function. Depending on the function, the type of port varies. The type of port is also dependable on the type of communication in use. The communication aspect and ports will be further explored in Section 2.2.2.2.

A SW-C is also made of their basic unit, which is called Runnable Entities (RE), or just runnables. A SW-C can have one or more runnables in its internal structure. This can be seen in Figure 2.5 [18].

A runnable is the most basic element in the ASL and represents a single function. This function can be a single instruction or a set of instructions, both cases inside a C-function, that represent the intentions of the SW-C. Therefore they are the active parts of the SW-Cs.

Definition 2.4: *A Runnable Entity is a single function and is the building block of the Software Components.*

So a runnable is a piece of code that represents an instruction and therefore needs to run to fulfil its purpose. The runnables are mapped to OS tasks, and since it is the OS that dictates when a task is activated, it is also the OS that says when a runnable runs, and it does so by using the RTE to emit an event signal that triggers the runnable.

When the program that will run in the ECU is being generated, there are some specifications that are used as inputs to the generation process, which can be seen in Figure 2.3. This generation occurs when the Basic Software Layer and the RTE are being generated, and the SW-Cs that will be part of the Application Software Layer are being assigned to the ECU. In these specifications,

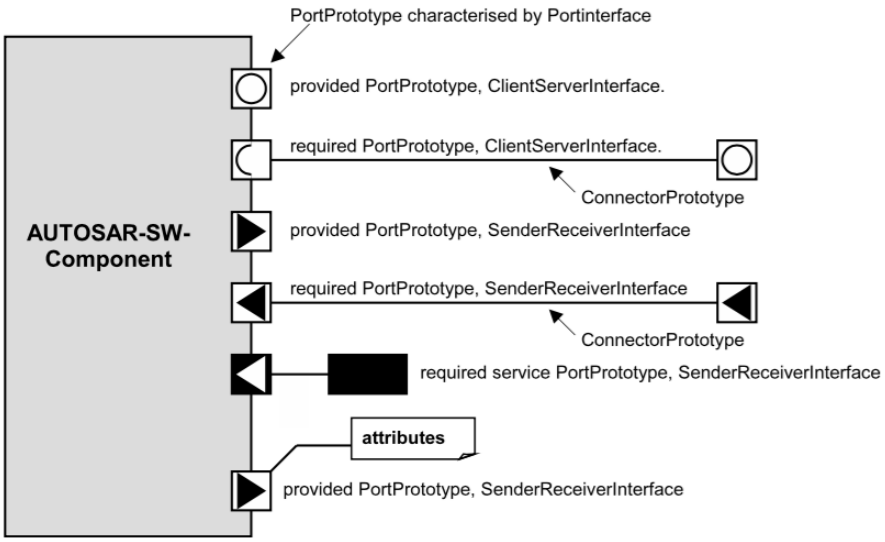

Figure 2.5: Representation of a Software Component

information about the runnables and the SW-Cs is present. Information like what resources the runnable would like to use, hence to which ECU it should be mapped, the order in which the runnables should run. It is also in this generation process, more specifically, during the generation of the RTE, that the RTE allocates tasks from the Operative System and maps these runnables to the tasks with a defined order, present in the configuration. These tasks can be thought as a stack in which the runnables are "placed". The connection between the runnables and the tasks is done by the RTE when, during the generation process, the RTE creates the task bodies, a piece of code that "glues" the runnable and the task. Once the program has been generated, and the system is running, the tasks will be activated by the OS, and consequently, the runnables that were mapped will run in an orderly manner, each one triggered by the RTE.

To summarise, an application represents one or more tasks, not to be mistaken with the Operative System tasks, and is comprised of one or more SW-Cs which are the active part of the application since they carry the instructions. In turn, a SW-C is made of one or more runnables which are its active part, and each one represents one function that will allow the carry of the instructions of the SW-C. These runnables are implicitly mapped, since it is the SW-Cs that are mapped to the ECUs, to an ECU and to its Operative System which will assign them an order in which they will run based on previously defined configuration [17] [18] [19] [20] [21].

In Figure 2.6 [20], a possible structure of an Atomic SW-C and two runnables, MainCyclic and Setting can be seen.

2.2.2.2 RTE

The Run-Time Environment is the middle layer of the AUTOSAR architecture. Its location can be seen in both Figures 2.3 and 2.4. In simple terms, it is, partially, the physical representation of the specifications of the VFB for an ECU. Together with some modules, present in the Basic

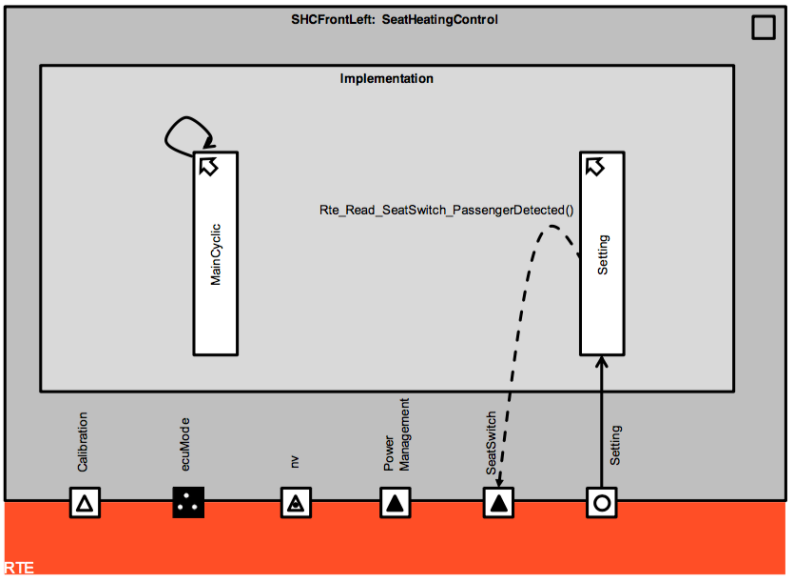

Figure 2.6: An Atomic SW-C with its ports and runnables

Software Layer, the representation of the VFB is complete. Although the Application Layer is independent of the ECU, as previously seen, the RTE is not. This means that when an ECU is being configurated, a tool called RTE generator will generate the RTE from the VFB specifications together with the ECU specifications. It is also during this configuration process that the SW-Cs are mapped to the ECUs as stated in Section 2.2.2.1.

The Run-Time Environment, RTE, implements the AUTOSAR VFB interfaces and thus is responsible for the communication between SW-Cs and the communication between SW-Cs and the Basic Software Layer. For the SW-Cs to communicate, they have in their structure ports, and the RTE establishes the connection between those ports. The communication can be of two major types, Sender-Receiver or Server-Client. The Sender-Receiver communication paradigm uses three types of ports, a require-port that only receives data, a provide-port that only sends data and a require-provide-port that is capable of both sending and receiving data. This communication paradigm is used when the signals that are to be transmitted consist of atomic data elements, thus being a single parameter. It is also used for signal groups.

The Client-Server communication paradigm uses a client which initiates the communication by requesting a service from the server. The server then provides the service and emits a response to the client. This communication paradigm also uses require-ports, provide-ports and require-provide-ports. The difference is they are specific for Client-Server communications.

Depending on the SW-C or the Basic Software Module, the port can be of several sub-types, each with their specific functionality and interface.

For the communication between SW-Cs and the Basic Software Layer, the RTE can be perceived as the translator between both layers. This characteristic allows, together with the Basic Software Layer, the SW-Cs to be ECU independent and gives them the important characteristic of being reusable. Due to this SW-C independence, the RTE makes sure that, regardless of the com-

ponent's location, the system runs as expected. As seen in the communications between SW-Cs, it is also possible to find here the same communication paradigms and the same ports.

Another major function of the RTE is the scheduling of the SW-C and consequently its constituents, the runnables. Scheduling means that the RTE dictates when and which runnable shall run in the ECU and thus creating a queue of instructions with a defined order. This scheduling allows for a smooth operation of the system and, if everything runs accordingly, avoids unexpected behaviours such as SW-C running and using the same resources simultaneously without permission. To schedule the SW-C, and to be more precise, their runnables, the RTE uses what are called *RTEEvents*. These are the signals that are transmitted by the RTE to the specific runnable and tell the runnable that it can start its instructions. The *RTEEvents* can be used in a *ScheduleTable*. A *ScheduleTable* is a table with runnables, their order and their respective time-stamps that specify when the runnable should start to run.

During the generation process of the RTE, the Basic Software Scheduler (BswS) is also generated. Since they're generated simultaneously and share resources, the Basic Software Scheduler can be considered part of the RTE. The function of the BswS is similar to the scheduling function of the RTE but, instead of scheduling runnables, the BswS schedules the schedulable entities of the Basic Software Modules, present in the Basic Software Layer. These modules are the entities responsible for providing the services to the runnables and consequently to the applications [17] [18] [19] [20] [21].

2.2.2.3 Basic Software Layer

The Basic Software Layer is the last of the three main layers of AUTOSAR. It is the one *closer* to the microcontroller and thus it is the one responsible for the direct communication with it.

The Basic Software Layer is responsible for communicating with the microcontroller, and it creates the ECU abstraction which, together with the RTE, allows for the SW-Cs to be developed without being concerned in which ECU they will be implemented. It is also in the Basic Software Layer that the system disposes of a set of services, such as the Operative System, Memory services, Module managers, among others.

With the functionality of this layer in mind, it is possible to define layers within the Basic Software Layer. They are the Microcontroller Abstraction layer, which contains drivers that permit the access to the microcontroller by the software. These drivers are software modules with the function of acting as a translator between a specific hardware and software. Then there is the ECU Abstraction Layer, which, similarly to the Microcontroller Abstraction Layer, has an API that gives the other layers and sub-layers the possibility to access and communicate with the ECU. Finally, there is the Services Layer which, as previously stated, contains services that can be used by the applications, the RTE and other Basic Software Modules. In the Services Layer, it is possible to find the Operative System, Basic Software Modules Managers, such as the Communication Module Manager, Memory Module Manager, ECU State Manager, the Watchdog Manager and Diagnostic Services. The implementation of this layer is mostly ECU and microcontroller dependent.

Figure 2.7 [17] shows the AUTOSAR layered system with the new sub-layers in the Basic Software Layer.

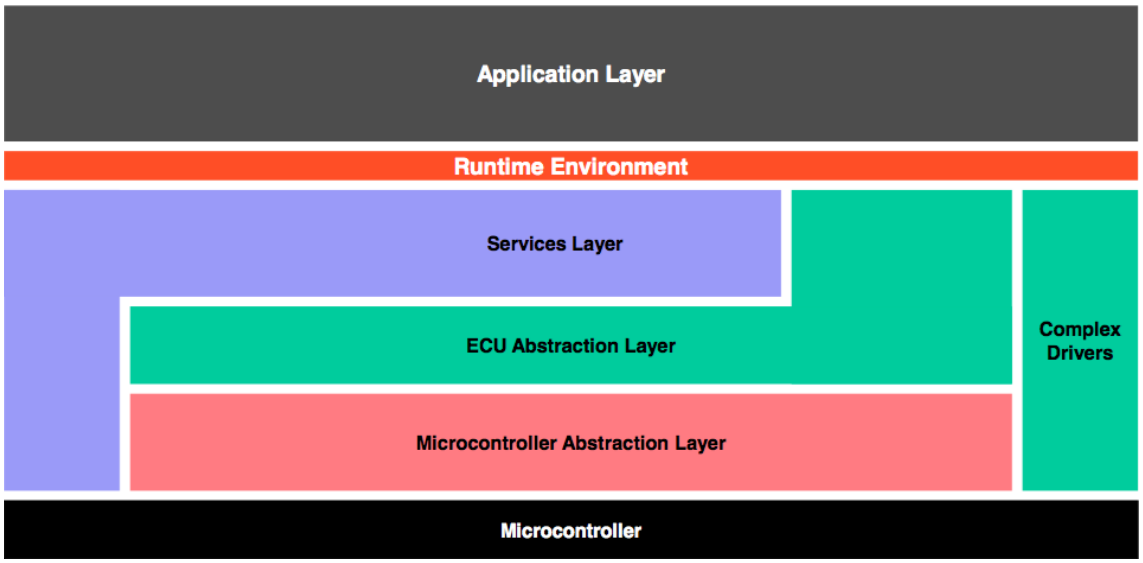

Figure 2.7: AUTOSAR Layered system and the sub-layers of the Basic Software Layer

As mentioned above, the Basic Software Layer is divided into sub-layers and inside this sub-layers are the modules. These modules are the components of the sub-layers, each sub layer with its modules and each module with its functionality. These modules can be defined as a group of software files that represent a functionality present in a specific ECU. So the drivers, the Module Managers themselves are all modules found in the Basic Software Layer of an ECU.

Some of the most important modules are the module managers, which include the *COMManager*, responsible for the communications, the *BSWModeManager*, responsible for switching between modules, *NVRAMManager*, responsible for managing the memory, *CryptoServiceManager*, responsible for managing the security modules. There is also the Operative System which can also be considered a module, and the *WatchdogManager*, responsible for monitoring the program flow and time constraints. The *WatchdogManager* will be explored in Chapter 3

Each module has its schedulable entities, which can be compared to the runnable entities present in the application layer. These Basic Software Schedulable Entities are what the BSW Scheduler schedules and are what executes the modules services.

As previously mentioned, it is in the Basic Software Layer, and more specifically in the Services Layer that the Operative System runs. The Operative System (OS) in AUTOSAR is based on OSEK/VDX. OSEK/VDX is an OS that resulted from the partnership by the same name. It has been used in ECUs for a significant amount of time and has been proven to be an efficient OS. Consequently, the AUTOSAR OS uses the same standards and principles of OSEK/VDX.

The AUTOSAR OS is an event-triggered based OS. This means that every task or Interrupt Service Routine (ISR) is triggered, enters the run state, by means of an event produced by the OS. This type of OS has its advantages. It allows for a well-defined hierarchy for the tasks and ISRs, which facilitates their scheduling. It eases the use of ISRs due to this hierarchy and its implied

priority, an ISR can have a higher priority relative to a task and thus it can interrupt the execution of such task. It also provides some protection against the incorrect use of the OS services. The OS also provides a set of APIs and a defined functionality which makes it easier to change from one ECU to another and therefore be reusable considering the applications.

The OS serves as a basis for the runnables, which exist in the Application Software Layer, and therefore provides the runnables with an environment on a processor, more specifically the microcontroller and the ECU. To do so, the OS uses what as already been introduced as Tasks. A task is a framework, structure, to which the runnables are mapped. This framework has a priority associated which will define the order in which the task is executed in the eventuality of several tasks being in the wait state. The lowest priority is represented by the number 0 and the higher the number, the higher the priority and the sooner the task will run.

Definition 2.5: *An Operative System Task is a queue with a priority associated to which Runnable Entities are mapped and executed according to their position within the Task.*

It is also possible to distinguish two types of tasks. The Basic tasks and the Extended tasks.

The first has three possible states. The running state, which means that the task is currently running, hence using the processing power available, the suspended state, which means the task is suspended and therefore is passive, and the ready state. A task enters the ready state when the necessary conditions for the task to run are met but the processor is not yet free, and so the task is in the ready state waiting for the processor to be free.

The second type of tasks, the Extended task, has a fourth state, the waiting state. The waiting state allows for the task to liberate the processor and for another task with a lower priority to enter its running state.The several states of the tasks can be seen in Figures 2.8 [12] and 2.9 [12]

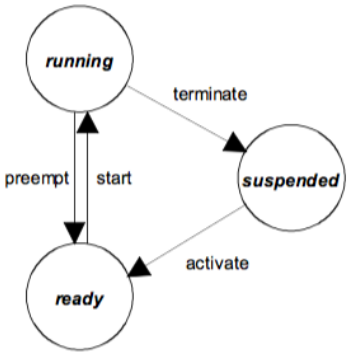

Figure 2.8: Basic Task states

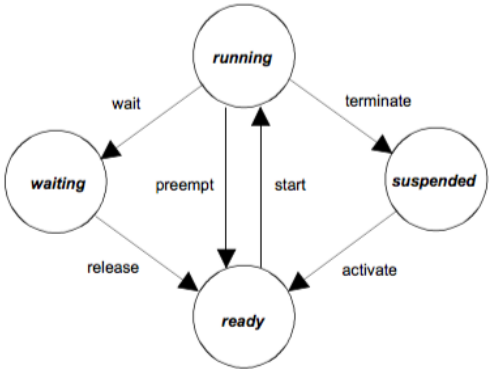

Figure 2.9: Extended Task states

As it was explained with the runnables, the tasks require scheduling. The scheduling can vary and be done with different methods. This scheduling deals with the case of one or more tasks having the same priority and thus a necessity of defining which of these tasks will enter the running state first. A possible solution is the First In First Out (FIFO) method, which dictates that the first task of priority n to appear is the first to run. A task is activated through OS Services. These can

either be *StartTask* or *ChainTask*. The task ends by itself, which means once it has done all its instructions it auto-terminates by using the service *TerminateTask* or it can be forced to terminate in the eventuality of a malfunction.

An example of how the tasks run and are executed can be seen in Figure 2.10 [12].

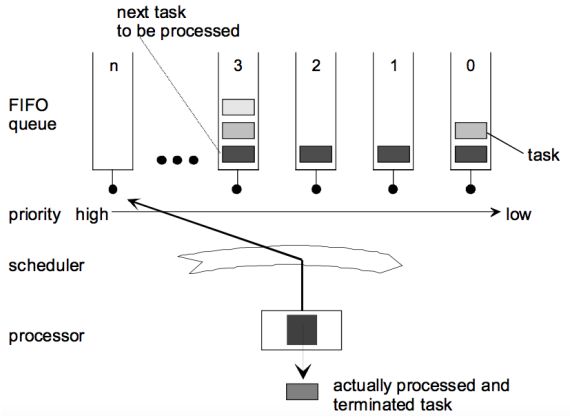

Figure 2.10: Scheduling and execution of tasks

The connection between the tasks and runnables happens during the configuration of the ECU, as previously mentioned when the runnables are mapped to one or more tasks. The mapping process can be made in several different ways, but a runnable that is not mapped to a task will not be activated and therefore will not run. The reason for this mapping is that the applications can represent an extremely high number of runnables but the OS has a limited number of tasks, mainly due to performance issues. Therefore it is impossible to map a single runnable to one task. Possible ways of mapping the runnables are approached by Khenfri *et. al* [24]. So once the runnables are mapped to the tasks, and the OS is running and has scheduled all the tasks, the ECU is ready to start executing the functions, represented by the runnables, and perform the actions requested by the applications.

Another basic object present in the OS is the Interrupt Service Routine also known as ISR. An ISR is a function with a higher priority than a task and with the ability to interrupt the running state of tasks. Due to this ability to interrupt tasks, the ISRs are subject to rules and have a limited number of OS Services that they can use. Another difference between ISRs and tasks is that ISRs are scheduled by the OS but are triggered by external events. An ISR might also activate a task. When that occurs, this specific task will have the highest priority of all tasks and start running once all ISRs have ended.

It was also previously introduced the concept of a Schedule Table. A Schedule Table is a mechanism which allows for the static scheduling and definition of a series of alarms and sets of Expiry Points. These Expiry Points allow for the activation of tasks, events, alarms. A Schedule Table contains a tick counter. This counter is the timer of the Schedule Table. So each Expiry Point is set to a specific tick counter and once the counter has reached the defined value, the Expiry Point runs and all defined actions within it are executed [17] [18] [19] [20] [21] [22]. Figure 2.11 [19] shows an example of a Schedule Table.

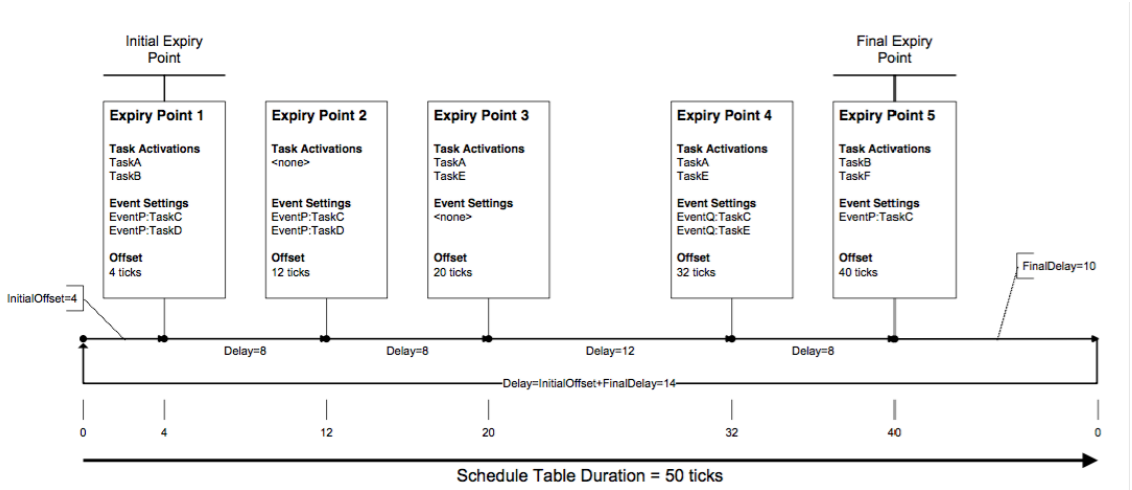

Figure 2.11: Anatomy of a Schedule Table

2.3 Machine Learning

2.3.1 History of the field Machine Learning

Machine Learning (ML) is the "field of study that gives computers the ability to learn without being explicitly programmed" phrased by Arthur Samuel [25].

The first *modern* computers appeared in the 1950's and 1960's, and with their appearance, the idea of making them intelligent also arose.

One of the most renowned and earliest works related to artificial intelligence and, consequently about Machine Learning, was the article written by Alan Turing. In this article, Turing approaches the learning problem by dividing it into two stages. The first is to simulate in a machine the brain of a child, while the second is to teach the *child-brain*, as one would teach a child, and make it evolve and thus resulting in an adult brain. Turing uses the theory of evolution as the underlying theory for creating the *child-brain* but he says that with a supervisor inducing mutations, the creation of a good *child-machine* might be faster. The author also introduces the concept of *punishment-reward learning* as the method for teaching the *child-machine* [26]. Thus, Turing introduces the concepts of *genetic algorithms* and *reinforced learning*.

In 1958, Frank Rosenblatt published an article, *The Perceptron: A Probabilistic Model for Information Storage and Organisation in the brain*, where he introduces the concept of a perceptron. The perceptron introduced by Rosenblatt is very similar to a NN and not to a single perceptron. Rosenblatt divides the perceptron in 3, or 4, areas, which correspond to the layers of a NN. The first area is the "retina" which is made of several S-points, or sensory units, that receive a Stimuli, the input and have a reaction to said Stimuli, for example, in an all or nothing manner. Then second area is the Projection Area, which corresponds to the hidden layer in a NN. This area is made of A-units, the association cells, which have several S-points connected to them. The A-unit fires, that is, produces an output, if the sum of the inputs, the signals from the S-points, is higher than a threshold. This area is not mandatory, and when it is non existent, the Retina connects directly to

the next area, the Association Area. The Association Area is very similar to the Projection Area. It is also made of A-units, that have as inputs signals from random A-units present in the Projection Area. The final area has the ”Responses”. These receive the outputs of the A-units from the Association Area, and, in a similar fashion of the A-units, will also fire if the sum of the inputs is higher than a threshold [27].

Figure 2.12 [27] shows the organisation of Rosenblatt's Perceptron.

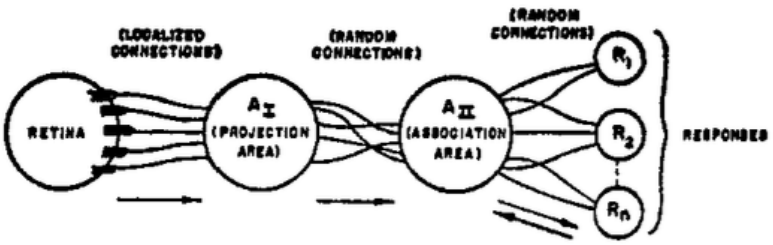

Figure 2.12: Organisation of a perceptron

One year later, 1959, a program capable of learning how to play the game of checkers was developed by Arthur Samuel. The author proposed two approaches for the problem. The first is the *Neural-Net Approach* which is described as being a randomly connected network, capable of learning through reward-and-punishment feedback. The second method, the one used to build the program, uses a well structured and specific network. This approach is what is now know as *Decision Tree Learning*. The program evaluates the current position of the pieces, "looks" a few moves ahead, assesses the new position and stores the positions [25].

In 1963, Vapnik and Lerner published an article introducing a new algorithm, the General Portrait Algorithm [28]. This algorithm would be the base for the Support Vector Machines (SVMs). The method proposed by the authors starts by placing the data in a Feature Space followed. Then it attempts to find a sphere, whose centre is called General Portrait, within which all points that belong to a certain subset, F_i of a pattern, Φ_i fall, and within no point of another subset F_j of another pattern, Φ_j fall.

In 1970 Seppo Linnainmaa wrote his master thesis where he introduced the concept of reverse mode of differentiation. This method is widely used today, and one of the applications is the back propagation error used in Neural Networks. This method consists in knowing the error between the estimated value and the real value and back propagating it along the network and thus changing and training the network [29].

In 1979 Hans Moravec successfully made a cart cross a room filled with obstacles without human intervention. The cart would move a meter, then it would stop, take some pictures about the surrounding environment and would analyse them so it could make a decision [30].

In 1981 Gerald Dejong introduced the concept of Explanation Based Learning, EBL. A program that learns using EBL is a program that learns with examples. For example, a program whose purpose is to play chess by learning with examples of chess games or specific important board positions [31].

In 1986 J. Ross Quinlan published a book called *Machine Learning* where he introduced one of the most used and common decision tree building algorithms the ID3 [32]. This algorithm will be explained further in this chapter, at Section 2.3.4.1.

In 1992 Boser *et al.* published a paper introducing the concepts of SVMs [33], which is introduced in Section 2.3.6.

In 2000 Schölkopf *et al.* published a paper where they adapted the algorithm introduced by Boser *et al.* [33], and modified it to be trained using only one class of data, thus creating a One Class Support Vector Machine (OCSVM) [34]. A brief explanation is given in Section 2.3.6.2.

In 2004 Tax and Duin also modified the algorithm proposed by Boser *et al.* [33], with the same intent as Schölkopf *et al.*, but the proposed algorithm is different than the one proposed by the Schölkopf *et al.*. A brief explanation is given in Section 2.3.6.2.

2.3.2 What is the field of Machine Learning

As previously stated, the field of ML is the "field of study that gives computers the ability to learn without being explicitly programmed" [25]. Ml attempts to create and develop algorithms that can learn from experience and make predictions based on the experience. This learning is usually regarding patterns that exist in the data used to teach the algorithm. After the pattern has been learnt, the algorithm can use the function that defines said pattern and use it to make predictions classify new data, among other possible uses.

Recently, there has been an increase in popularity of ML, and the reason for this increase can be attributed to mainly two factors.

The first is the exponential increase in available data. Today, the amount of new data being generated is overwhelming. This is due to the rising in popularity of the Internet of Things, social networks, among others. This extreme amount of data can be analysed to detect patterns and help make decisions. This is extremely useful in several areas such as marketing, finances and the stock market, politics, automation, among others. Machine Learning algorithms are ideal for the analysis of data and finding patterns in the data. However, they require a lot of computational power, which leads to the second factor.

Moore's law states that approximately every two years, the number of transistors on an integrated circuit doubles [35]. This means that computational power is constantly increasing at a rapid pace. Combined with this increase, the technology is also getting cheaper. So the problem mentioned in the first factor is not as relevant as one might have thought.

Some of the most common uses for Machine Learning are:

- **Pattern recognition** for example in speech or writing. An example is the several search engines and the keyboards in the cell phones that suggest words based on what was written;

- **Autonomous Driving** where ML algorithms, more specifically Vision ML algorithms, are used to recognise objects such as other vehicles and traffic signs that will then be used to teach and train the vehicle on how and what decisions to take;

- **Gaming** Machine Learning algorithms have been widely used to develop programs that can play games like chess and checkers. For example, recently a program beat for the first time an expert in the Chinese game Go [36].

2.3.3 How does Machine Learning work

To use ML algorithms, there can be considered five steps. The first step is to collect the data intended to be analysed. The data can be saved in a variety of formats, pictures, videos, sound, files, for example, text files.

After the data has been collected, a model, or algorithm needs to be chosen. The choice of the model depends on the type of collected data and the intent of the user with said data.

Almost in every case, the collected data needs to be prepared before it is given to the algorithm. For example, an algorithm does not receive a picture and looks at the overall picture. In reality the algorithm will look at every individual pixel and their characteristics.

The fourth and fifth step are the training and testing of the algorithm, respectively. These two steps are further explored in Section 2.3.3.2.

2.3.3.1 Types of algorithms

In order to analyse the data, an algorithm must be chosen. There are several types of algorithms and two main ways of grouping them, either by the type of training or similarity between algorithms. Following the second type of grouping, some of the most popular groups of algorithms are Neural Networks, Bayesian, and Decision Trees. Two additional methods used by several algorithms are also worth mentioning, the algorithms that use clustering as a method to analyse data and those who use regression.

A NN tries to replicate a human neural network regarding the organisation. Regardless of the type of NN, the structure can always be described as a network of neurons, the basic unit of the NN, which are connected to each other with a defined flow. They receive data as input, and, depending on the connections, they produce an output. The neuron of the NN receives an input, depending on the input, it alters its internal state and produces an output. The neuron also possesses an activation function that dictates the produced output. Figure 2.13 [37] represents a simple neuron.

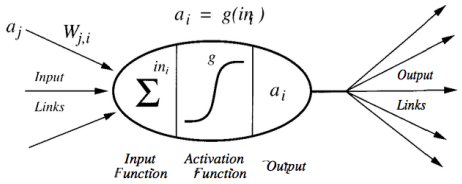

Figure 2.13: Structure of a simple Neuron

The NN works by connecting several neurons, attributing weights to the connections. During the training phase, both the weights of the neural networks and the activation functions of the

neurons can be modified, thus, representing the learning of the algorithm regarding the data being used as inputs. Depending on the type of NN being implemented, the structure of the neuron can change, as well as the number of middle layers, the layers between the input layer and the output layer.

The algorithms that belong to the clustering group, group the data in clusters, as the name of the group indicates. An example of a problem that can be tackled with a clustering algorithm is the recognition of objects. The objects are characterised by several points of data, that can be grouped into a cluster that identifies the object. This can be useful in autonomous driving. One of the most common clustering algorithms is the K-means algorithm.

The k-means algorithm takes the collected data as input and groups them into clusters. The grouping is done by first selecting a k number of data points. Then the algorithm chooses new data points and places the new point into the cluster whose centre point, one from the original k. This placement is based on the distance from the new point to each centre of the clusters. The objective is to minimise the means of the distances of the points to the centre of the clusters, where a point is assigned to one cluster, the one whose centre is the closest.

In the regression group, the objective is to produce a continuous output instead of discrete. Thus, instead of the data having a label, for example, a name, the produced output is a model where future data points can be used as input and the model will give a predicted output value. This can be useful to make predictions based on the behaviour and evolution of the collected data. An example of a possible area where it can be applied is in finances. The behaviour of consumers regarding a specific product might be able to be characterised by a function that helps predict the periods of the year where that product is most wanted, and therefore, help the company have the right amount of stock during the correct time of the year.

One of the most common methods of regression is the Linear Regression. In Linear Regression, the algorithm tries to find a line that establishes a relation between the input data points and the outputs. In its most basic form, it can have inputs x, outputs y, and the model would be of the form given by equation 2.1, where A is the inputs coefficient and B is a second coefficient that gives and addition degree of freedom to the model. In higher dimension, the model would be a plane or a hyperplane.

$$y = Ax + B \tag{2.1}$$

The Bayesian algorithms, is a group of probabilistic algorithms that are based in Bayes' theorem, represented in equation 2.2, where $P(A|B)$ is the probability of A occurring knowing that B happened, $P(B|A)$ is the opposite, B occurring knowing that A occurred, $P(A)$ is the probability of observing A and $P(B)$ is the probability of observing B. All the algorithms in this group are also based on the assumption that all attributes are independent of each other given a specific class, that is, that the occurrence of one input has no influence in future inputs or was influenced by previous

inputs.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.2}$$

Bayesian Networks are one of the most common algorithms in this group. This type of algorithms tries to establish relations between the variables and their probability of occurrence. For example, the objective is to know if the conditions are good to play tennis, and the variables are outlook, which can be sunny, overcast or rain, the humidity, which can be high or normal, wind, either strong or weak and the wetness of the ground. Then the algorithm establishes the probabilities of the outlook being sunny, overcast or rain, where the probability of rain also depends on the wetness of the ground, using Bayes' theorem. Then it also calculates the probability of humidity being high or normal, and the same for wind, either strong or weak. The network then outputs the probability of the conditions being good depending on certain specific states of some variables [38]. The Decision Trees algorithms are explored in detail in Section 2.3.4.

2.3.3.2 Training and testing the algorithms

After an algorithm has been chosen, the next step is to train the algorithm using the collected data. Regarding the training of the algorithms, Russel and Norvig [37], refer three distinct types of learning based on the feedback that the learning agent receives. These are:

There are three main types of training, Supervised Learning, Reinforced Learning and Unsupervised Learning.

Supervised Learning is a specific type of learning that uses labelled data, that is, the data has already been classified, and therefore, the algorithm knows beforehand what the desired output should be. This could lead to faster training times, but it is also easily to occur overfitting. Overfitting occurs when the algorithm was overly trained with the training data to an extent that the algorithm will give false results if the real inputs deviate to some extent from the training data. Therefore, it might classify the entire training set correctly, but it might also not be able to deal with new data.

Unsupervised Learning is the opposite of Supervised Learning. In this case, the algorithm has access to data, but it has no information regarding this data. This type of learning is suitable to find relations in the training data, and from the previous algorithms, the clustering group is a good option.

Reinforced Learning is more similar to the way human beings learn. The algorithm *interacts with the environment*, it uses trial and error, to learn what to maximise a reward, which could be an objective function. Two big areas that use Reinforced Learning is Game Theory and Control Theory. For example, a Neural Network could learn to play backgammon by trial and errors, the plays, and the reward would be winning the game. So if it won the game, the decisions made were good, if it lost, the decisions were bad.

Recently, a new type of learning was proposed by Hadfield-Menell *et al.* called Cooperative Inverse Reinforced Learning. This type of learning is more specific to Artificial Intelligence,

but it could still be used in other fields. The way this type of training works is by showing an algorithm experts performing some action with an *environment*, it still is reinforced learning, and the objective of the algorithm is to find what is the objective of the expert, the reward, and what actions maximise the reward [39].

After the algorithm has been trained, it is ready to be tested. If the test results are satisfactory, the algorithm is ready to be used. Otherwise, the algorithm can be further trained.

The following algorithms are the ones that are relevant for this dissertation, thus, they will have a more detailed explanation.

2.3.4 Decision Trees

In the Machine Learning field, there are several different algorithms that tackle the problem of giving a computer the ability to learn without being explicitly programmed. One of those algorithms is the Decision Tree algorithm.

A Decision Tree is an algorithm normally used to solve classification problems such as labelling data based on features of the data, for example, the famous tennis example [40], where the objective is to know if the user should play tennis based on the outlook, temperature, humidity, and wind.

The generated tree can be divided into two main sub-types, classification trees and regression trees. The first term is used when the output of the tree is a label that classifies the input data, and the latter is when the output of the tree can be a real number. Regarding the previous tennis example, the Decision Tree used is a classification tree. Regardless of this division, both sub-types use similar algorithms and have the same working principles.

A Decision Tree is characterised by having a root node (the beginning of the tree) internal nodes. Leafs nodes (which give the output of the tree) and branches which connect the nodes and correspond to a set of if-then rules with the objective of classifying data step by step based on the data's features and improving human readability.

The root node, or simply root, of a Decision Tree, is the beginning of the tree and divides into the first layer of internal nodes. This division is made by choosing one of the existing features of the data, using the tennis example, it could be the outlook, and the root divides into n internal nodes, where n is the number of possible and different values that the feature outlook can have.

The root node gives origin to the first internal layer and its nodes. The nodes in the first inner layer correspond to the next chosen features, in the tennis example, one of the selected features for a particular branch could be, for instance, the temperature. The internal nodes are connected to the root through branches where each branch has one of the n possible values for the feature that was chosen as root.

The first layer of internal nodes can give origin to the second layer of internal nodes or, if the branches originated from the internal node give the output of the tree, the output nodes are called the leaf nodes, thus being the end of that part of the tree. The resulting tree for the tennis example can be seen in Figure 2.14 [40]

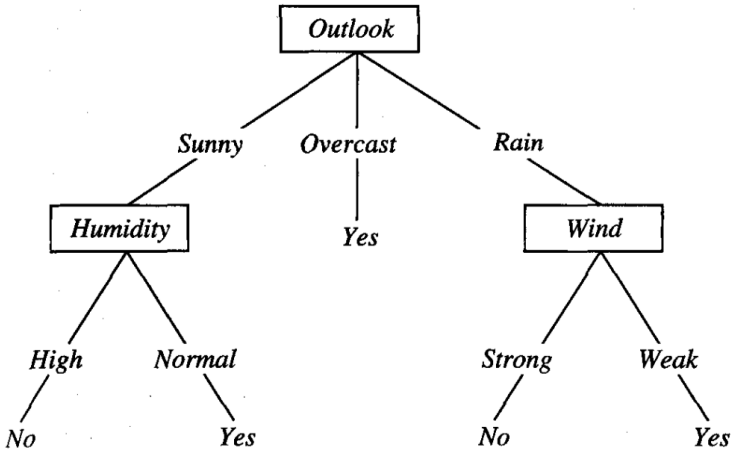

Figure 2.14: Decision Tree for the Tennis Example

To help with the construction of a decision tree, several algorithms have been developed for that purpose. The most common ones are ID3, C4.5 and C5, CART, and MARS [41].

2.3.4.1 ID3 Algorithm

The ID3 algorithm is an iterative algorithm whose working principle will be explained since it is one of the most used and both C4.5 and C5 are improvements of the ID3 algorithm. C5 is an improvement of the C4.5 algorithm.

First and foremost, before building a decision tree or training any Machine Learning algorithm, it is of the utmost importance to have training data.

After the acquisition of the training data, the ID3 algorithm will select a subset of the data and use it as the foundation of the tree. The elements that constitute the subset are chosen at random. The selected subset is called the training window.

Definition 2.6: *A training window is a consecutive subset of the total acquired data that was chosen and will be used to form a decision tree.*

After obtaining the initial training window, the ID3 algorithm will start the construction of the tree.

Firstly it will analyse the different available features and will evaluate the gain in information of the tree for each feature. This calculation has the objective of choosing as root the feature that can provide the tree with the biggest amount of information for the final tree to be as small and generic as possible.

After choosing the first feature, the root is divided into the first internal layer where each branch has one of the possible values for the chosen feature without any repetition.

Then, ID3 repeats the calculation of the information gain associated with each of the remaining features to choose the next feature that will be added to the tree. After the decision, the new feature is added and the process repeats until all the features have been chosen.

After the Decision Tree has been built, it will be tested using the remaining training data that is not part of the training window.

If all the produced outputs are correct, then the algorithm terminates, and the built tree is the final one.

Else, a new and smaller subset of the remaining training data is chosen and added to the initial training window, and the building process is repeated.

This iterative process is repeated until all the objects are classified correctly. In an extreme case, the training window will contain all the initial training data [40].

As previously stated, both C4.5 and C5 algorithms are improvements of ID3. The main differences being the new capabilities that these algorithms provide. The new algorithms have the ability to deal with features that assume continues values and missing values. They have the capacity of pruning the tree, which occurs after the tree has been built. For the pruning, the algorithm goes through the tree and tries to find branches that do not provide useful information and replacing them with leaf nodes. The C5 algorithm is also capable of building and reaching the final Decision Tree significantly faster than the previous versions, C4.5 and ID3 [42] [43].

2.3.5 Graphs

A graph is a structure containing two or more objects and their relations, represented by the connections between said objects. These objects are represented by what are called vertices of the graph and the connection between a pair of objects, or vertices, is known as an edge. The vertices connected to the vertex being focused are called adjacent vertices.

Graphs are not only used in graph theory but also in several fields of engineering. A common diagram used both in electrical engineering and computer science is a state machine. This is very similar to a directed graph with some additions such as inputs, outputs and the edges are associated with a condition, that means, the state machine goes from one vertex to another if the condition associated with an edge occurs.

There are several types of graphs, and three types will be briefly introduced. The undirected graph, the directed graph and a weighted graph.

The undirected graph is characterised by having edges that do not have a direction. This means that there is no sense of direction of where does the edge starts and ends. So if there are two vertices, vertex 1 and vertex 2, and they are connected by an edge, both edges (1,2) and (2,1) are the same.

Figure 2.15 shows a small undirected graph.

In a directed graph there is an orientation, thus, using the example from the undirected graph, the vertices (1,2) and (2,1) will be different.

In a directed graph an edge can be of two types, an out-edge or an in-edge. An out-edge is an edge that "leaves" a vertex and an in-edge is any edge that "enters" the vertex.

This concept of leaving and entering can be better understood by looking at Figure 2.16. Here it is possible to see that vertex one only possesses out-edges, these are represented by a red colour.

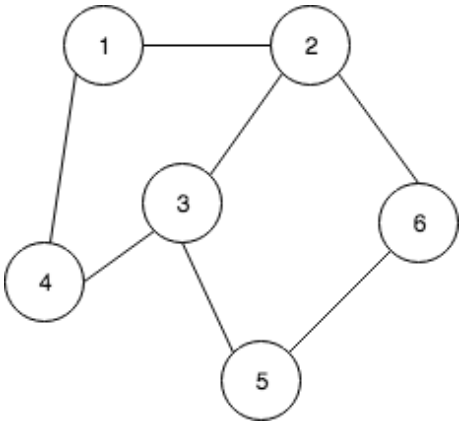

Figure 2.15: Example of an undirected graph

Vertex five possesses only in-edges and these are represented by a blue colour, and one is blue, the common arrow to vertex one. All the other vertices possess both in and out-edges.

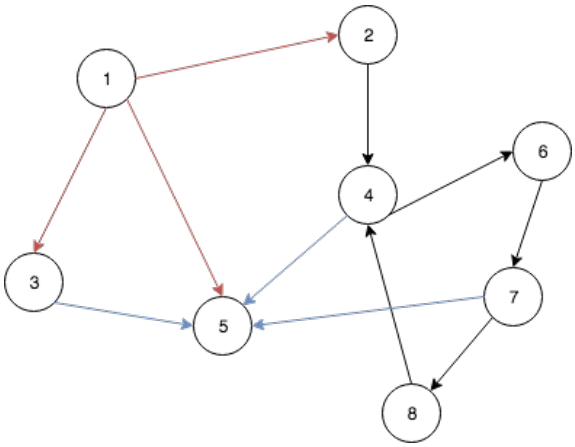

Figure 2.16: Example of a directed graph

The third type explored in this chapter is the weighted graph. This graph can be either directed or undirected, but the novelty is that each edge has a weight. This weight can represent a variety of variables such as a cost, distance, capacity, time, a probability. These graphs are very useful to help understand and solve optimisation problems such as the famous travelling salesman problem. This problem approaches the difficulty of finding the shortest route that visits every city in a group of cities. This problem can be escalated and used in many different contexts, for example, the distribution of packages by a mailman.

These graphs can be used together with other algorithms to find the optimal path where one of the main goals is to find such path as quickly as possible. Figure 2.17 represents a weighted graph.

2.3.6 Support Vector Machines

A Support Vector Machine is a type of machine learning algorithm whose main purpose is to separate both the input data into different planes by finding an hyperplane that separates the data.

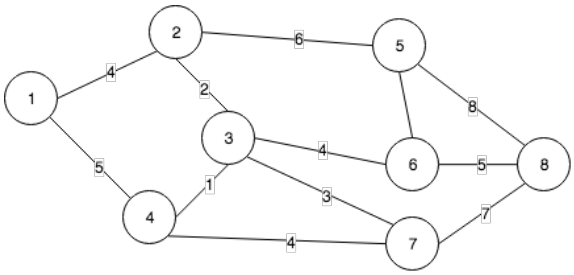

Figure 2.17: Example of an undirected weighted graph

The data is separated into what is commonly called classes or categories. The number of different classes obtained can vary and depends on the type of algorithm that is being implemented. Two different cases will be introduced further in this subsection [44].

2.3.6.1 Working Principle of a SVM

As stated in the introduction of SVM, the purpose of this algorithm is to separate the input data into different classes. During the training phase, the SVM receives the training data as inputs, and the data is distributed in a feature space.

Definition 2.7: *The Feature Space is the mathematical space where all the problem's variables exist and has a dimensionality of* $\mathbb{R}^n$.

The number of dimensions, n, of the feature space corresponds to the number of features that are being considered for the data. For example, if we take the tennis example used in the subsection 2.3.4, where the objective is to separate the days where the output is to play tennis from the days where the output is to not play tennis, the plane has four dimensions, outlook, temperature, humidity and wind.

After the input data has been placed in the plane, the algorithm starts. The objective is, as previously mentioned, to find a function defining an hyperplane that separates the data into the desired number of classes, in the tennis example two, to play tennis or to not play tennis.

Figure 2.18 shows the separation between two classes of data.

Using the figure 2.18, it is possible to define the dots, both red and blue, that are crossed by the dashed line as the support vectors. They are called vectors due to the vectors associated with these points, from the origin of the feature space to the point itself. These are the most difficult points to classify because they are the data points closest to the function separating the two classes. The support vectors are the points that will influence and help the SVM find the hyperplane, in the picture example the line, that best separates the data.

The best hyperplane is defined as the one that possesses the biggest margins between the different classes. Using figure 2.18 as an example, the margins are the distances between the line and the dashed lines, and the objective is to find the hyperplane that separates the data and has the biggest margins possible. The reason for wanting the biggest margins as possible is that it establishes a gap for the future data and thus, trying to minimise the misclassification of the data.

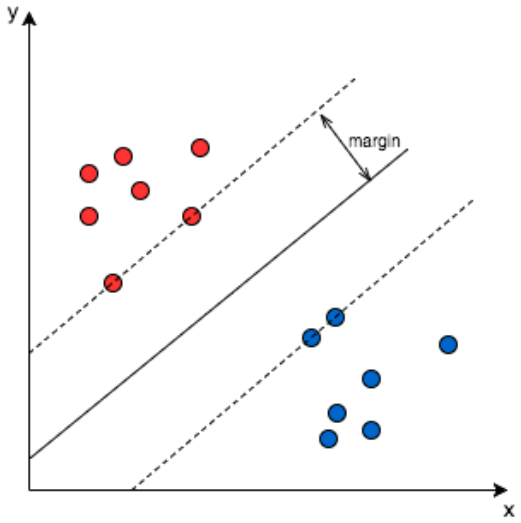

Figure 2.18: Support Vector Machine separating the red dots from the blue dots

Therefore, even though several hyperplanes separate the data, only one has the biggest margins. The hyperplane is defined by the equation:

$$w^T x = 0 \tag{2.3}$$

Where w is a vector that is normal to the hyperplane, thus being a vector that helps to define the hyperplane.

Finding the hyperplane is mathematically the same as finding two hyperplanes that separate the data and each one crosses the support vectors, where the distance between them is the biggest and then finding the hyperplane that is situated in the middle of the two previously discovered.

The equation 2.4 show the maximisation problem.

$$m = \frac{1}{||w||} \tag{2.4}$$

Where m is the margin, w is the vector that is normal to the hyperplane. So if w's value diminishes, the margin increases and vice versa. So by minimising the norm of vector w, the margin is maximised, which is the objective, to find the hyperplane that has the largest margin associated.

However, it is not always possible to find two hyperplanes that can separate the data.

Figure 2.19 shows an example of that case.

When that happens, the SVM "transforms" the feature space into a new feature space. The difference is that the new feature space has a higher dimension than the original one. This is useful because by giving the feature space a new dimension, the data might be linearly separable. It is possible to do this transformation an unlimited amount of times when at the limit, the feature space would have an infinite amount of dimensions. Nonetheless with every new dimension, the computational power required for the algorithms increases as well.

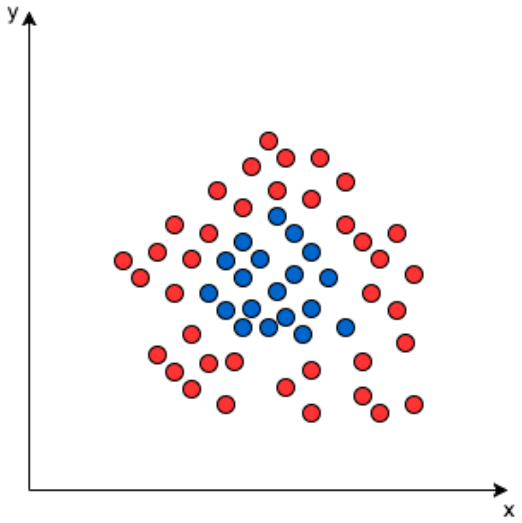

Figure 2.19: Support Vector Machine separating the red dots from the blue dots

Giving a new dimension to the Figure 2.19 could result in Figure 2.20 and thus, it is possible to find a hyperplane that linearly separates the data.

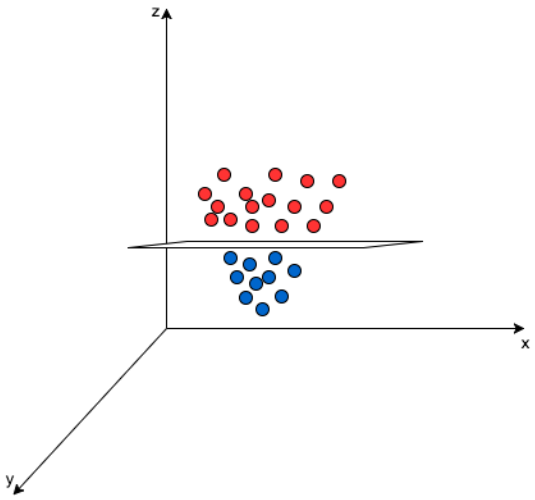

Figure 2.20: Support Vector Machine separating the red dots from the blue dots in the new feature space

In a problem where the data has several features, the dimensionality could quickly become a problem, but it turns out that the SVM does not require to work in the new feature space explicitly. It only needs to compute pair wise dot products with the training data. To compute the dot products, the SVM firstly uses what are known as kernel functions, which transform the initial feature space into a new feature space with a higher dimensionality. Then it is possible to mathematically prove that the algorithm only needs to calculate the dot products of the vectors in this new feature space in order to obtain the hyperplane [45].

After the hyperplane has been found, considering all the training data, the training phase is

complete, and the SVM is ready to be used. As the SVM receives new input data, it produces as an output a class to which the received input data belongs. This class is based on the position of the new input data in the feature space, and it is a relative position to the hyperplane that was defined during the training phase.

To summarise, the SVM receives the input data existent in a feature space of dimension n, where n is the number of features of the data and tries to separate such data into different classes. The separation is done by an optimal hyperplane. This hyperplane is defined by being equidistant to the support vectors while maintaining the biggest margin possible between them. The support vectors are the data points of each data class that are the closest to the other class.

If a linear hyperplane can not be found in the current feature space, the "kernel trick" is used where the dot product of the input data is calculated in the new feature space through the use of a kernel function and thus finding the hyperplane in the new dimension. After the training phase, the new input data is mapped into the feature space, and the SVM produces as an output the class to which the input data belongs, based on its position relative to the hyperplane [44].

2.3.6.2 One Class SVM

In the family of algorithms of Support Vector Machines, there are several types which tackle different types of problems. There are multi class SVM, one class SVM, binary SVM, and they can be associated with different algorithms, depending on the problem.

The One Class Support Vector Machine is interesting regarding this dissertation since it is trained using only one type of data. The purpose is, during running time, to find anomalies that do not fit the profile of what was learned to be the normal data.

There are two main types of OCSVM, one proposed by Schölkopf *et al.* [34] and the other proposed by Tax and Duin [46].

The first type is very similar to the concept previously introduced. The difference is that the hyperplanes try to create the biggest margin between the data points and the origin of the feature space. After the training period, a new data point will be labelled as an anomaly if it is located on the side of the origin, regarding the hyperplane supported by the support vectors from the training data. An example of this type of SVM can be seen in Figure 2.21.

The second type, is called a Support Vector Data Description (SVDD) by the authors, and it creates a sphere that encompasses all the data points and then tries to minimise the volume of the sphere. After the training is done, when it receives a new data point, this point will be classified as an anomaly if it falls on the outside of the sphere and it will be labelled as a normal data point if it is located inside the sphere. An example of this type of SVM can be seen in Figure 2.22.

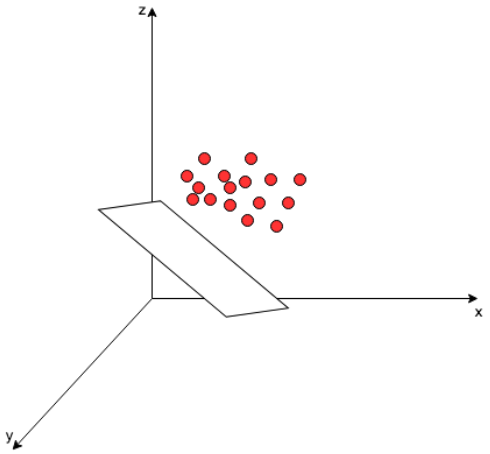

Figure 2.21: The SVM proposed by Schölkopf *et al.*

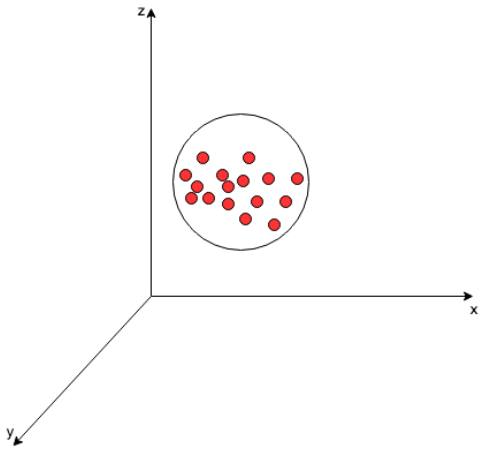

Figure 2.22: The SVDD proposed by Tax and Duin

Chapter 3

State-of-the-Art

3.1 Introduction

This chapter will present the State-of-the-Art, that is, what is currently being done that relates to the work presented in this dissertation.

In a first approach, methods for monitoring systems will be presented at section 3.2, thus starting with a broader perspective.

Followed by a section introducing the available tools within the AUTOSAR software, that help with the task of monitoring the behaviour of the ECU, section 3.3.

The final section 3.4, introduces work whose purpose is to monitor ECUs, and therefore, monitoring the software of the ECUs, AUTOSAR, using the tools introduced in the previous section.

3.2 System Monitoring

A System monitoring system is a system designed to monitor other systems. In this section different types of systems whose goal is to monitor systems with the purpose of finding errors or misbehaviours will be presented.

Murphey *et al.* [47] use a combination of a model-based approach and machine learning methods to monitor and diagnose faults in electric drives.

A model-based approach consists in creating a model of the system that is being studied. This model represents the ideal and correct behaviour and will be the base for the comparison of the actual system and its behaviour.

Murpheys' *et al.* [47] model has a set of inputs, such as a DC voltage, a reference torque, followed by the components, represented by a series of mathematical equations whose purpose is to simulate the behaviour of the components. The system has an output which is the torque delivered by the induction motor.

The model was then tested for normal and faulty operations resulting in two groups of data. The reason for testing the model with both situations is that it is easier to compare a faulty situation with the normal one instead of comparing two faulty situations since the differences in the latter

case can be quite subtle. The data generated from this model was then sent to the diagnosis system. Here the signals had to be treated before they could be used to train the Neural Network (NN).

There were nine possible fault scenarios, six for single-switch faults and three for post-short-circuit faults, so two different NN were built.

The first was a NN made of two independent NNs with one output based on a *winners takes all* approach to obtain the output. This NN consists on several parallel NNs that are trained independently from one another. Each one produces an output that serves as an input in a final function [48]. The first NN was responsible for the six single-switch faults while the second for the post-short-circuit faults. To distinguish all the outputs, the output layer produced a k bit value. So in the first NN, k was seven, one normal operation and six faulty, and so the output would be something like 0010000, where 1 was the operation that happened.

The second NN had multiple neurons in the output layer. In this case, there were ten neurons in the output layer, one for the normal operation and the other nine for all the different faulty operations. If a fault situation has occurred, the respective output neuron will activate. The output layer produced a similar result to the k bit output produced by the other NN.

The first NN achieved an accuracy of 100% in just one iteration of the previously mentioned algorithm. The second NN detected with 100% accuracy all the single-switch faults but had only a 90% and 92% accuracy when detecting post-short-circuit faulty class 1 and class 2.

Finally, the diagnostic system was tested using bench generated data, much closer to real data. The lowest detection accuracy was 96% which is a great result and thus proving the method and algorithm to be a good choice for future work [47].

Feng *et al.* [49] also used a behaviour model approach to detect cyber-attacks. The specific attack that was of interest was the distributed denial-of-service (DDoS) where a system or a group of systems flood a target system traffic, impeding new connections to be made and therefore stalling the attacked system.

The studied system was the network of their campus, and they monitored the TCP packets sent to the /24 darknet to test the tool. Even though the tool was tested using this specific data, its implementation allows it to work in any network system.

It begins by collecting data over a long period, from said system. Then it builds a frequency distribution, where it relates the number of unique access to each port and how long it took for that number of unique accesses to that port to happen. It then uses this distribution to establish and extract what the normal behaviour is. The most common occurrences are denoted as normal, and the anomalies are the least probable ones, the most unlikely to happen, and are therefore disregarded. Finally, it is ready to detect possible attacks. It starts by counting the number of unique accesses to a port, similar to the training. After the counting is done, it compares the obtained count with the count that was established as normal during the training for the same port. If the obtained count is substantially greater than the count obtained during the training, then an alert is given indicating that an attack has possibly occurred in this port.

3.3 Monitoring Mechanisms in AUTOSAR

3.3.1 Introduction to AUTOSAR

The AUTomotive Open System ARchitecture, or AUTOSAR, is a partnership between manufacturers and suppliers from the automobile industry that joined to create a set of standards and software to answer and deal with the increase in complexity in the modern vehicle.

The software, with the same name, AUTOSAR, is a software architecture present in every ECU of the vehicle. Its working principle consists in having applications that constitute a set of instructions. These instructions correspond to actions that will be carried out by actuators present in the vehicle, that receive these instructions from the ECUs. Therefore the applications are linked to the ECUs and make use of the services provided by the ECU. The software can be divided into three layers, the Application Software Layer, the Run-Time Environment and the Basic Software Layer. The applications are situated in the Application Software Layer, the Run-Time Environment establishes the connection between the other two layers and is responsible for the communication between applications. Finally, the Basic Software Layer offers the services that allow for the execution of the instructions given by the applications and communicates directly with the vehicle components. A more detailed explanation is given in Chapter 2.

3.3.2 Introduction to the mechanisms

With the objective of monitoring the ECU's behaviour, there are a few tools in the AUTOSAR software that facilitates the monitoring. The main tools are the use of System Hook functions, a tool called VFB Tracing, where VFB stands for Virtual Function Bus, and the Watchdog Manager.

These tools offer the possibility of monitoring different parts of the AUTOSAR system, each one with their method, and thus allowing for the monitoring of the behaviour of an ECU by using one or a combination of these tools. The first two methods have a similar working principal, and although the third mechanism differs from the first two, all three are capable of monitoring smaller components within the software, and by combining the behaviour of those single components, it is possible to obtain the bigger picture. It is possible to detect when one of those components starts its execution when it ends, how long it took, which component came before and which will come after. By having this information, it is possible to assemble a sequence of actions and their timing and later use this observed behaviour to compare with the current behaviour and detect any deviation from what was considered normal under careful and controlled observation.

3.3.3 System Hooks

System Hooks are what are called hook functions and, within the AUTOSAR Software, they are implemented by the user and called are later invoked by the Operative System.

A hook function is a reactive function, which means it reacts to a certain action or event and then runs the code that is inside the hook function. This means that if we have a defined hook function to be activated when an event A occurs, then when the event A happens, the hook function

will be called and execute its code. This allows the user to modify the behaviour of an operative system, an application or even a specific software component. In the AUTOSAR software, the same concept applies. There are a set of already predefined hook functions that can be used to execute new user defined code into the Operative System. Since the AUTOSAR software, and to be more precise, its Operative System is based on the OSEK Operative System, as it is explained in Chapter 2, its hook functions are very similar.

The AUTOSAR Operative System provides a set of hook functions that can be declared and modified by the user, and these are also activated by specific events. These hook functions have a higher priority than the normal tasks, which means that regardless of the priority of the task that is running, if the event that activates a declared and defined hook function occurs, the hook will be called before the next task that was supposed to run.

The available hook functions are the *ErrorHook*, *StartupHook*, *ShutdownHook*, *PreTaskHook*, *PostTaskHook* and *ProtectionHook*. All these functions exist but in order to be used they need to be declared and their behaviour defined by the user, prior to the system starts to run [12] [19] [21].

3.3.3.1 *ErrorHook*

The *ErrorHook* function is used to deal with errors that occur in the system. The literature defines two types of errors, Application Errors and Fatal Errors. The first is when the operating system was unable to execute the requested service, but there are no signs of the operating system internal data corruption. The latter is when the operative system cannot assure that its internal data is not corrupted. Depending on the type of Error that occurred and where it occurred, the function will be called with a different type of parameter.

AUTOSAR allows the user to define several *ErrorHook* functions, for example, one for each application. What happens when that application sets off its *ErrorHook* function, is that it will call the main *ErrorHook* function and adapt the general code to the specific application.

Although has it was previously mentioned, the behaviour of the function is defined by the user, there are some guide standards to help define the function's behaviour. First, a switch case, where depending on the parameter that the function received, it will execute a different action. After the case has been found, a second routine can occur to use a service call to obtain more information about the error. The Figure 3.1 [12] exemplifies this behaviour [12] [19] [21].

3.3.3.2 *StartupHook* and *ShutdownHook*

The *StartupHook* and *ShutdownHook* functions have similar purposes. They both allow for the user to define specific behaviour for the operative system to execute during the system start-up and shutdown.

Figure 3.2 [12] shows how the *StartupHook* function works. The system (re-)starts, and before the operative system kernel starts, which is the core of the operative system, the hook function is called and may execute specific user defined initialization procedures. The *ShutdownHook* works similarly but during the shutdown of the system. So before the system shuts down, the hook

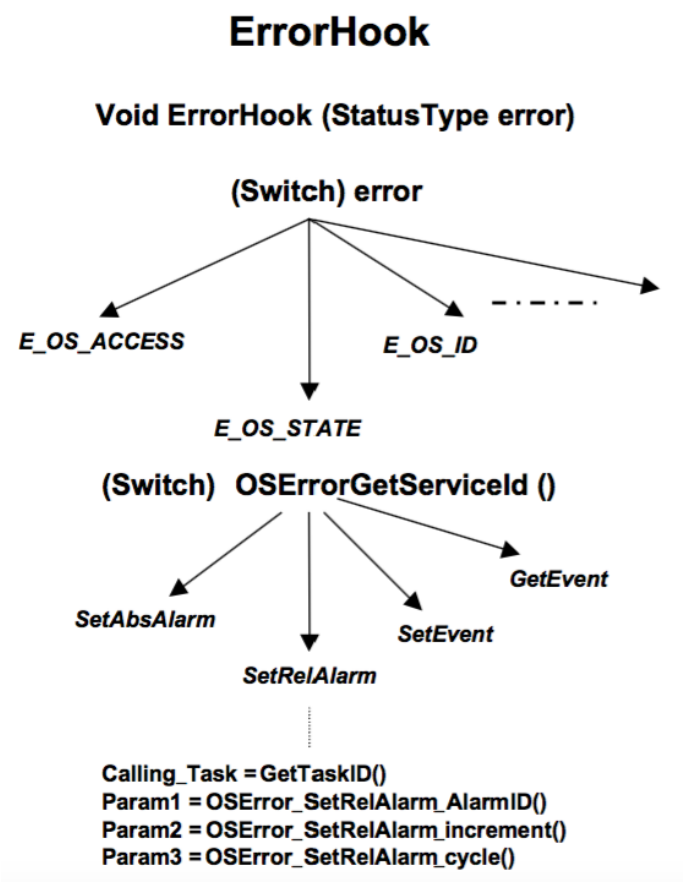

Figure 3.1: *ErrorHook* routine

function is called and can execute user defined termination procedures. This hook function can also be activated when a Fatal Error occurs.

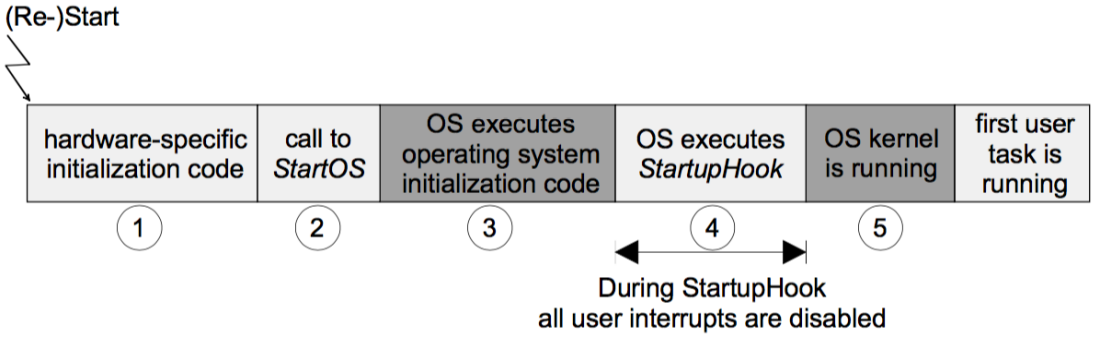

Figure 3.2: *StartupHook* during system startup routine

3.3.3.3 *PreTaskHook* and *PostTaskHook*

The *PreTaskHook* and *PostTaskHook* functions are very similar, the main difference being when they are activated. Both are activated when there is a switch from one task to another, the *Pre-*

TaskHook is activated when a new task enters its running state and the *PostTaskHook* is called before a task leaves its running state and the new task is called. These functions allow the user to obtain information regarding the running tasks. One example is to measure the task runtime. A timer can be activated in the *PreTaskHook* and measured again when the *PostTaskHook* is called, thus obtaining the elapsed time. Service Calls might also be used in these functions to obtain more specific information such as the task id. From all the hook functions provided by the AUTOSAR software, these two are the most useful for the purpose of this work [12] [19] [21].

Figure 3.3 [12] shows how the *PreTaskHook* and *PostTaskHook* can be used. Task 1 will terminate and thus entering the suspended state. As previously explained, when that occurs, the *PostTaskHook* is called which will execute its user defined behaviour. Then, just before task 2 enters its running state, the *PreTaskHook* function is called executing its user defined code.

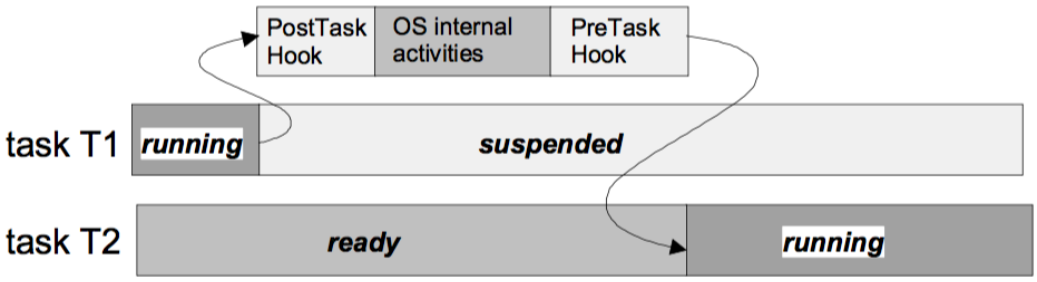

Figure 3.3: Usage of *PreTaskHook* and *PostTaskHook*

3.3.3.4 *ProtectionHook*

The *ProtectionHook* function is the only hook function that does not exist in OSEK and is therefore new in the AUTOSAR system. This function is called every time a major error occurs, and its purpose is to protect the system against the consequences of such error. These errors can be, for example, when a task runs for a much longer period than what was expected, the resources used by a task surpassed what was allocated, as for example, it exceeded the allocated memory.

This function is much more restricted regarding what the user can define in its behaviour. It has four possible actions, that can be activated depending on the parameter that the function received upon its activation.

- It can forcibly terminate a Task or a category 2 Interrupt Service Routine which is causing the problem;
- It can forcibly terminate the OS-application the Task or the Interrupt Service Routine belongs to;
- It can Shutdown the system;
- It can do nothing.

This hook function can also be useful to detect the extreme cases and help control the tasks memory usage, which is an advantage regarding the other two tools that will be approached in this chapter since none of them can directly monitor the memory [12] [19] [21].

3.3.4 VFB Tracing

The VFB Tracing is another available mechanism in AUTOSAR that can be used to achieve the purpose of this work, to monitor the behaviour of an ECU.

VFB is, as the name states, a virtual bus which allows for Software Components, the building blocks of the application in AUTOSAR, to communicate with one another and with the middle layer, the RTE, as it was previously mentioned in Chapter 2. It is virtual because, even though it does not exist physically as a bus, it is where the mappings and configurations are set and where the implementation of the RTE and other Basic Software modules are based on, which are the physical representation of the VFB.

So the VFB Tracing presents the user with a set of functions that can be used to trace events that happen in the VFB, which is the same to say that happen in the RTE and some other Basic Software Modules.

These functions are very similar to the mechanism previously introduced, in which they are also hook functions, but in this case, they are specific for the VFB. This means that they are a useful mechanism to monitor the SW-Cs, or in other words, the behaviour of the applications and its interactions with the ECU [20].

The available hook functions can be divided into seven groups:

- RTE API Trace Events
- Basic Software Scheduler API Trace Events
- COM Trace Events
- OS Trace Events
- Runnable Entity Trace Events
- BSW Schedulable Entities Trace Events
- RPT Trace Events

3.3.4.1 RTE API Trace Events

The RTE API Trace Events are events related to the Application Program Interface of the RTE. The available functions are *RTE API Start* and *RTE API Return*. The first function is activated when a SW-C interacts with the RTE API and makes a call. The second function is activated just before the RTE API returns control to the SW-C that invoked the call. With this two functions, it is possible to know when a SW-C made a call and when that call ended [20].

3.3.4.2 Basic Software Scheduler API Trace Events

The Basic Software Scheduler API Trace Events are related to the API of the Basic Software Scheduler, which is also part of the RTE, as it is explained in Chapter 2. The Basic Software Scheduler schedules the schedulable entities of the Basic Software Module. The schedulable entities can be though of as the SW-Cs of the modules, and the modules as the available applications in the Basic Software Layer. The available hook functions of this type in VFB Tracing are similar to the previous type, *BSW Scheduler API Start* and *BSW Scheduler API Return*.

The first is activated when a Basic Software Module makes a call to the BSW Scheduler API. The second is when the BSW Scheduler returns the control to the module which made the call.

These functions are useful to know when a service, which is one or more entities present in the invoked module, existent in the ECU in the Basic Software Layer, is active and performing the desired initial action commanded by the application in the Application Software Layer, that in turn commanded the Basic Software Layer, where the call occurred [20].

3.3.4.3 COM Trace Events

COM Trace Events are events that occur when the RTE is interacting with the Communication Module to use communication services.

The available hook functions are the following, *Signal Transmission*, *Signal Reception*, *Signal Invalidation*, *Signal Group Invalidation*, *COM Callback*.

Signal Transmission is activated when the RTE intends to transmit a signal to another ECU, inter-ECU communication. So before the signal is transmitted, this function is called.

Signal Reception is activated when the RTE has successfully attempted to read an inter-ECU signal.

Signal Invalidation is activated when the RTE has detected an invalid request of an inter-ECU signal, either transmission or reception.

Signal Group Invalidation is activated by the RTE when a signal group, which is when the signal consists of one or more group of signals, has made an invalid request of an inter-ECU signal group, either transmission or reception.

COM Callback is activated when the RTE intends to start a COM call-back, which is a notification of a certain event of the completion of a process.

All these functions are useful to know when a SW-C needs to access services from another ECU and therefore needs to perform inter-ECU communication [20].

3.3.4.4 OS Trace Events

The OS Trace Events hook functions are one of the most important group of functions provided by the VFB Tracing, even though some are redundant when used together with the System Hooks. The available functions are *Task Activate*, *Task Dispatch*, *Task Termination*, *Set OS Event*, *Wait OS Event* and *Received OS Event*.

The *Task Activate* hook function is activated immediately before the activation of a task, so when a task enters the ready state. To further read about the different states of the task, go to Chapter 2.

The *Task Dispatch* hook function when a task has entered its running state, which is the same to say it has started to be executed, similar to the *PreTaskHook* introduced in section 3.3.3.3. *Task Termination* is activated just before a task leaves its running state, which is similar to the *PostTaskHook*. The usefulness of using both system hooks and the VFB Tracing is that it has an adjacent redundancy which can increase security and consequently the safety of the vehicle. But the System Hooks are preferable because they exist in the Operative System, so they are at the same level as the tasks, and they do not require the activation of the VFB Tracing.

Set OS Event is called when the RTE attempts to set an OS Event. An event is any action that occurs in the OS related to an OS object, a task, an Interrupt Service Routine, and is what makes the OS function since it is an event based OS. *Wait OS Event* is called when the RTE attempts to wait for an OS Event. *Received OS Event* is called when the RTE returns from the wait state after waiting for an OS Event [20].

3.3.4.5 Runnable Entity Trace Events

This group is also significant since the Runnable Entities are the building blocks of the SW-C, which in turn are the building blocks for the applications. For more information go to Chapter 2.

The two functions in this group are *Runnable Entity Invocation* and *Runnable Entity Termination*. The first is called just before a Runnable Entity starts to be executed and the second is activated immediately after the Runnable Entity was executed. These two functions allow the monitoring of Runnable Entities and therefore the actual execution of the SW-Cs and applications. This is more specific than monitoring tasks since a task has one or more Runnable Entities mapped to it [20].

3.3.4.6 BSW Schedulable Entities Trace Events

The BSW Schedulable Entities Trace Events is similar to the previous group, but this time related to the Basic Software Scheduler and the schedulable entities present in the Basic Software Modules.

The two available functions are *BSW Schedulable Entity Invocation* and *BSW Schedulable Entity Termination*, which, again are similar to the previous ones. So the *BSW Schedulable Entity Invocation* is activated just before the execution of a BSW Schedulable Entity and the second immediately after the execution of a BSW Schedulable Entity [20].

3.3.4.7 RPT Trace Events

This group of functions will not be explained because it does not serve any purpose regarding this work.

3.3.5 Watchdog Stack

The *Watchdog Stack* is a group of modules present in the AUTOSAR Basic Software Layer. Therefore, it is an available service in the ECU, and if the user intends to use it, it should configure the *Watchdog Stack* by properly declaring it and implement its use in the code. The *Watchdog Stack* is composed by three modules, the *The Watchdog Manager*, the *Watchdog Interface* and the *Watchdog Driver*. *The Watchdog Manager* is responsible for managing the two other modules and providing the means to monitor programs running in the same ECU as the *Watchdog Manager*.

The *Watchdog Interface* is an interface that allows the *Watchdog Manager* to select the correct *Watchdog Driver*, thus being an abstraction of the ECU hardware.

To monitor programs, the *Watchdog Manager* gives three different methods to monitor different aspects of a program. It allows the supervision of the program flow, which means it can verify if the sequence of actions during the execution of the program is the same as the expected sequence. It allows verifying if the program is fulfilling the timing requirements regarding deadlines for the program actions. The third and last mechanism allows the supervision of the aliveness of the program. This means that the *Watchdog Manager* can verify if a program is "alive" and by alive it is meant if the program is running or runs within a specific time interval.

Before choosing and implementing the monitoring method, the user needs to choose which program it wants to monitor. This program can be a complete application or just a part of it, and it will be called the *Supervised Entity* (SE). The *Supervised Entity* has no direct relation with any of the existing entities in AUTOSAR, Software Components, Runnables Entities, Basic Software Module Entities. It can be a single function, an entire algorithm, an entire Operative System task. And an algorithm or task can be made of several SEs. One restriction is that a SE cannot be divided into different tasks, hence it must exist in a single task.

A second important concept is the concept of a *check point*. A *check point* is an additional piece of code that is introduced into the program and is what will alert the *Watchdog Manager* of the current state of the SE. This check point acts as a flag or an alert, so when the check point is reached, the *Watchdog Manager* is alerted and can perform the necessary actions, if any at all. For an entity to be considered a SE, it needs to have at least one check point associated. A SE can have more than one check point, and the number of check points in a single SE is directly related to the accuracy of the monitoring, since the *Watchdog Manager,* can monitor the SE closely. However, the more check points there are in a SE, the lower the performance of the overall ECU since it will require more resources to be able to handle all the check points [50].

3.3.5.1 Monitoring of the program flow

As previously mentioned, the *Watchdog Manager* creates the possibility of monitoring the flow, or sequence of instructions, of a *Supervised Entity*. It does so by using the checkpoints and verifying if the order in which they are being executed is the same order in which they were placed in the code. So if a SE has three check points, A, B and C, and they were placed in the SE in that respective order, the *Watchdog Manager* will not detect an error if they are executed in that same

order. But, if the order in which they are seen by the *Watchdog Manager* is, for example, A, C and B, then an error occurred.

As it was also previously mentioned, the more check points a SE, the better the monitoring over it, the disadvantage being the increase in resource consumption. The user should also define at least two check points in a SE since only one is insufficient to verify any sequence of actions done by the SE.

The *Watchdog Manager* gives the user the possibility to define a tolerance of false program flows. This means that in a SE with, for example, ten check points, it might happen that two of them occur in a different order. If the user had previously defined a tolerance of two, then the *Watchdog Manager* will disregard those two mistakes.

The monitoring of the program flow can be useful to detect illegal program counters jumps within the whole system [50].

3.3.5.2 Timing Monitoring

The Timing Monitoring method observes if the checkpoints placed in the SE are reached before a deadline. This type of monitoring can be achieved by assigning the check points a deadline with respect to the time counter in which the SE is running. Using the previous example of a SE with three check points, A, B and C, A needs to be reached at 5ms after the beginning of execution, B at 10ms and C at 15ms. If all these deadlines are respected, then the *Watchdog Manager* declares that there were no errors during execution. However, if one of the deadlines was not fulfilled, then the *Watchdog Manager* might declare that an error has occurred.

Similar to the program flow monitoring, it is also possible to declare a tolerance for the number of missed deadlines, therefore, if the tolerance is two missed deadlines and in the execution of the program there was one missed deadline, the *Watchdog Manager* will not warn the user of that missed deadline.

The defined deadlines have two associated values, a minimum and a maximum value. The check point should not be reached before the minimum nor after the maximum value.

The main purpose of Timing Monitoring is to check the temporal, dynamic behaviour of the supervised entity but it can also be useful to detect random or irregular jumps of the time-base tick counter. Moreover, it can also be an indicator that a program is taking too long to be executed and there might be something wrong [50].

3.3.5.3 Alive Supervision

The Alive Supervision is related to the Timing Monitoring since it is essentially a derivation of the latter. It checks how often a check point is reached and therefore if the SE is "alive". For example, the ECU might expect a periodic signal from a sensor, and a certain task requires that measurement which, if associated with a check point, would mean that during normal operation, the check point would be reached with a similar period as the signal from the sensor.

If the only objective of the user is to know if the SE is alive, it only needs to use one check point, but if a SE has several functions, each one with their functionality, the user might want to use one check point for each function and verify if they are being called as often as expected.

For this type of control, the user has to define other parameters besides the check points that define a SE. The user needs to state how many check points per cycle are there in a SE, the length of a supervision cycle and the lower and upper values for the tolerance for the number of expected alive indications [50].

3.3.5.4 Behaviour when an error occurred

When the *Watchdog Manager* has encountered an error, regardless of the monitoring method that encountered the error, it has three possible actions [50].

- It can send information regarding the detected fault;
- It can initiate a reset of the microcontroller after a watchdog timeout;
- It can immediately initiate a reset of the microcontroller.

3.4 ECU Monitoring

"Specification-based systems construct a representation of correct behaviour for an entity based on the current system policy together with the expected usage of the entity. The observed behaviour of the entity is compared to its specification, and any deviations are reported" [51]. What this means is that a set of specifications that represent the correct behaviour of the system is defined and the actual behaviour of the system will be compared to these specifications. The specifications can include the whole system or just a specific part of it. They need to be sufficient to describe a part or a function of the system, otherwise, the comparison might fail.

3.4.1 RTA-TRACE

The RTA-TRACE is a tool provided by ETAS, a company that specialises in embedded systems which allow the monitoring in real time of applications embedded in a specific hardware, such as an ECU.

This tool is installed in a computer and is comprised of two parts, the server and the client. The server is responsible for receiving the data and presenting it to the user and therefore is located on the computer. The client gathers, stores and sends the data to the server and is located in the to be monitored hardware. In the specific case of interest, in the ECU.

Regarding the AUTOSAR software structure within an ECU, this tool looks into the operating system, located in the Basic Software Layer. For more detailed information go to Chapter 2. It allows the user to control task activation's, resource-locks, alarms and user defined *tracepoints*.

The tracing is done by slightly altering the configuration files of the ECU and placing a set of events that will serve as the flags for the program running on the computer.

After the configuration and during runtime, the user is presented with an interface like the one in Figure 3.4 [52].

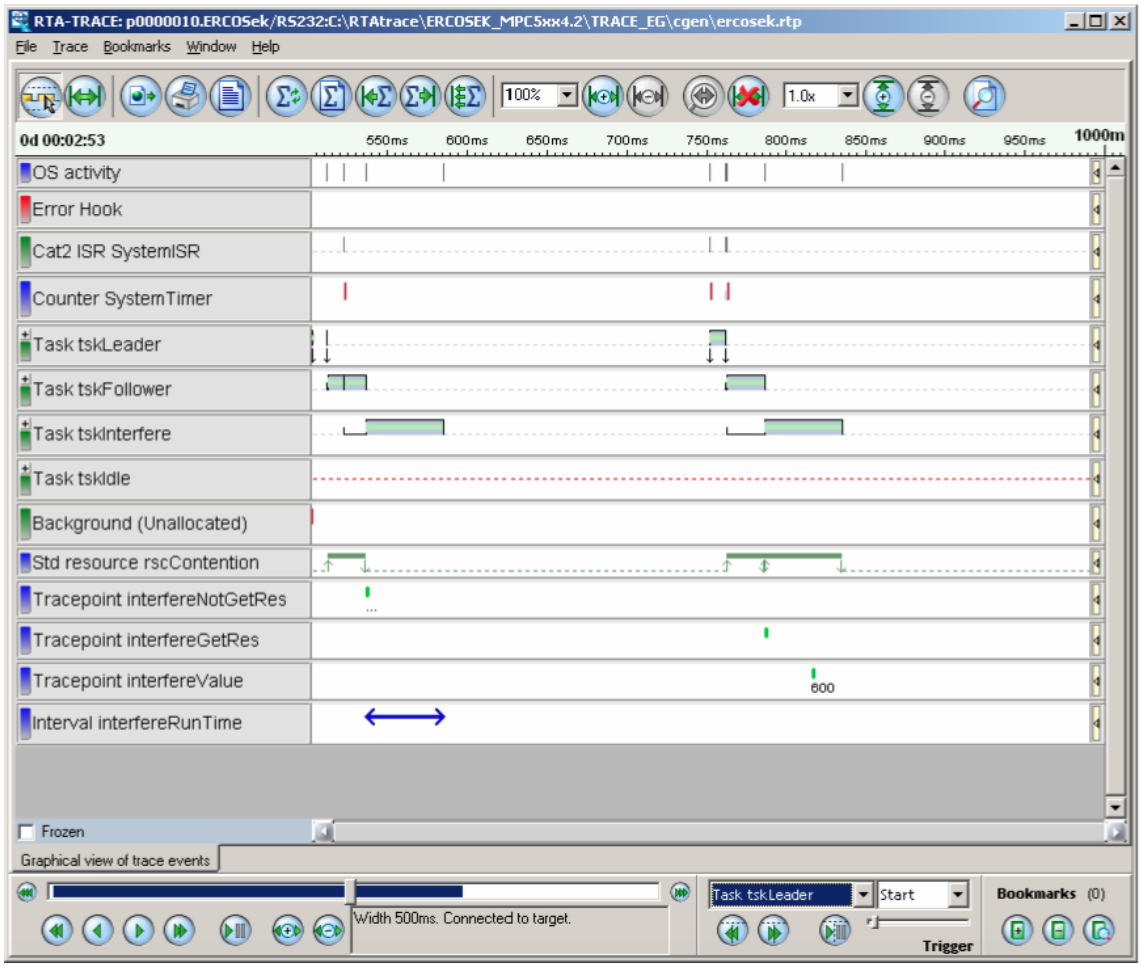

Figure 3.4: User Interface during runtime of the RTA-TRACE

So it is possible to see the occurrences of the specified events and functions which will allow the user to determine if the observed behaviour is within what was expected or not [53] [52] [54].

Patrick Frey [55] proposes the use of a tool in AUTOSAR, VFB Tracing and information regarding this tool is available at section 3.3, to monitor and model a Real-Time Control-System, in specific AUTOSAR.

After the given introduction to the tools and the system itself, Frey starts with the introduction of signal paths and how to obtain such paths. He defines a signal path as "the path between a sensor and an actuator that sense and affect the same physical quantity". To obtain such paths, the author uses the tool VFB Tracing available in AUTOSAR and its Hook functions. This tool allows the detection of a group of events, such as communication events, between SW-Cs and ECUs, the activation of tasks, which, when combined, correspond to the path that the signal uses from the sensor to the actuator.

An example of the final path that is obtained can be seen in Figure 3.5 [55]. Here it is possible

to see the receiving of a signal to initiate the application; then a task is initiated followed by the reading of sensor1. Then the adjustment of the throttle's position is made by reading and writing the desired position, and finally, a signal is sent to the actuator.

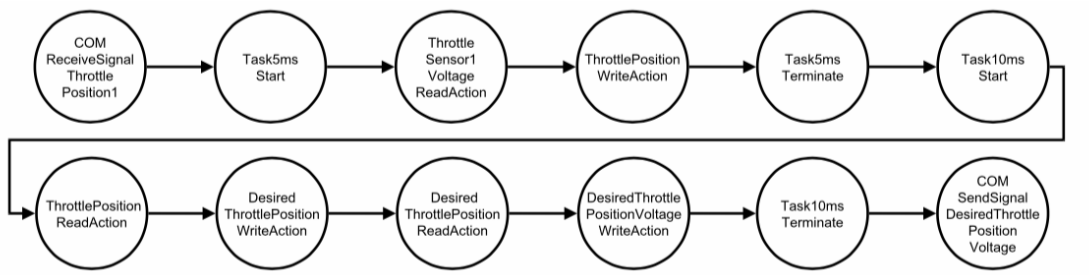

Figure 3.5: Obtained path of the throttle control application

For a more detailed description of how the author can obtain a path like the one in the Figure 3.5, the author introduces some new concepts in the AUTOSAR. Firstly the author defines the concept of event chain. An event chain specifies the order of occurrence of a set of events and action classes. This chain can be atomic or composite, similar to the SW-Cs introduced in Chapter 2. An atomic event chain is a chain comprised only of event and actions that belong to a single SW-C, while a composite event chain possesses one or more atomic chains. The event chain will be the structure in which the order of occurrence of the events and actions is saved. To help determine that order, the author introduces the measurement of temporal distances between events and action, more precisely the distance between events instances and action instances. Frey defines an event instance as being the time stamp of when an event class was visited, this event class is a specific place in the AUTOSAR. The same concepts are used for actions, where an action instance is the time stamp of when an action class was visited, an action class being a durational action marked by being started by one event class and terminated by another event class. These distances are of extreme importance since it will inform the author of the duration of action classes and the order in which events instances and action instances occur, being this order the one that will be used to create the event chains.

For the detection of the events and actions, the author uses, as previously stated, the VFB Tracing to detect the individual events and actions.

After obtaining the events and actions and their corresponding time stamp, the author uses an algorithm to build the actual event chain. This algorithm identifies all the feasible paths between the events, the restriction being their occurrence. The chosen path is the shortest one, and this is the one that will be converted into an event chain and later saved.

Chapter 4

Problem and solution outline

4.1 Introduction

This chapter is divided into two main sections.

The first section, Section4.2, gives a more detailed description of the problem and introduces the focus and how the solution presented in this document tries to solve that problem.

The second section, Section 4.3, describes the system in which this work is inserted followed by an explanation of every part of the system. The last subsection of this section is a comparison of this solution to what was introduced in the state of the art chapter, Chapter 3.

4.2 Description of the Problem

The problem that this dissertation tries to solve was briefly introduced in Chapter 1. With the rapid evolution in technology and the trend to have a global network connecting almost everything, even the automotive industry followed this trend with the implementation of Bluetooth and other wireless technologies onto a vehicle. Like every technology that has been created, this implementation has its advantages, but it also possesses downsides. One of the main ones is that the vehicle is now remotely accessed and this can be exploited. As it was also seen in Chapter 1, this exploitation has already started to happen, and it has been proven that it is possible to hack a car and override the driver. This can be done but taking advantage of weaknesses in the software and communications that are happening within the vehicle. This dissertation focuses on the software.

The choice of focusing on the software was made because by trying to bypass the driver and the normal behaviour of the car, the intruder influences that might be detected will have repercussions. Repercussions like the activation of tasks that were not supposed to run, a task might run longer than it was supposed, or the opposite, impede a task from running which might cause the brakes to malfunction. These modifications will have repercussions in the behaviour of the software present at the vehicle's ECUs.

Therefore, the solution proposed in this dissertation, monitors the behaviour of said software, by firstly learning what is considered to be the normal behaviour and then trying to detect any

anomaly, that is, behaviour that differs from what is regarded as normal but that it is not necessarily an error.

4.3 Structure of the chosen solution

This dissertation is inserted in a larger project whose purpose is to monitor different aspects of the vehicle, such as ECUs and the networks. The monitoring is done by not only monitoring the ECUs but also the network that connects the ECUs. Figure 4.1 shows a representation of the software responsible for the monitoring.

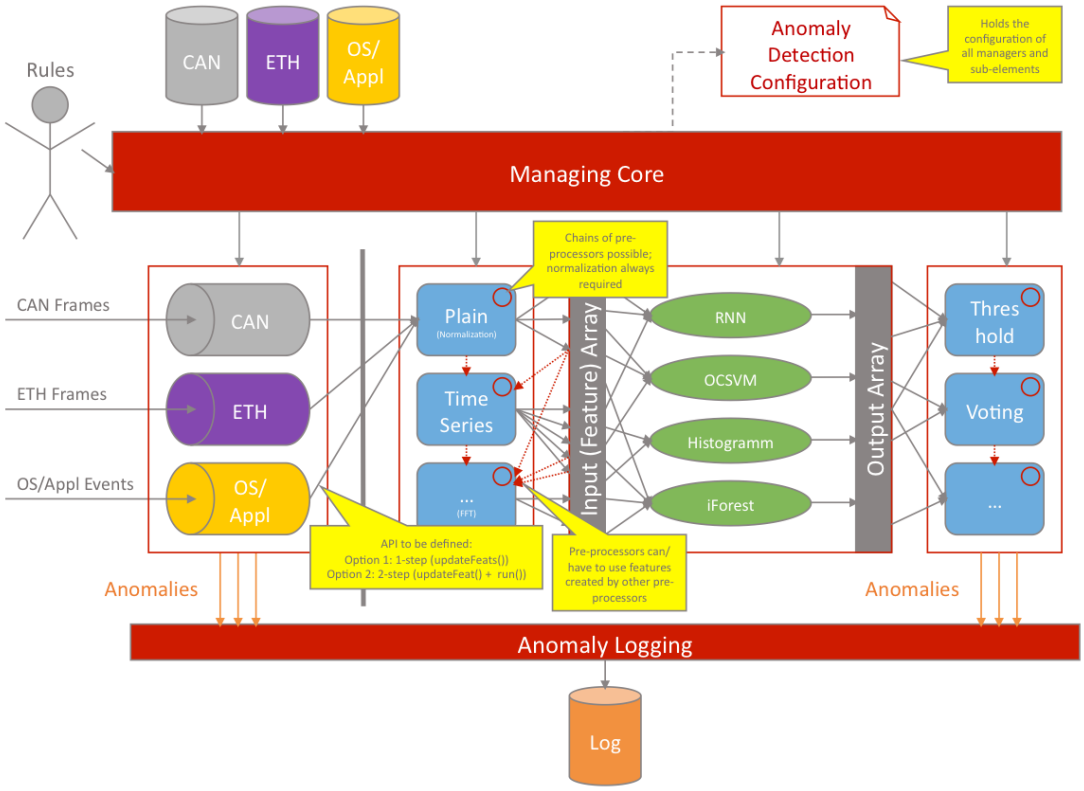

Figure 4.1: Structure of the monitoring software

The Managing Core (MC) is the kernel of the software. It is the part responsible for the activation of the entire system, thus, managing the whole system.

The rest of the system is divided into a sequence. First, the data to be analysed is received as an input. These data can be from the network, which can vary depending on the cables and the protocols used, such as CAN, Ethernet, FlexRay, or it can be from the ECUs' software. Depending on the received data, the MC activates the Static Check. The Static Check analyses the data and can filter possible anomalies that do not require a Machine Learning algorithm to be learned and detected. In the case of a network message, this could be a message that does not comply with

the rules of said protocol. As for the ECUs' software, it could be a task that is supposed to run cyclically and it misses a cycle.

Followed by the Static Check, the MC activates the required Pre-Processors for processing and preparing the new data. The preprocessing is required since the Machine Learning Algorithms use a specific format of data which does not correspond to the raw format in which the data is obtained. This pre-processing can be simply altering the structure of the data or, in a more intrusive manner, they can be processed using signal processing techniques, for example applying the Fourier Transform to the data.

Once the data has been treated, it is ready to be given as an input to the Machine Learning algorithms, which have been previously trained. The MC activates the machine learning core, where the machine learning algorithms exist, and then these algorithms analyse the data. It is important to note that there is more than one type of algorithm implemented, and, depending on the type of data, not all of them might be activated.

The final step is to analyse the outputs of the Machine Learning algorithms. The MC activates the different analysers that will decide if the original data was an anomaly or not. These analysers can be a simple voting system, where if the majority of the ML algorithms classified the data as anomalous, the data is considered as such. It can be a perceptron, where the algorithms and their respective outputs have a weight associated and the activation function will decide if the data was or not anomalous. Regardless of the type of analyser, the result is stored in a Log that can be later analysed by humans that decide the relevance of the data and how to proceed further.

This dissertation focused on the ECUs' software and from the whole system, it built the pre-processors for the data and ML algorithms for that data.

Comparing this solution to what was introduced in Chapter 3, it does not require a human observer, unlike RTA-TRACE which only shows what is happening. It can learn what is perceived as normal behaviour and independently detect anomalous behaviour after the training phase. It uses similar methods as the work by [55], but goes further since it tries to learn what are the normal sequences of tasks and SW-Cs and tries to detect any anomaly.

It focuses on the autonomous monitoring of the ECU's software which is also a novelty in the automotive industry, as far as the research could tell.

Chapter 5

Implementation

5.1 Introduction

This chapter enumerates all the steps taken in order to achieve the final results and how all the parts of the system were implemented.

It starts by listing the types of errors that could be detected in an ECU software, the information required to detected said errors and the tools necessary to obtain that information.

Following will be the obtaining of the information and the preprocessing done to it, required by the next phase, the Machine Learning algorithms.

The next section will explain the used algorithms, their working principle and the output of the algorithms.

5.2 Types of Errors

As seen in the previous chapter, 4, in a first step, the classification of the possible errors and the information needed to detect them was defined.

The classification of the errors can be shaped into a tree [56] [57]. In this tree it is possible to see that the errors can be divided into three main types, those that concern the values, those related to the time aspect and the final type, provision. Then, in each of these types, it is possible to further divide the errors into two sub-types. For the value type we have coarse, when the discrepancy in the value is significant, and subtle, when the discrepancy is subtle. Regarding the time type, we have early and late, so early can be an event that occurred earlier than it was expected and late if an event occurred later than it was expected. For the final type, provision, it can be of a sub-type, omission, which is when something never happened, and comission, which is the opposite, something did happen when it was not supposed to. The corresponding tree can be seen in the Figure 5.1.

This classification was then used regarding the OS tasks, SW-Cs, runnables and the overall system.

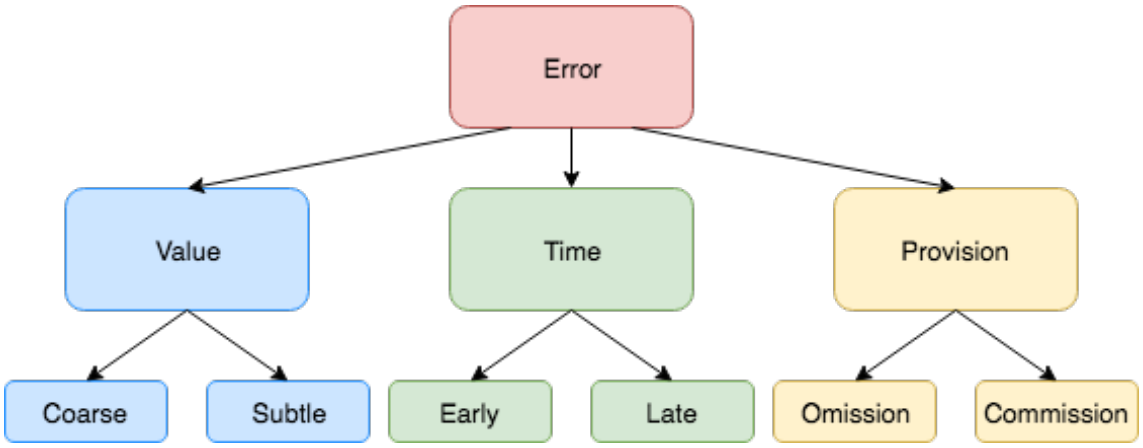

Figure 5.1: Classification of the errors

With respect to the OS tasks, the type of errors defined was divided into five different categories, Activation, Runtime, Interruption, Memory Use, and Service Calls. These categories are properties of the events and in those properties, the types of errors present in the previously introduced tree can occur. The activation category concerns the activation of the events; the runtime category regards the time that elapsed between the activation and termination of the events; the interruption category concerns if any interruption occurred during the execution of an event; memory use tells how much memory the event used; service calls concern possible service calls that were made by an event. An event can be any OS task, SW-Cs, runnables. Some of these properties can also be applied to the overall system.

Within the Activation category, it is possible to detect errors regarding the Time and Provision types. In the time type what can occur is the OS task was activated earlier or later than what it was supposed to. Regarding Provision, the task can be activated when it was not supposed to be activated, Commission or the task was never activated when it was supposed, Omission. In order to detect these errors, we need to know the time at which the task starts and the end time. That can be achieved by using the OS Hook functions, introduced in chapter 3, section 3.3.3.

Then comes the Runtime category. Here all three types of errors can occur. With respect to the Value type, it can be both coarse and subtle, that is, the task ran way too many times than it was expected or the task ran slightly faster or slower than what it was supposed, respectively. Regarding the Time type, the task may have ended earlier than it was supposed, thus having a shorter runtime, or it may have ended later than it was supposed, therefore having a longer runtime. Finally, the Provision type relates to the fact that the task had no runtime, this is, it terminated immediately after it began, or it ran when it should never had. This last type is similar to the Provision type regarding the Activation of the task. To detect these errors, the information needed is the time elapsed between the activation and termination of the task, which can also be obtained using the OS Hook functions.

The Interruption category relates to the interruption of a task. Again, all three types of errors can occur. Regarding the value type, it is Coarse when the task was interrupted significantly more

than what is deemed normal, and Subtle when the task is interrupted, either slightly more or less than normal. Regarding the time, the task can be interrupted earlier or later than normal, and finally, we have an Omission error when the task was never interrupted when it should have been, or Commission when the task was interrupted when it should not have been. For the detection of these errors, we need to know the number of interruptions that occurred during the execution of a task, and their respective time stamps. The information can be obtained using both OS Hook functions and the VFB Tracing, also previously introduced in chapter 3, section 3.3.4.

The Memory category regards the memory usage of the task. Also here it is possible to obtain all three types of errors. The Value type refers to the amount of memory used by the task when it differs wither greatly or slightly. The Time type refers to when the memory is used, either sooner or later than expected and finally, the Provision type, if the task used memory when it was not expected for the task to use or the opposite when the task never used memory when it should have. Unfortunately, due to the nature of the AUTOSAR software and the available tools, it is not possible to easily obtain information regarding memory usage. The only possible way would be to constantly check for the task pointer in the associated memory stack, but this method would be resource exhaustive, and it could alter the monitoring of the system.

Finally, there is the Service Calls category. This category is related to the Service Calls executed by the task. The Value type refers to the number of Service Calls executed by the task, either when the number varies in a greatly or slightly regarding the normal value. The time type indicates when a Service Call was either executed earlier or later than normal. The Provision type happens when the task, either, executes a Service Call when it should not have or it does not execute a Service Call when it should have. To detect these errors, we need to know the number of Service Calls that the task executed during its runtime and their respective time stamps. These information can be partially obtained using the OS Hooks. Partially because there is the need to distinct when a task was interrupted and when it actually ended.

Regarding the SW-Cs, the types of errors are very similar to the OS Tasks. The difference is that only the VFB Tracing provides the needed information. In order to know if an Activation type error occurred, we need to know the time at which the SW-C usually starts and the activation of the SW-C. Both can be obtained using the VFB Tracing. For the Runtime type, the information needed is the activation and termination of the SW-C, which can be obtained using the VFB Tracing. For the Interruption category

The last set of errors is related to the whole tasks or SW-Cs. All three types of errors can occur, Value, Time and Provision.

In the errors of the type value, the coarse is related to the number of tasks or SW-Cs that are running and occurs if the number differs greatly from what is considered the normal behaviour. If it only values slightly, it is the sub-type subtle.

In the type time, if a sequence starts occurs earlier then expected, it is of the sub-type Early, and if it starts later than expected, it is of the Chapter-type Late. Finally, if a sequence is executed when it should not have been, it is the sub-type commission and if it is not executed when it should have been, it is the sub-type Omission, both from the type Provision.

It is important to clarify, that all the errors of the type Value and sub-type Subtle, differ slightly but the difference is still bigger than a threshold. The difference is that the values are close to the threshold, and in the Coarse sub-type, the values are far away from the threshold.

Regarding the Memory category, it is also not feasible to trace and monitor the memory usage of a SW-C, not only because the SW-C is made of Runnable Entities, where each has its memory usage, but because we would need to constantly check the memory pointer for each RE. Finally, for the Service Calls, the information needed is the same as in the OS Tasks and can be partially obtained using the VFB Tracing.

The final possible origin of errors is related to the global aspect of the software. Here the objective is to analyse the sequence in which the tasks or SW-Cs are being executed, the total number of tasks or SW-Cs and if there is a difference to what is regarded as normal.

For the Value type and Coarse sub-type, that would mean that there is a big discrepancy between the number of tasks/SW-Cs running from what was supposed to be, either too many or too few. For the detection of these types of errors, the information required is the current number of tasks/SW-Cs running and the system time. This information can be compared to the equivalent information that was previously learned during the training phase.

For the Time type, there isn't any relevant error that can occur and be monitored.

For the provision type, there is the Commission sub-type, which is when a transition between tasks/SW-Cs happens that was never supposed to happen. The opposite can also happen, and that corresponds to the Omission sub-type, which occurs when a task/SW-C does not run when it should have. Both these cases might produce an anomalous subsequence. The information required to detect these errors is the sequence of tasks/SW-Cs that were and are being executed in the current use of the software.

5.3 Obtaining the Data and Pre-Processing it

The data used to train and test the algorithm was given by the advisor Marc Weber. The data was obtained by performing measurements during driving situations with a real car and therefore are not from a simulation.

The data is related to the accelerator pedal and its position. So it is a sequence of the position which is used as an input of an ECU that will then use that position to control engine and ultimately adjust the speed of the car.

The data is not ideal since it can be thought of as a black box where we only know the pedal position and do not know how the tasks are running and being executed in the ECU, but it is sequential and is a good starting point for analysing the sequences of tasks in an ECU.

From the errors previously introduced, these data allows the algorithm to look for anomalous sequences which represent the Provision type of errors for the overall software.

This type of error has a high importance since an anomalous sequence can indicate a malfunction of the vehicle or an unplanned behaviour in the vehicle induced by a third party that gained access to the vehicle. Due to this reason, this was the error that was chosen to be the main focus.

The data was acquired at a much faster rate than the pedal position changed and therefore it needed to be filtered. Due to this fact, the obtained values could only change every eight acquisitions, and the first step of the pre-processing was to eliminate the seven values in between every possible change.

The next and final step was to give the data a format that could be read and used by the algorithm in order to build the Decision Trees and the Graph. This was done by writing the positions into a text file and writing a letter t before every position. The letter t gives a uniformity to every position, thus, making it easier for the algorithm to know where every new position starts and to create the vertices' names.

5.4 Machine Learning Algorithms and their implementation

As previously mentioned in chapter 4, this dissertation is inserted into a bigger project. This fact means that there were some constraints related to the implementation. The main constraint was the language used to program and implement the algorithms. Once this dissertation started, part of the system was already built, and the language used was c#. The two main reasons for that decision were that it is an object oriented programming language and it had a good support of libraries regarding Machine Learning, the accord-framework [58].

Another library used to implement the algorithms presented in this dissertation was QuickGraph [59]. This library facilitates the creation of graphs by providing functions that allow the creation of the vertices and the edges.

As for the algorithms, the first to be implemented was the Decision Tree algorithm together with the Graph.

5.4.1 Merged Decision Trees

Due to the nature of AUTOSAR system and the fact that it is an event driven OS, where an event is followed by another event and so on, there is a natural sequence associated to the data. Therefore the first idea for an algorithm was to use decision trees, where the root node would be the first event that was activated in the OS, usually related to the initiation of the system, which means there exists some consistency in the system itself. Then, the tree begins to being built by analysing what are the events that can follow the root node. Given that the system is limited to a certain number of OS tasks, SW-Cs and runnables, and in theory there is a sub sequence that should occur every time an application, made by SW-Cs which in turn are made by runnables, is executed, there is also a very high repetition of sub sequences in the system. So the idea of after a pre defined depth of the trees, they start to merge and transform into a graph was used.

The algorithm starts by constructing a Decision Tree, and once the branch has reached the maximum depth previously defined, it becomes a graph.

The initial construction of the tree is similar to what was previously seen, the difference starts when the tree reaches its maximum depth. Once the maximum depth has been reached, the several trees, one for each different task, become a graph that merges all the trees.

The pseudocode of the algorithm can be seen below and is followed by an explanation.

Algorithm 1 Merged Decision Tree Algorithm

```
1:  input ← sequence of execution of the components
2:  for i ← 1, sequence length do
3:      for j ← i, sequence length do
4:          Update vertex name
5:          if vertex already exists then
6:              Check if the vertex is connected to the previous one
7:              if not then
8:                  create the edge
9:          else
10:             create the vertex
11:             Check if the vertex is connected to the previous one
12:             if not then
13:                 create the edge
```

Algorithm 1 represents the used algorithm to create the Decision Trees merged into a graph.

This algorithm receives as an input an array of strings. This array possesses the tasks that occurred and are saved in the array by order of occurrence. Therefore, if a sequence of tasks, such has *t1*, *t2*, *t3*, occurred, the input array will be: **Tasks[t1, t2, t3]**.

Then two for cycles are used in order to iterate over the input sequence. The first cycle, dictates where the sequence to be analysed and used to build the Tree starts. The second cycle goes through the sequence and builds the corresponding tree of the first task of that sequence, given by the first iteration.

After beginning both iterations, the algorithm obtains the name of the current node. This is done by merging the tasks names into a single string until the maximum depth is reached. As an example, if only one task was analysed, the node name is just the name of said task. But if two tasks have been analysed, the current tree node name is, for example, *t1t2*. The name grows until the maximum depth is reached, which was previously defined. Thus, if the maximum depth is five, the nodes' names will only have a maximum of five tasks, for example, *t1t2t3t2t3*. If the maximum depth has been reached, the first task is eliminated from the name, and the new task is added at the end.

Once the node name has been determined, the algorithm checks if the node already exists. It does so by recurring to a dictionary that contains all the node names. In c# a dictionary is a data structure that has keys and their associated values. The keys are unique and are only associated with one value, but a value can have more than one key. In this case, the keys are the node names and the values the edges that connect the nodes. So, the algorithm looks for the node names, the keys, and checks if they exist in the dictionary or not.

If they do not exist, the algorithm creates the new node and, if it is not the first node, the first task of the sequence being analysed, it links it to the previous node by creating an edge between them. If the node already exists, the algorithm checks if the current node and the previous node are already connected. If they are, it skips to the next iteration, if they are no, it links the previous

node to the node whose name is equal to the one being analysed, also by creating an edge between them.

The creation of the nodes and edges was done by recurring to the existent library QuickGraph [59].

Finally, the algorithm updates the name of the previous node by replacing it with what was the current node's name.

After the structure is built, it is the saved to two files, one containing the vertices and the other containing the edges.

It is also important to notice that the algorithm will build several trees with the same sequence. The reason for this is that during the testing phase and the runtime, an error can occur at any position of the new sequence. Thus the algorithm needs to be able to handle that situation and by creating the trees, the algorithm is able to recover from the error and start again from the event that followed the anomaly.

This algorithm is used every time there is a need to create a new structure. But in order to save resources and avoid redundancies, the algorithm above was slightly altered to a second algorithm whose function is to update a previously built structure.

The main difference is at the beginning of the algorithm. The updated algorithm starts by opening both files, the vertices and edge files. Then it quickly creates the dictionary from the files. After the dictionary is created, it then starts reading the new data and creating the trees. If the new vertices or edges already exist in the dictionary, they are ignored. If they do not exist, then a new node and or edge are created into a new graph. Once the new data has been fully analysed, the update algorithm updates the vertices and edges file with the new vertices and edges.

Algorithm 2 represents the pseudo code for the updated algorithm

Algorithm 2 Merged Decision Tree Algorithm

```
1:  input ← sequence of execution of the components
2:  Create a dictionary with the graphs saved
3:  for i ← 1, sequence length do
4:      for j ← i, sequence length do
5:          Update vertex name
6:          if vertex already exists then
7:              Check if the vertex is connected to the previous one
8:              if not then
9:                  create the edge
10:         else
11:             create the vertex
12:             Check if the vertex is connected to the previous one
13:             if not then
14:                 create the edge
```

The combined use of these algorithms allows for the creation of the Merged Decision Trees (MDTs) and its training.

5.5 Training and Testing of the Merged Decision Trees

The training was done in an iterative way. That means that instead of using all the data available for the training at once, the training data was divided into sets, and the algorithm was trained by receiving one set at a time. The reason for this choice was the possibility of testing the algorithm at the end of each iteration and observing the evolution and possible improvement in the algorithm.

The total data was separated into different files. Each file corresponded to a different driving situation. Each unprocessed file contained, on average 40000 values. These were processed, and the result became 5000, due to the repetition of the same value eight times in a row.

On the first iteration, the used algorithm was the **algorithm** 1. After the first iteration, the first Merged Decision Tree was obtained. Then the obtained structured was tested against a set of data.

Then a second file was used for the second iteration using **algorithm** 2. The final structure was tested with the same set of data that was used in the previous iteration.

This process was repeated until the majority of the data had been used to train the Merged Decision Trees. In the end, a final test was made in order to obtain the final results.

The training process is represented in Figure 5.2

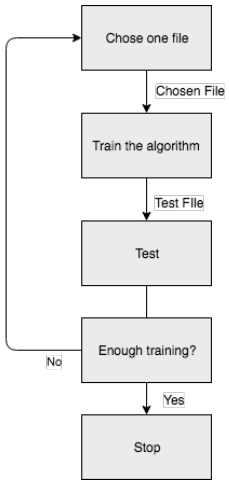

Figure 5.2: Training Process

Regarding the different types of training introduced in Chapter 2, this training can be considered supervised learning, since we know and have the data labelled as either normal data or anomalous data. It is important to notice that at this point, the data used to train the algorithm is solely normal data.

The actual testing was done by loading the created structure, similarly to what is done in Algorithm 2. Followed by receiving the sequence of tasks/SW-Cs/RE or one at a time, if it is in real time, and creating the vertex name. Then it checks in the graph for the existence of the vertex and if the vertex is associated with the transition that occurred, if it is not a root node. If both exist in the graph, the algorithm waits for the next task/SW-C/RE or moves in the sequence and repeats the process. If either the vertex or the edge was not present in the trained algorithm, it increments

an error counter and adds, either the vertex or the edge to an array. Once the process is terminated, the algorithm prints both the error counter and all the errors that it encountered.

The testing data was also obtained in a similar way to the normal data but errors were manually introduced in the data set. These errors were random numbers, some within the possible scope of the data and some that were impossible to normally occur and they were also randomly introduced between the normal data, without any particular distribution.

5.6 Improvement to the Merged Decision Trees Algorithm

5.6.1 Obtaining the Information

Upon receiving the software CANoe from Vector Informatik, the AUTOSAR software was modified to obtain the required information to be used by the algorithm.

The modifications used two of the mechanisms introduced in Chapter 3, more precisely, the System Hooks and the VFB Tracing.

In the System Hooks, three hooks were implemented, the *StartupHook*, the *PreTaskHook* and the *PostTaskHook*.

The *StartupHook* was used to obtain a list of existing tasks in the AUTOSAR. Since the system is static, that is, once the software is generated it cannot be changed, it is possible to obtain the existing tasks in the OS. This functionality was implemented in this hook function, thus, once the system initializes, an array containing all the tasks is created.

Both the *PreTaskHook* and *PostTaskHook* have similar implementations. They both obtain the Id of the task that is about to run or was terminated, and then they obtain the name of the task from the array containing the tasks by looking for the task in the position of the array corresponding to the Id. Finally, they save the task into a file that will contain the information that will be used either for training or testing.

Regarding the VFB Tracing, a hook function was implemented for both the start and return of every RE existing in the Main function of the simulated ECU. So if there is a RE called *MathMainFunction*, two hook functions were implemented for this RE, the *MathMainFunctionStart*, that indicates that this RE is about to start, and *MathMainFunctionReturn*, that indicates that the RE just terminated its activity.

All these hook functions concerning the activation or termination of RE are very similar. They all use the same function, ImplementHookFunction, which had two parameters. The first is an indicator telling if it is the Start or Return hook, and the second, the name of the RE. Then the ImplementHookFunction was implemented in a separate file. The function took both parameters, forming a new name, and saved that name in the same file where the tasks were being saved.

The second group of hook functions that were implemented in the VFB Tracing, in CANoe, concerned the API calls that the RE made to the RTE. The implementation of these hooks is very similar to the RE hooks, also having two parameters and calling a new function implemented in a separate file.

Once the functions were implemented, the system was ready to run and give as an output, a file containing the sequence of tasks, RE, and API calls made.

5.6.2 Improvement of the algorithm

The next step that was made was an improvement to the algorithm previously introduced. This improvement consisted in creating a hierarchy of files, each containing their own MDT. This means that during the process that previously corresponded to the creation of the MDT, now there will be several MDT, each in their separate file. The first file contains the MDT for the sequence of tasks. Then, for every task that has RE that ran, a new file is created, whose name is the same as the task name, that will also contain an MDT for the sequence of RE of this specific task. The same happens with the RE and API calls, every RE that had API calls, will have their file, whose name is the same as the RE, containing the sequence of API calls that were made by the RE.

During the testing phase, the algorithm will receive the entire sequence, containing both tasks, RE and API calls. Then it will start by creating the tasks sequence and verifying that sequence using the tasks file. Then, if a runnable appears, it will go to the file whose name is the same as the task containing the RE, and it will verify if the sequence of REs is normal. If during the verification of the REs sequences, an RE makes one or more API calls, then the algorithm will jump to the correct file, whose name is the same as the RE, and verify if the receiving sequence of API calls is present in the file or not. Once the API calls terminate, the algorithm returns to the task file containing the REs and once the REs of the task stop, the algorithm will return to the original file, the one containing the task sequences.

The pseudocode for the algorithm can be seen in 3. Similar to the first version of the algorithm, it also receives as input the sequence of tasks, RE and API calls, without any separation.

Then, the algorithm checks if the new input is a task, a RE or an API. Depending on what it is, it will update the assigned vertex name, in order to keep track of the subsequence. Then, it will verify if that subsequence has already occurred. If it has, it will verify if the previous subsequence that preceded the current subsequence is already connected in the MDT that is being built. If it hasn't, it creates the connection. If the current subsequence is a root node, the case were it is the first input, then there is no need to check for the preceding subsequence, since it is non existent.

If the subsequence is new, then a new vertex is created, and the verification of a possible prior subsequence is checked, as in the case of the vertex had already been added to the MDT.

Once the entire input sequence has been analysed, the algorithm saves the MDTs in several files. One containing the tasks MDT, then, one for each task that had RE containing the RE MDT and the same for RE that made API calls.

Figure 5.3 represents a simple example of the files that would be created by the algorithm. T*n* represent the tasks, R*n* represent RE and API*n* represent the API calls.

Regarding the testing of the improved algorithm, due to time constraints, it was not possible to fully test the algorithm and see its effectiveness with the data that was initially intended.

Algorithm 3 Merged Decision Tree Algorithm

```
1:  input ← sequence of execution of the components
2:  for i ← 1, sequence length do
3:      for j ← i, sequence length do
4:          Check if the new input is a task, a RE or an API call
5:          Update the correct vertex name                ▷ either for tasks, RE or API Calls
6:          if vertex already exists then
7:              Check if the vertex is connected to the previous one
8:              if not then
9:                  create the edge
10:         else
11:             create the vertex
12:             Check if the vertex is connected to the previous one
13:             if not then
14:                 create the edge
15: Save the tasks sequence
16: for i ← 1, number of tasks with RE do
17:     Save the RE sequence in a file named tasks[i]
18: for i ← 1, number of tasks with API calls do
19:     Save the API calls sequence in a file named RE[i]
```

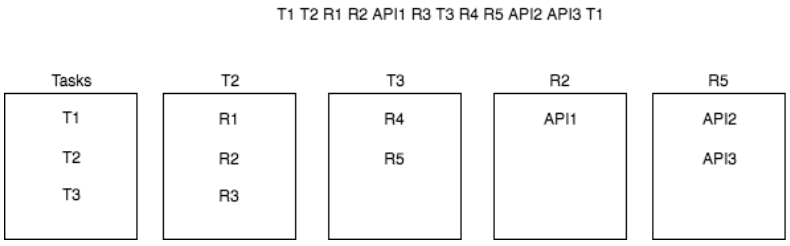

Figure 5.3: Final result of the training phase of the algorithm

5.6.3 One Class Support Vector Machine

The implementation of a One Class Support Vector Machine was started. This would be of the type introduced by Schölkopf *et al.* [34], where an hyperplane is defined that separates the data from the origin with the goal of having the largest margin possible.

This implementation was being done with the Accord Framework [58]. Due to time constraints, the OCSVM was not implemented. The previous algorithm was deemed as more important and relevant, and therefore the previously mentioned improvements to the previous algorithm were made.

Chapter 6

Obtained Results

6.1 Introduction

This chapter presents the obtained results and their discussion.

The results are presented according to their order of execution. Each result will be accompanied by an explanation of the meaning of the results and an evaluation of the result. After the first iteration of results, all the others will also be compared to the previously obtained results allowing for a better understanding of the evolution of the algorithm and the benefits of the training.

6.2 Obtained results

For the reasons explained in Chapter 5, only the Merged Decision Trees algorithm was implemented. In the same chapter, it was explained that the training and testing were done in an iterative way. That means that instead of using all the training data in one training instance, it was separated into different files and they were used one at a time. After the end of each training iteration, the resulting structure was tested with the same testing file, in order to keep the same variables. The reason for this choice was that by doing the iterative training, it was possible to observe the improvement of the structure and how it was improving.

The file used for testing had 3612 different values, all in sequence. From this number, 200 were manually introduced errors. These had values that did not respect the interval that the real values could assume, some were just empty lines, corresponding to missing values, and the rest were values that were possible to obtain the measurements. The last group was of interest because it could originate false negatives, that is, the label assigned by the algorithm would be a normal sequence when in reality it was an anomalous sequence.

6.2.1 First Iteration

The results for the first iteration can be seen in Table 6.1. After just the first iteration, the algorithm was able to detect all the 200 introduced errors. It did not misclassify the values that in the middle of the real values could pass as a normal sequence. Thus there were 0 false positives in total. It

did though, classify 358 subsequences as an anomaly when they were not. These are the false positives. Subsequences that are normal but were labelled as an anomaly.

Predicted Condition	True Condition	
	Anomaly	Normal
Anomaly	200	358
Normal	0	3054

Table 6.1: Results of the first iteration

From these results, it is possible to calculate some factors that help evaluate the algorithm, more precisely the relevance and accuracy of the algorithm.

To analyse the relevance of the algorithm, two parameters were chosen, the Recall and Precision. Recall can be calculated with the Formula 6.1 and the meaning is, from all the relevant data, in this case, the anomalies, present in the total data, how many of them were detected. Using the results from Table 6.1, the Recall value is 1. So all the anomalies were correctly labelled as such, and there were no False Positives, anomalies misclassified as normal data.

$$Recall = \frac{\sum TP}{\sum TP + \sum FN} \tag{6.1}$$

The Precision parameter evaluates the ratio of relevant data, the anomalies, in the total data classified as such. It can be calculated using Formula 6.2. A high Precision value means that from the data labelled as relevant, anomaly, the big majority are anomalies and were not misclassified. A small Precision value means that the from the data labelled as anomalous, the majority is in reality, normal. Using the results from Table 6.1, the Precision value is 0,358 or 35,8%. So the majority of the data classified as being an anomaly was in fact normal.

$$Precision = \frac{\sum TP}{\sum TP + \sum FP} \tag{6.2}$$

The relevance of the algorithm is the combination of both parameters. If the algorithm had a high Precision value but a low Recall value, it would mean that it was retrieving mostly anomalous data, but it was still missing most of it, therefore having a high number of False Negatives. If it was the opposite case, a Low Precision Value and a high Recall value, the algorithm was detecting most of the anomalies, but it was also classifying a significant amount of normal data as an anomaly.

The final parameter is the Accuracy, which tells how accurate the algorithm is. A high accuracy value means that the algorithm is labelling most of the data correctly, either as normal or as anomalous. A low Accuracy value represents an algorithm that is misclassifying a significant amount of data, either with a high value of False Negatives, False Positives or a combination of both. Using the results from Table 6.1, the Accuracy after one training instance is 0,901 or 90,1%.

$$Accuracy = \frac{\sum TP + \sum TN}{\sum T} \tag{6.3}$$

6.2.2 Second iteration to the tenth iteration

The results for the second to the tenth iteration can be seen in Table 6.2. TN stands for True Negative, which are the data that was correctly labelled as normal. TP stands for True Positive, which are the data that was correctly identified as being anomalous. FN means False Negatives, which occurs when the data are labelled as normal when it was, in fact, anomalous. FP stands for the False Positives, which are the data that was considered as anomalous but was in fact normal.

The same parameters analysed in the previous subsection can also be seen for every iteration. It is possible to see that the Recall value is constant, 1, which means that in every iteration, all the relevant data, the errors, were correctly labelled. The Precision increases with the number of training instances. This means that the algorithm is increasing the ratio of relevant data that is labelled as anomalous. This means that the number of false positives is decreasing, and therefore the algorithm is learning not to misclassify the normal data. With the increase in Precision, the accuracy of the algorithm also increases. Both these increases were expected, since the algorithm is using more data to compare with the testing case, and therefore, knows more normal cases which resulted in the decrease of False Positives.

Iteration Number	TN	TP	FN	FP	Recall (%)	Precision(%)	Accuracy(%)
2	3070	200	0	342	100	36,9	90,5
3	3091	200	0	321	100	38,4	91,1
4	3100	200	0	312	100	39,1	91,4
5	3112	200	0	300	100	40	91,7
6	3120	200	0	292	100	40,7	91,9
7	3127	200	0	285	100	41,2	92,1
8	3127	200	0	285	100	41,2	92,1
9	3128	200	0	284	100	41,3	92,1
10	3134	200	0	280	100	41,7	92,3

Table 6.2: Results form the second to the tenth iteration

6.2.3 Last iteration

The last training instance, the eleventh, produced the results that can be seen in Table 6.3. There were 3137 True Negatives, 200 True Positives, 277 False Positives and 0 False Negatives. Comparing to the previous iteration, the number of False Positives reduced again, increasing the Precision and Accuracy of the Algorithm. Using the results from Table 6.3, the Recall value is again 1, meaning that the algorithm keeps labelling all the anomalous data correctly. The Precision value is 0,419 or 41,9%. The Accuracy of the algorithm is now 0,924 or 92,4%. Both values increased again, meaning that the algorithm is becoming more accurate and relevant with every training iteration, as it was expected. The increase in the values is slow, and the justification will be presented in the next subsection as well as the discussion of the results.

Predicted Condition	True Condition	
	Anomaly	Normal
Anomaly	200	277
Normal	0	3137

Table 6.3: Results of the last iteration

6.2.4 Discussion of the Results

The obtained results can be seen in the previous subsections, in Tables 6.1, 6.2, and 6.3.

The Accuracy value was high from the first iteration which is a good indicator of the value that the algorithm has to the overall system since a high Accuracy value means that the algorithm is labelling the large majority of the data correctly.

The Recall values were always 1, which means that the algorithm never missed an anomaly by misclassifying it as normal data. This can be explained by analysing the algorithm itself. It receives a sequence of values, and from that sequence, it builds all the existent subsequences with length equal to the depth of the trees, a value that was previously defined. The meaning of this is that if the algorithm encounters a new subsequence during the testing phase, it will always classify it as being an anomaly. Therefore the only way for a False Negative case to appear as if the anomaly is equal to a normal case, which, from the definition of anomaly, cannot happen.

The Precision value also increased with almost every iteration, but it started and finished below 50%. This means that the majority of the data labelled as anomalous was in fact normal. The reason for this high rate of False Positives is in the algorithm and the nature of the data. As previously stated, if the algorithm encounters a new subsequence during the testing phase, it will classify it as being anomalous. Regarding the data, it had 1000 possible values that it could take, but analysing the data, from these 1000 different possibilities, only 172 were used. Nevertheless, these 172, in an extreme case, there are $7,66e17$ different subsequences of length 8. The data is not completely random and follows a certain pattern, but the number of possible subsequences is still very high which means it is expected that from iteration to iteration there will always exist new cases that will be the False Positives. The way to improve this factor is to train the algorithm extensively, decreasing the odds of it encountering new normal data.

With all these considerations, it is possible to say that the algorithm is accurate, with a final accuracy of 92,4%. It was also able to detect all the errors from the first iteration which is a good indicator of the relevance of the algorithm. The number of False Positives was, in the end, still high, but with more training data it should drop to lower values, closer to zero. The sensor data had some unpredictability associated, which does not happen in an ECU. This unpredictability regards the fact that the accelerator pedal, when pressed down, can assume a very large number of positions, and the same for when it is relived. So even if the driving path is the same, the position might vary by one degree or a tenth of a degree and that will represent a new case, which, if used during the testing phase, might represent a false positive case. Finally it is important to considerate that a false positive is preferred to a false negative. The reason is that these errors can occur in a

vehicle during usage and they might translate to the decrease in safety for the passengers and the extreme might result in casualties. Therefore, it is preferable to detect and record false positives that can be later analysed and ignored, than letting anomalies pass, false negatives which might be actual errors.

Chapter 7

Conclusion and Future Work

7.1 Introduction

This chapter is divided into two sections. The first is the conclusions of this dissertation and the comparison of the final results with the objectives. The second section presents suggestions for the future work that can be done to improve the resulting system and where the system could be used.

7.2 Conclusions

With the advances in the automobile industry regarding autonomous driving and the increase in connectivity of the vehicles, on one side, the safety of the passengers might increase due to the vehicles being able to analyse situations faster, but with the increase in connectivity, the security of the vehicles decreases and the safety can be put at risk.

From these conflict, the need for a solution that increases the security, and consequently the safety of the vehicle, arises. Some potential solutions started to be investigated, and this dissertation is one of them. The solution of which this dissertation is part of uses Machine Learning algorithms to learn what is the normal behaviour of the ECUs present in the vehicle and attempts to detect misbehaviour that might occur. It focuses both on the signals sent from one ECU to another, but it also tries to monitor the ECUs themselves, which is the focus of this dissertation.

The information used to learn the behaviour was the order in which the different SW-Cs, tasks and RE are executed by the ECU. A misbehaviour would be any sequence that does not conform to what normally happens.

The main goal of this dissertation was the implementation of a prototype that uses a Machine Learning algorithm to detect misbehaviour of the ECU. This main goal was achieved by first classifying the possible errors that could occur and be detected in an ECU. The figuring what information was required in order to detect them and how that information could be obtained.

Then, based on research of the existing Machine learning algorithms, two types of algorithms were chosen, the Decision Trees and the Graphs, and they were merged into a single algorithm.

The first version of the algorithm was implemented, tested, and the final results were presented in Chapter 6. The final results are satisfactory with an accuracy of 92,4%. The main reason for this value not being as high as expected was due to the nature of the data. A justification for this value was given in Chapter 6, but to summarise, it is mainly due to the nature of the sensor data and the randomness associated with it.

The algorithm was also capable, during the test phase of outputting an error log, showing the total number of errors and which transitions were labelled as an anomaly, thus, fulfilling another objective.

Then, once the CANoe software arrived, the first step made was to obtain the information. This was done using the mechanisms introduced in Chapter 3.

Once the information was being obtained without any problem, the previous algorithm underwent an improvement that allowed the algorithm to have some causal knowledge regarding the ECU behaviour. This means it is now able to know under which task the REs run, and what API calls each RE or task make.

Due to time constraints, it was not possible to test the improved algorithm, although it is expected that the results would improve and be much closer to 100%.

7.3 Future Work

Regarding future work, the first step would be to properly test the improved algorithm with several different use cases, such as, a new RE being introduced, a new task with RE that were introduced by a third party, and therefore might correspond to a possible attack, alter the behaviour of RE, change tasks and RE priorities.

A second step would be to implement a One Class Support Vector Machines, OCSVM, for the tasks in order to analyse their features. This would give the prototype the ability to detect all the defined errors in Chapter 5.

This solution could not only improve the security and safety of the vehicle, but it could allow the industry to go one step further. Since it uses machine learning, it could learn the behaviour of a specific driver which, allied with the increase in connectivity, could distinguish when the driver is incapacitated to drive, could be able to detect carjacking, among others.

References

[1] R.R. Teetor. Speed control device for resisting operation of the accelerator, August 22 1950. US Patent 2,519,859. URL: https://www.google.com/patents/US2519859.

[2] All Tesla Cars Being Produced Now Have Full Self-Driving Hardware, October 2016. URL: https://www.tesla.com/en_GB/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware.

[3] S. Biswas, R. Tatchikou, and F. Dion. Vehicle-to-vehicle wireless communication protocols for enhancing highway traffic safety. *IEEE Communications Magazine*, 44(1):74–82, January 2006. doi:10.1109/MCOM.2006.1580935.

[4] M. Steger, M. Karner, J. Hillebrand, W. Rom, C. Boano, and K. Römer. Generic framework enabling secure and efficient automotive wireless SW updates. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2016. doi:10.1109/ETFA.2016.7733575.

[5] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive {CAN} networks—practical examples and selected short-term countermeasures. *Reliability Engineering e System Safety*, 96(1):11 – 25, 2011. Special Issue on Safecomp 2008. URL: //www.sciencedirect.com/science/article/pii/S0951832010001602, doi:http://dx.doi.org/10.1016/j.ress.2010.06.026.

[6] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society. URL: http://dx.doi.org/10.1109/SP.2010.34, doi:10.1109/SP.2010.34.

[7] Sophie Curtis. Hacker remotely crashes Jeep from 10 miles away. *The Telegraph*, July 2015. URL: http://www.telegraph.co.uk/news/worldnews/northamerica/usa/11754089/Hacker-remotely-crashes-Jeep-from-10-miles-away.html.

[8] Y. Harata, Y. Nishimura, M. Kimura, H. Nagase, and M. Takigawa. A simplified serial communication network within a vehicle. In *IEEE 39th Vehicular Technology Conference*, pages 437–442 vol.1, May 1989. doi:10.1109/VETEC.1989.40110.

[9] CANoe - ECU Development & Test. Available at https://vector.com/vi_canoe_en.html. URL: https://vector.com/vi_canoe_en.html.

[10] AUTOSAR: Motivation & Goals. http://www.autosar.org/about/basics/motivation-goals/. URL: http://www.autosar.org/about/basics/motivation-goals/.

[11] AUTOSAR: Background. http://www.autosar.org/about/basics/background/. URL: http://www.autosar.org/about/basics/background/.

[12] OSEK/VDX. OSEK/VDX Operating System Specification 2.2.3, February 2005. Available at http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf. URL: http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf.

[13] ETAS. ETAS - Innovative solutions, engineering services, consulting, and support for the development of embedded systems. https://www.etas.com/en/. URL: https://www.etas.com/en/.

[14] Vector. Vector: Software + Services for Automotive Engineering. https://vector.com. URL: https://vector.com.

[15] Elektrobit Automotive - embedded software for cars. https://www.elektrobit.com/. URL: https://www.elektrobit.com/.

[16] The EDA Technology Leader. https://www.mentor.com/. URL: https://www.mentor.com/.

[17] AUTOSAR. Layered Software Architecture Release 4.3.0, November 2016. Classic Platform Document Status: Final DOcument Identification No 053 Available at http://www.autosar.org/standards/classic-platform/release-43/software-architecture/general/. URL: http://www.autosar.org/standards/classic-platform/release-43/software-architecture/general/.

[18] AUTOSAR. Software Component Template Release 4.3.0, November 2016. Classic Platform Document Status: Final Document Identification No 062 Available at http://www.autosar.org/standards/classic-platform/release-43/methodology-and-templates/templates/. URL: http://www.autosar.org/standards/classic-platform/release-43/methodology-and-templates/templates/.

[19] AUTOSAR. Specification of Operating System Release 4.3.0, November 2016. Classic Platform Document Status: Final Document Identification No 034 Available at http://www.autosar.org/standards/classic-platform/release-43/software-architecture/system-services/. URL: http://www.autosar.org/standards/classic-platform/release-43/software-architecture/system-services/.

[20] AUTOSAR. Virtual Function Bus Release 4.3.0, November 2016. Classic Platform Document Status: Final Document Identification No 056 Available at http://www.autosar.org/standards/classic-platform/release-43/main/. URL: http://www.autosar.org/standards/classic-platform/release-43/main/.

[21] AUTOSAR. Specification of RTE Release 4.3.0, November 2016. Classic Platform Document Status: Final Document Identification No 084 Available at http://www.autosar.org/standards/classic-platform/release-43/software-architecture/rte/. URL: http://www.autosar.org/standards/classic-platform/release-43/software-architecture/rte/.

[22] AUTOSAR. General Specification of Basic Software Modules Release 4.3.0, November 2016. Classic Platform Document Status: Final Document Identification No 578 Available at http://www.autosar.org/standards/classic-platform/release-43/software-architecture/general/. URL: http://www.autosar.org/standards/classic-platform/release-43/software-architecture/general/.

[23] AUTOSAR: Motivation & Goals. URL: http://www.autosar.org/about/basics/motivation-goals/.

[24] F. Khenfri, K. Chaaban, and M. Chetto. A novel heuristic algorithm for mapping autosar runnables to tasks. In *2015 International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*, pages 1–8, Feb 2015.

[25] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959. URL: http://dx.doi.org/10.1147/rd.33.0210, doi:10.1147/rd.33.0210.

[26] A. M. Turing. Computing machinery and intelligence. *Mind*, October 1950.

[27] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[28] V. Vapnik and A. Lerner. Pattern Recognition using Generalized Portrait Method. *Automation and Remote Control*, 24, 1963.

[29] Seppo Linnainmaa. Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden taylor-kehitelmänä. Master's thesis, University of Helsinki, 1970.

[30] Hans Peter Moravec. *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8024717.

[31] Gerald DeJong. Generalizations based on explanations. In *IJCAI81, the Seventh International Joint Conference on Artificial Intelligence*, pages 67–69, 1981.

[32] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986. URL: http://dx.doi.org/10.1023/A:1022643204877, doi:10.1023/A:1022643204877.

[33] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM. URL: http://doi.acm.org/10.1145/130385.130401, doi:10.1145/130385.130401.

[34] Bernhard Schölkopf, Robert Williamson, Alex Smola, John Shawe-Taylor, and John Platt. Support vector method for novelty detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 582–588, Cambridge, MA, USA, 1999. MIT Press. URL: http://dl.acm.org/citation.cfm?id=3009657.3009740.

[35] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, Sept 2006. doi:10.1109/N-SSC.2006.4785860.

[36] Dave Lee. Google's AI beats world Go champion in first of five matches. *BBC News*, March 2016. URL: http://www.bbc.com/news/technology-35761246.

[37] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Englewood Cliffs, N.J, 1995.

[38] Finn V. Jensen. Bayesian networks basics.

[39] Pieter Abbeel Stuart Russell Dylan Hadfield-Menell, Anca Dragan. Cooperative inverse reinforcement learning. November 2016.

[40] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, New York, 1997.

[41] Lior Rokach and Oded Maimon. *Data mining with decision trees: theory and applications*. World Scientific, Hackensack, New Jersey, second edition edition, 2015.

[42] Badr Hssina, Abdelkarim Merbouha, Hanane Ezzikouri, and Mohammed Erritali. A comparative study of decision tree ID3 and C4.5. *International Journal of Advanced Computer Science and Applications*, 4(2), 2014. URL: http://thesai.org/Publications/ViewPaper?Volume=4&Issue=2&Code=SpecialIssue&SerialNo=3, doi:10.14569/SpecialIssue.2014.040203.

[43] Rutvija Pandya and Jayati Pandya. C5. 0 Algorithm to Improved Decision Tree with Feature Selection and Reduced Error Pruning. *International Journal of Computer Applications*, 117(16):18–21, May 2015. URL: http://research.ijcaonline.org/volume117/number16/pxc3903318.pdf, doi:10.5120/20639-3318.

[44] Christopher M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, 2006.

[45] Michael I. Jordan. CS281b/Stat241b: Advanced Topics in Learning & Decision Making The Kernel Trick, 2004.

[46] David M. J. Tax and Robert P. W. Duin. Support vector data description. *Mach. Learn.*, 54(1):45–66, January 2004. URL: http://dx.doi.org/10.1023/B:MACH.0000008084.60811.49, doi:10.1023/B:MACH.0000008084.60811.49.

[47] Yi Lu Murphey, M. A. Masrur, ZhiHang Chen, and Baifang Zhang. Model-based fault diagnosis in electric drives using machine learning. *IEEE/ASME Transactions on Mechatronics*, 11(3):290–303, June 2006. doi:10.1109/TMECH.2006.875568.

[48] Guobin Ou and Yi Lu Murphey. Multi-class pattern classification using neural networks. *Pattern Recognition*, 40(1):4 – 18, 2007. URL: //www.sciencedirect.com/science/article/pii/S0031320306002081, doi:http://dx.doi.org/10.1016/j.patcog.2006.04.041.

[49] Yaokai Feng, Yoshiaki Hori, and Kouichi Sakurai. *A Behavior-Based Online Engine for Detecting Distributed Cyber-Attacks*, pages 79–89. Springer International Publishing, Cham, 2017. URL: http://dx.doi.org/10.1007/978-3-319-56549-1_7, doi:10.1007/978-3-319-56549-1_7.

[50] Daniel Ritcher Christian Leder. Microsar wdgm, technical reference, 2016. Version 1.1.0, Vector Informatik.

[51] U. E. Larson, D. K. Nilsson, and E. Jonsson. An approach to specification-based attack detection for in-vehicle networks. In *2008 IEEE Intelligent Vehicles Symposium*, pages 220–225, June 2008. doi:10.1109/IVS.2008.4621263.

[52] ETAS Group. RTA_trace Getting Started Guide, January 2009. Available at https://www.etas.com/en/downloadcenter/7046.php.

[53] ETAS Group. RTA_trace User Manual, December 2008. Available at https://www.etas.com/en/downloadcenter/7044.php. URL: https://www.etas.com/en/products/download_center.php?productfamily=5F6317EE59CD497B9F17E3E9EB383540&product=D28A9769203A4A6585EFC717396BE10F&doctype=null.

[54] ETAS Group. RTA_trace Configuration Guide, January 2009. Available at https://www.etas.com/en/downloadcenter/7735.php. URL: https://www.etas.com/en/products/download_center.php?productfamily=5F6317EE59CD497B9F17E3E9EB383540&product=D28A9769203A4A6585EFC717396BE10F&doctype=null.

[55] Patrick Frey. *A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems*. Dissertation, Universität Ulm, January 2011. URL: https://oparu.uni-ulm.de/xmlui/handle/123456789/1770.

[56] A. Bondavalli and L. Simoncini. Failure classification with respect to detection. In *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–53, Sep 1990. doi:10.1109/FTDCS.1990.138293.

[57] J. A. McDermid and D. J. Pumfrey. A development of hazard analysis to aid software design. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 17–25, Jun 1994. doi:10.1109/CMPASS.1994.318470.

[58] Accord.NET Machine Learning Framework. URL: http://accord-framework.net/.

[59] QuickGraph, Graph Data Structures And Algorithms for .NET. URL: https://quickgraph.codeplex.com/Wikipage?ProjectName=quickgraph.