Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Filipe da Costa Capa

Integration of ROS2 with a simulation environment

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Filipe da Costa Capa

Integration of ROS2 with a simulation environment

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor Doutor Paulo Francisco Silva Cardoso
Doutor Adriano Dídimo Machado Carvalho

December 2021

ACKNOWLEDGEMENTS

First, I would like to thank my thesis advisor, Dr. Prof. Paulo Cardoso. His input during the research and writing was valuable.

I would also like to acknowledge Dr. Adriano Carvalho for his comments as the second reader of the dissertation.

Finally, I would like to thank my family, especially my parents, for their support during this year.

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 039334; Funding Reference: POCI-01-0247-FEDER-039334].

STATEMENT OF INTEGRITY

ABSTRACT

Currently, the University of Minho owns a driving simulator, from now on referred to as Driving Simulator Mockup 2-Wheeler (DSM-2W), which mimics a real driving environment for motorcycles. This simulator can reproduce diverse driving scenarios, like driving on different roads, traffic, and weather conditions, and is mostly used to test how the driver reacts to stimulus from subsystems under test in a particular scenario.

The simulator has several components, namely, the Mock-up, which represents the motorcycle physically, the software responsible for the simulation environment, that is also projected on a screen, called SILAB [1] as well as several other subsystems and respective software, which all together form a complex distributed system. SILAB creates realistic graphic environments, has different models to control the behavior of other drivers and pedestrians, generates 3D sounds, and facilitates the personalization of the simulation scenario.

Robot Operating System 2 (ROS2) [2] provides a set of tools and software libraries that facilitate the development of robot systems and applications. With the increasing reliance on software, sensors, and actuators in the automotive domain, it makes sense to view cars [3] and motorcycles as robots. Therefore, it also makes sense to use ROS2 in the simulation domain to solve the problems at hand.

This dissertation describes how ROS2, a well-known and accepted middleware for robotic applications, can also play a role in these contexts acting as a universal interface between motorcycle simulators and external subsystems and thereby significantly improving the system's expansibility and those subsystems' portability and reusability.

KEYWORDS SILAB, ROS2, driving simulator, middleware, DSM-2W.

RESUMO

A Universidade do Minho possui um simulador de motas, denominado Driving Simulator Mockup 2-Wheeler (DSM-2W), que imita um ambiente real de condução de motas. Esta ferramenta consegue reproduzir diversos cenários de condução, como conduzir em diferentes condições de estrada, tráfego, bem como em diferentes condições meteorológicas. Esta ferramenta é sobretudo usada para testar como o condutor reage a estímulos de vários sub-sistemas em teste em cenários particulares.

O simulador possui diversos componentes, o Mock-up, que representa a mota fisicamente, o software responsável pela projeção do ambiente de simulação no ecrã, chamado SILAB [1], mais um conjunto de sub-sistemas e o respetivo software, que no conjunto formam um complexo sistema distribuído. O SILAB cria ambientes de simulação realistas, tem diferentes modelos para controlar o comportamento dos outros condutores e dos pedestres, gera sons 3D e facilita a personalização do cenário da simulação.

O Robot Operating System 2 (ROS2) possui um conjunto de ferramentas e bibliotecas para desenvolver aplicações para robôs [2]. Com o aumento do uso de software, sensores, e atuadores no contexto automóvel, faz sentido equiparar veículos automóveis [3] e motas a robôs Portanto, também faz sentido usar o ROS2 para resolver problemas neste contexto.

O objetivo desta dissertação passa por mostrar como o ROS2, um middleware bastante utilizado em aplicações para robôs, pode ter um papel importante em contextos de simulação ao atuar como uma interface universal entre sub-sistemas a testar e um simulador de motas e consequentemente melhorar a extensibilidade do simulador e a portabilidade e reusabilidade desses sub-sistemas.

PALAVRAS-CHAVE SILAB, ROS2, simulador de condução, middleware, DSM-2W.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

In this chapter, the context is explained as well as the motivations and the goals of this dissertation. Finally, an overview of the entire dissertation is illustrated.

1.1 CONTEXT

Nowadays, maintaining concentration while driving and riding is becoming even more complicated. The evolution of technology now allows motorcycle vendors to provide more and more interfaces that take away the driver's focus, such as navigation, multimedia, and telephone systems. On top of that, during the last few years, the increase of smartphone use while driving and riding [4; 5] became another source of distraction that reduces the rider's attention. This lack of concentration and attention affects riding performance and safety, which is one of the main causes of accidents [6].

In this way, verifying if a new interface is ready to be integrated or still needs to be tuned in order to be safe for general use is crucial. Even though a simulation environment is not the same as in the real world, it has, among other advantages, the possibility to test many scenarios without risking the rider's and the public's safety, to learn more about subsystems under test, and to validate their use. As a result, several motorcycle simulators have been developed for this and other research purposes. These simulators can be used to verify how the rider reacts to the new interfaces' stimulus [7; 8; 9], e.g., to test if the focus is maintained at a safe level.

1.2 MOTIVATION

The University of Minho owns a riding simulator, from now on referred to as Driving Simulator Mockup 2-Wheeler (DSM-2W). DSM-2W uses SILAB [1] as the main software responsible for the simulation environment. Simulation software, like SILAB, usually offers an Application Program Interface (API) to export their functionalities in order to integrate new subsystems to validate and evaluate how riders respond to them. This approach can constrain the development of new subsystems since the development teams have to study the simulator's specific API and specifications to integrate them, making them less portable and reusable. Using a well-known interface can be a solution to this problem since new subsystems do not need to know the specific simulator or software being used, nor its API and specifications. Those subsystems and the simulator can rely on a non-specific and standard interface to interact. This approach eases the integration of new subsystems with a real vehicle and with other simulators that use the same standard interface and, consequently, increases the simulator's expandability.

Robot Operating System 2 (ROS2), a set of software libraries and tools, was initially designed to build applications for robots [2]. Currently, however, it is increasingly common to find ROS2 in the automotive domain as the communication middleware between different components in the vehicle [3]. Besides, ROS2 is supported and used by a large community, has been used in academic and industrial projects, and is the proposed common framework of the RoboCup competition [10]. With this in mind, relying on ROS2 in the DSM-2W as a flexible and distributed interface to integrate subsystem make sense.

1.3 OBJECTIVES

The main objective of this dissertation is to offer a well-known interface to SILAB, more specifically ROS2, improving its expandability. This integration has as main advantage the use of a well-known interface facilitating the integration of software and heterogeneous hardware. From this integration, a new architecture emerges, where ROS2 acts as SILAB's abstraction layer, i.e., SILAB's non-standard interface is "hidden" behind ROS2's standard. Since SILAB is not designed for this approach, the implementation of this architecture is the main challenge of this dissertation.

In order to validate the architecture, a set of use cases will be integrated onto the DSM-2W through ROS2. Some use cases are simulation outputs, e.g., a group of LEDs in the rider's jacket to warn others that the rider is pressing the brakes. On the other hand, other use cases are simulation inputs, e.g., the activation of the turn indicator when the rider makes a specific gesture using the hand and the rest of the arm.

1.4 DISSERTATION ORGANIZATION

This dissertation is structured as follows.

In Chapter 2, driving and riding simulators' characteristics as well their extension mechanisms are presented. Then, some middleware is presented. Besides, the underlying concepts, features, and how to develop applications with ROS2 is described.

In Chapter 3, the concepts, configuration model, and how to develop software modules in SILAB are described. Besides, the most relevant components of the DSM-2W are presented.

In Chapter 4, the advantages of a ROS2-based solution over the extension mechanisms already offered by SILAB are described. Besides, in this section, the proposed solution, the structure of a ROS2 SILAB crossover software module, and a tool to ease their development are presented.

In Chapter 5, several use cases which validate the idea of integration of a well-known interface onto a simulator environment as well the advantages observed during the development are presented.

Finally, in Chapter 6, the conclusions regarding the work done in this dissertation and the future work are presented.

2

STATE OF THE ART

In this chapter, the technologies that make up this dissertation is presented. First, several driving and motorcycle simulators are exposed. Then, some middleware is presented. Finally, ROS2 and its concepts are described.

2.1 DRIVING SIMULATORS

Nowadays, simulation is used in the most varied contexts, such as automobiles and motorcycles. Usually, simulators try to offer a realistic experience by displaying a virtual environment and integrating underlying hardware in a Mock-up of a real car or motorcycle. External devices are often part of these simulator environments to make them more realistic and test new concepts and ideas. Although capturing data from the real world is more precise and valuable, it is not always practical, repeatable, or the public and subject's safety can be compromised in particular situations.

In the last few years, the interest and awareness of government entities to support research in this area increased. An example was the support of the Cooperation in Science and Technology Action (COST Action) by the European Union to help research the safety of powered two-wheelers with several studies using motorcycle simulators [11].

The next sections describe some known driving and motorcycle simulators' and their extension mechanisms, i.e., how to add new subsystems for human factors research onto the simulators.

2.1.1 *MORIS motorcycle Simulator*

One of the first motorcycle simulators is called MORIS [12] and was part of the Esprit project. This motorcycle simulator emerged from a partnership between industry and academic partners. MORIS is designed as a tool to acquire data on motorcycle handling and stability and to collect data about the rider's behavior. As a result, reducing the number of road tests required for motorcycle components prototypes is expected.

The simulator tries to give the rider the most accurate sensations as those perceived using a real motorcycle by offering visual, auditory, and several types of movement sensations. Visually, the rider receives feedback from a graphical interface that represents the virtual scenario. Regarding auditory feedback, the simulator produces realistic sounds from the virtual environment. In addition, the Mock-up has a motion-based platform to reproduce the dynamics of a real motorcycle allowing movement sensations. The platform provides movement sensations by having several degrees of freedom: vertical, lateral, and longitudinal displacements and roll, pitch, and yaw angles.

Regarding the simulator's extension mechanisms, despite the simulator's goal being to test motorcycle components prototypes, how difficult it is to add a new subsystem or if it is possible to build custom software modules is not referenced.

2.1.2 *SafeBike*

The University of Padua also built a motorcycle simulator [13] with the goal of studying the rider's behavior in different scenarios, train riders, and test and validate the addition of new subsystems. This simulator has realistic vehicle dynamics and provides constant feedback to the rider. The vehicle dynamics are capable of several types of movement, like counter-steering, capsize, weave and wobble instabilities, wheeling, skidding, kick-back. Regarding the feedback to the rider, the simulator is capable of displaying a realistic simulation environment in a 180º cylindrical screen and 5.1 surrounding sound of the engine, wind, and other audio cues.

The simulator has a Mock-up equipped with several sensors to detect the steering torque, the body leaning, the throttle, the front brake, the rear brake pedal, the clutch, and the gearshift lever. The inputs allow the rider to control the motorcycle in the virtual environment to move the Mock-up and produce audiovisual cues. In addition, the virtual scenario engine allows to design scenarios with specific rules to control the behavior of other vehicles, traffic lights, and pedestrians. Integrating subsystems and external devices is possible through a Controller Area Network (CAN) bus.

2.1.3 *MotorcycleSim*

MotorcycleSim [14] is a motorcycle simulator developed by the University of Nottingham to research motorcycle ergonomics and human factors. These purposes include testing motorcycle and rider's equipment prototypes, road safety, education, accident investigation, and rider's skills, attitudes, and behavior.

The simulator uses the STISIM Drive® software, which is widely used for car driving simulation but also supports a variety of motorcycle characteristics. The STISIM Drive® software supports gaming wheels to control the vehicle in the simulation environment, but motorcycles are different from cars, e.g., motorcycles have two brakes. For this reason, the motorcycle controls had to be interfaced with a gaming wheel, namely the throttle, braking, gear selection, and steering angle inputs. The software allows to personalize scenarios for controlled experiments and recording different measurements, e.g., velocity and acceleration. This simulator uses the Mock-up's input to update and project the virtual environment onto a screen in front of the rider. The visual effects displayed on the screen give acceleration and braking effects by changing the pitch degree, the tilt to enhance the leaning perception and the rider's field of view. The virtual environment also outputs 5.1 realistic surrounding sound for a more immersive experience. Also, MotorcycleSim supports positive steering (turn the handlebar in the direction to turn) and counter-steering (lean the motorcycle to the opposite direction the handlebar is turned). Since MotorcycleSim is based on the STISIM Drive® software, adding new subsystems to the simulator by creating custom software modules should be possible. Besides, no other extension mechanism is referenced.

2.1.4 *Cruden*

In 2007, Cruden developed its first motorcycle simulator prototype, but with the advance of technology, a new version was built [15]. The last version of the simulator has a Mock-up of a real motorcycle mounted on top of a Stewart motion platform with six degrees of freedom. The Mock-up consists of six rigid bodies connected by seven joints, developed using Simscape Multibody™ toolbox. The Mock-up combines a Honda CB-750 motorcycle with several Cruden-made components.

The simulator has two cameras mounted on the top of the platform to estimate the upper body position, allowing the rider to maneuver the motorcycle without the handlebar. The visual feedback is given to the rider by a head-mounted display. The visual feedback depends on the rider's head position in the real world, estimated by an infrared sensor, a magnetometer, a gyroscope, and an accelerometer. This estimation allows to compute where the rider's head is in the virtual world. The computer running the vehicle dynamics model receives feedback from several input devices and outputs to the audio, vision, and motion components to give cues to the rider.

This simulator uses the Cruden Panthera software, which has the ePhyse as the heart of the simulator. ePhyse is a generic interface that allows custom software modules to communicate over a network. These custom software modules can be linked to the simulator environment, which should allow to add new subsystems to the simulator.

2.1.5 *Daimler*

Daimler [16] was another manufacture that developed a moving-base driving simulator to be used within the vehicle development process. The simulator possesses Traffic and Experiment Control software that allows to generate autonomous and deterministic traffic, place obstacles, and control pedestrians. This software has a Graphical User Interface (GUI) to control and monitor software modules. Besides, a 360º screen and the rear mirrors LCDs provide visual cues. The simulator is inside a dome mounted on a hexapod platform on a single axis for linear motion. In addition, inside the dome, a motion platform provides lateral and longitudinal movement. Regarding the communication between the vehicle's interfaces and the simulation software, the CAN extension mechanism was the only method referenced.

2.1.6 *CARLA*

CARLA [17] simulator is also mentioned to have a more in-depth vision of simulators, even though this type of simulator is not in the scope of this dissertation. CARLA is a driving simulator created to support the development, training, and validation of autonomous driving systems, both for perception and control. CARLA is designed as a server-client system. The server runs the simulation and renders the scene. On the other hand, the client sends commands to the server and receives sensor readings.

This driving simulator is open-source and provides some features to use freely. The API allows to control the interaction between the server and the client, traffic generation, pedestrian behavior, weather, and sensors during a simulation. The type and the number of sensors used in the simulation are configurable and include LIDARs, cameras depth sensors, and GPS. Besides, generating new maps using a popular tool like RoadRunner is possible. In addition, CARLA provides the integration with ROS and ROS2 via ROS-bridge. This integration allows CARLA to have an extension mechanism based on a popular interface.

2.2 MIDDLEWARE

In the more traditional architectures, applications use an Object Request Broker (ORB) middleware, allowing program calls to be made to other computers, providing transparency through Remote Procedure Calls (RPC). Applications using an ORB do not know if they are requesting a local process or a remote computer. The client application makes requests to a *stub* that forwards the request to the server. The server-side has a skeleton to handle the request, which returns a response to the stub. The stub, in turn, delivers the server's response to the client application.

However, this traditional architecture has a strong coupling between client and server, and in a simulation environment where components just want to share data with each other is not the most appropriate solution. Rather than being based on RPC, a Message-Oriented Middleware (MOM) solution is an option. In this architecture, the MOM is a middle layer for the whole distributed system that acts as a broker to send and receive messages. The MOM handles concurrency, distribution, connection establishment, and message sending from producers to consumers. In this architecture, elements are loosely coupled since replacing producers does not affect consumers. Within this architecture, a publish-subscribe paradigm is usually the most opted solution. In the publish-subscribe paradigm, publishers produce data that is categorized. Then, the middleware is in charge of delivering the data only to consumers interested in the message's category.

These MOM architectures are mainly popular in Information Technology (IT), as components can be distributed over heterogeneous environments, such as hybrid, cloud, or IoT having a multitude of different sources and consumers of data. An event-driven architecture can connect these distributed applications with an efficient, scalable, and agile method. For these environments, there are two popular solutions, namely, Solace and Apache Kafka®.

Solace [18] offers an event streaming platform to design, deploy, and manage event-driven architectures across cloud and IoT environments. This software provides high-performance global event routing for applications connected to a Solace event mesh. Solace supports three types of communication: First, in the Publish-Subscribe messaging pattern, producers publish new messages that are received by the Solace event broker. In turn, the event broker sends a copy of the message to every bound subscriber that is interested in the data, i.e., consumers that subscribed to the message's topic. The event broker will not save those messages persistently or acknowledge if the consumers received those messages. Second, it supports guaranteed messaging us-

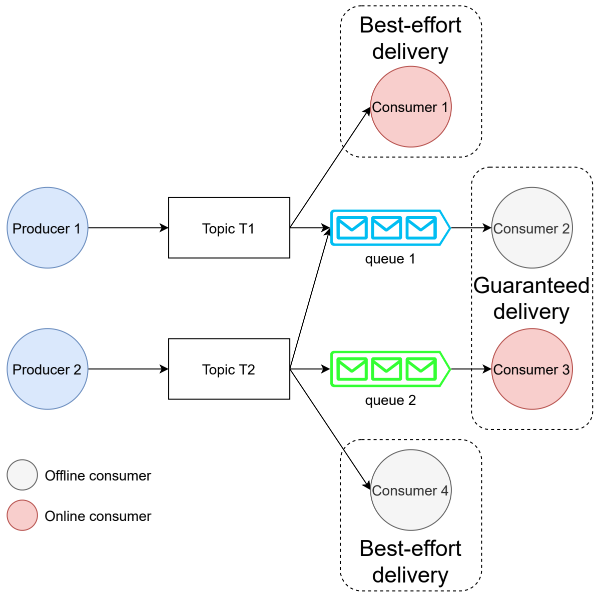

Figure 1: Publish-Subscribe with Point-to-Point communication.

ing Point-to-Point communication. Point-to-Point communication uses the concept of queue, which is a broker's object that saves messages persistently until they are consumed. In Point-to-Point communication, producer applications send messages to a specific queue, and bound clients extract messages from it. Third, Solace supports both Publish-Subscribe with Point-to-Point simultaneously, ensuring that even in case of failure, published messages are delivered to consumers. Solace has the Topic Subscription on Queues feature that allows queues to subscribe to a set of topics. This feature allows queues to receive and persist messages directly destined to them and messages that match the topics the queue is subscribed to. An example diagram of this feature is shown in Figure 1.

Apache Kafka® [19] is an event streaming platform that captures data from event sources in form of a stream of events. These events are stored durably to be manipulated, processed, and routed to different destinations. Apache Kafka® allows client applications to publish and subscribe to a stream of events, to store them durably and reliably, and to process them as they occur in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Apache Kafka® uses a Publish-Subscribe messaging pattern where producers write events or messages to a topic to be read by consumers interested in that data. The Apache Kafka® cluster contains a list of networked brokers that act together as a single unit. Producers or publishers are applications that are potential sources of events. When a publisher produces an event, the brokers store it in their topics. Subscribers or consumers are applications that poll data from the broker topics. Producers do not know anything about the consumers that will read the data and consumers do not know anything about the producer that wrote the events they consumed. A topic is a collection of related messages or events that are durably stored and can be compared to a log. When a publisher produces an event, the message is associated with a topic for the broker to store it at the topic's log without changing or deleting old messages. The Apache Kafka® cluster makes producers and consumers decoupled from each other. To increase scalability, Apache Kafka® spreads topics over partitions assigned to different brokers. Besides, each partition is replicated multiple times over different brokers to guarantee fault tolerance. An example of how Apache Kafka® could distribute partitions by several brokers is shown in Figure 2.

Some programming languages try to deal with distribution by offering several primitives. Erlang [20] is a concurrent, functional programming language and runtime environment that implements the actor-based model, a type of MOM. Erlang is designed for distributed, fault-tolerant, soft real-time, highly available systems, and applications are built of lightweight Erlang processes. The Erlang runtime system provides strict process isolation, does not share any resources, and each process maintains a local state and has its own stack, heap, and garbage collector. Erlang has a set of primitives to create processes and send messages between them to simplify concurrent programming, unlike most languages that need external libraries. The only way for processes to communicate in Erlang is through a message passing mechanism. Sending is asynchronous, does not give an error even if the destination process has already terminated. The communication between processes in different machines is transparent, i.e., they do not need to be aware of the distribution. Each process has a mailbox, i.e., a queue of messages sent by others that have not been consumed yet and are kept by arrival order. A process uses a specific Erlang primitive to retrieve messages from the mailbox. When a message is consumed and removed from the mailbox, the process resumes execution.

In a robotic context where multiple producers and consumers of data exist, a distributed approach is also necessary. Multiple robotics platforms offer these solutions, being Open Robot Control Software (OROCOS) [21] one of them. OROCOS is a software with a component-based architecture that aims to provide platform independence and stand-alone components to control distributed robotic systems. Besides, it aims to offer modularity and flexibility by building systems from smaller pieces, hardware abstraction, support to several languages, real-time support, and ready-to-use components for kinematics, dynamics, planning, sensing control, and hardware interfacing. OROCOS software code is split into modules that can be of three different types: supporting, robotics, or components. Supporting modules do not have robotics functionalities but are needed to build robotics systems, e.g., numerical libraries. Robotic modules implement specific algorithms, e.g., kinematics or motion planners. The components are built from robotic and supporting modules and are a basic unit of functionality that executes one program. In OROCOS, a component exposes an algorithm to others and offers and uses services, a collection of flow ports, operations, and properties. Flow ports allow components to communicate with each other by publishing or receiving streams of data. A component has input and output ports. Input ports receive data from other components, while output ports send data. Reading and writing from ports is real-time and thread-safe. A component executes its algorithm by receiving data in its input ports. Another way to communicate with other components is through operations. Operations allow client components to invoke target methods. Finally, properties allow to configure a component in run-time.

2.3 ROBOT OPERATING SYSTEM

The most popular robotic platform is Robot Operating System (ROS) that is used in multiple research and investigation environments, not only for robotics but also for autonomous driving [3]. It offers a set of applications,

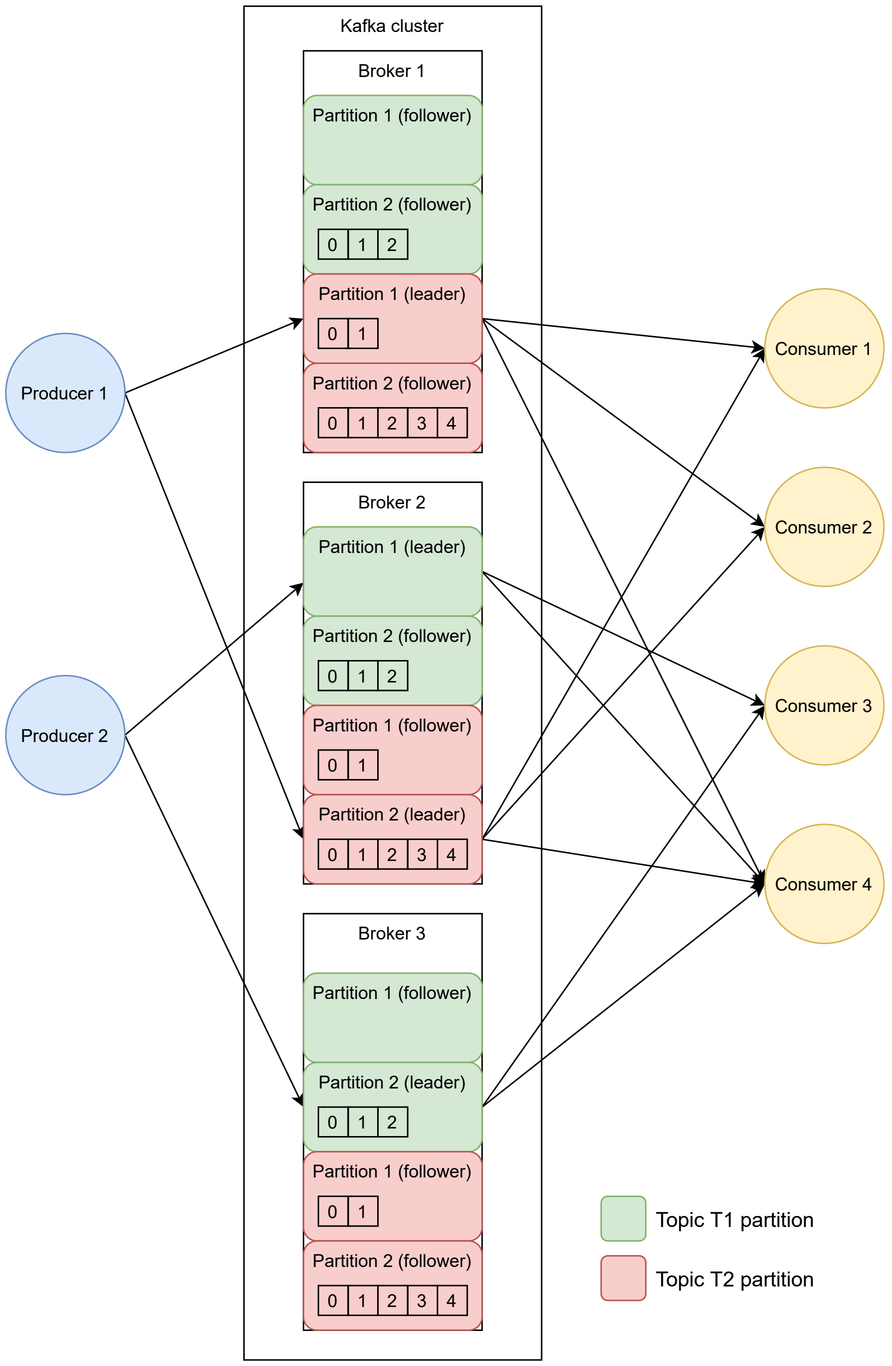

Figure 2: Kafka's cluster topics and partitions distribution example.

libraries, and tools to build robotic (and other) applications and has features for heterogeneous computing, such as hardware abstraction, message-passing between processes, and package management. It offers an entire ecosystem with multiple packages for common robotics tasks, a build system that defines the dependencies between packages, and multiple useful tools for testing, debugging, and visualizing and recording data.

The main communication mechanism used by ROS middleware is message-oriented, namely, a publish-subscribe messaging pattern. In this communication mechanism, the processing takes place in the nodes that can be of two different types: The producers, also called publishers, send messages to a specific class, or like is called in ROS a topic. Consumers, also called subscribers, receive the published data by producers if they are interested. In this protocol, publishers do not send messages directly to a specific receiver, instead, the data is categorized, and any subscriber only receives these messages if they are interested. A single node is not limited to be a publisher or a subscriber, in fact, they can publish messages and subscribe to any desired topics.

2.3.1 *Robot Operating System 2*

Robot Operating System 2 (ROS2) is the next generation of the middleware ROS that not only maintains the advantages of ROS but also has some improvements [22; 23]. Conceptually, the same publish-subscribe messaging pattern is used, but internally ROS2 was redesigned.

One of the improvements of ROS2 was the replacement of the personalized communication protocol TCPROS and UDPROS by the Data Distribution Service (DDS) standard [24]. This standard is more reliable, durable, scalable, and suitable for real-time applications. ROS uses a custom serialization format and transport protocol and a central discovery mechanism, requiring the system to have a master process, a central point of failure. On the other hand, ROS2 uses an abstract middleware interface, which provides a serialization format, transport, and a distributed discovery protocol making ROS2 more fault-tolerant and flexible.

The discovery process is how the nodes determine to talk to each other. When a node starts, all the other nodes in the same network are informed of its presence. The nodes that received the initial request respond with information about themselves to create the necessary connections. Besides, nodes periodically advertise their presence. When a node goes offline, the other nodes are advertised.

The connections are created automatically and will only happen if both nodes have compatible Quality of Service (QoS), i.e., if both nodes have compatible connection settings.

This discovery mechanism is an improvement compared to the old system used by ROS. ROS uses a master node, a central point of failure, to manage the connections between nodes. In addition, in ROS, all notification changes are handled by the master. On the other hand, in ROS2, changes are published to all nodes.

The DDS [25] is a machine-to-machine middleware that aims to address the needs of real-time applications and implements the Real-Time Publish-Subscribe protocol. It has a generic use that addresses the needs of applications like aerospace and defense, air-traffic control, autonomous vehicles, medical devices, robotics, and other applications that require real-time data exchange. ROS2 uses DDS as its communication middleware, which provides node discovery, message definition, message serialization, and a publish-subscribe messaging pattern. Applications that use DDS are decoupled, and the nodes do not have to know the other participants, their

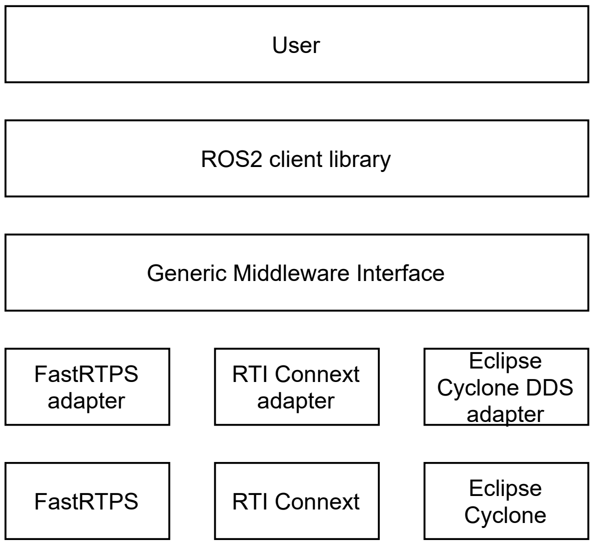

Figure 3: ROS2's middleware layers.

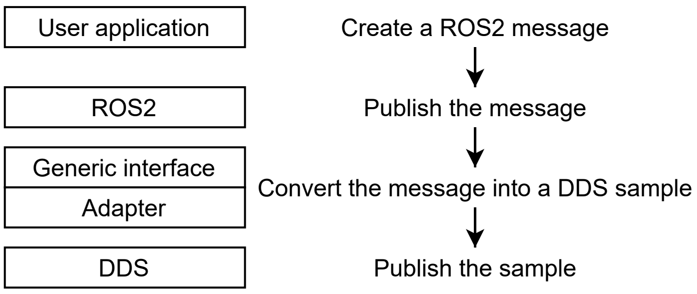

Figure 4: ROS2 message publishing flow.

location, nor if they exist. ROS2 provides an interface to underline the complexity of DDS by hiding additional layers of abstraction. ROS2 supports different DDS implementations, allowing one to choose the option that fits better.

The architecture of ROS2 can be divided into five layers, as shown in Figure 3. The first layer is the user package, which contains the user code. Second, the client libraries layer contains the ROS2 implementation layer. Third, a generic middleware interface to connect ROS2 with a generic DDS vendor. Then, the DDS adapter, a layer to connect ROS2 with a specific DDS vendor. Finally, the DDS implementation layer.

In ROS2, the message exchange starts with the user application publishing ROS2 messages using the API. Then, the ROS2 client library publishes the message to be converted by the middleware interface into a DDS sample. Finally, DDS publishes the sample into a global data space, accessible by any application. The flow to publish a ROS2 message is shown in Figure 4.

On the other way around, the subscriber flow starts in the DDS layer. DDS receives interested samples that will be converted into ROS2 messages by the middleware interface. Then the subscriber callback defined in the ROS2 client library is invoked. Finally, the user application processes the received message. The subscriber's flow to receive a new message is shown in Figure 5.

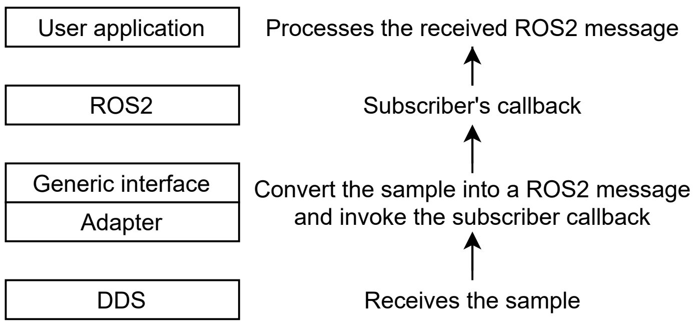
User application
Processes the received ROS2 message
ROS2
Subscriber's callback
Generic interface
Adapter
Convert the sample into a ROS2 message and invoke the subscriber callback
DDS
Receives the sample

Figure 5: ROS2 message receiving flow.

2.3.2 *Application development*

ROS2 has an API for the C++ and Python programming languages to create nodes, publish and receive messages. With this API, creating applications or external libraries to be used by other applications is possible.

ROS2 relies on the concept of workspace, a folder containing ROS2 packages. Sourcing the setup files to use ROS2's main workspace is necessary, where the default ROS2 packages, tools, and examples are accessible.

The user can create its workspaces with its own packages. Sourcing the setup files is necessary to access the packages of a user workspace, just like ROS2's main workspace. The user workspaces are overlays of the main workspace.

ROS2 workspaces can have any number of packages of two types: CMake or Python, as shown in Listing 2.1. To separate the code from the files generated by ROS2, creating packages inside the *src* folder and building them at the root of the workspace is a good practice. As a result of the compilation process, ROS2 generates the *build*, *install*, and *log* folders. The *install* folder contains the script to source the workspace.

```
workspace_folder/
    build/
    install/
    log/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

Listing 2.1: Folder structure of a ROS2 workspace

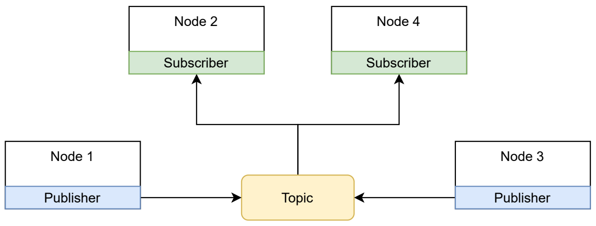

Figure 6: ROS2 publish-subscribe messaging pattern.

A package is a container of ROS2 code. Using packages is the best way to organize, build, and share ROS2 code. ROS2 allows to create packages with CMake to build C++ packages or with Python.

The package structure depends on its type. Both C++ and Python packages require a *package.xml* file that contains meta-information, e.g., about other package dependencies. In addition, CMake requires a *CMake-Lists.txt* file that describes how to build the C++ code. Regarding Python, the *setup.py* file, which contains instructions on how to install the package, and *setup.cfg* file, used by ROS2 to find the package files, are required.

A ROS2 node is a self-contained execution process, like a program that runs concurrently. Nodes are part of the ROS2 graph and are responsible for some tasks. Each node can send and receive data using the publish-subscribe messaging pattern. This communication pattern enables nodes to be developed independently and decoupled. ROS2 client libraries offer the *rclcpp* and *rclpy* packages to develop ROS2 nodes for the C++ and Python programming languages, respectively.

Any number of nodes may form a complex ROS2 system. In these systems, topics are a vital element of the publish-subscribe messaging pattern. Topics are like a bus, where a node can publish messages to be read by subscribers. A string and a data type identify a topic. The string is the topic's name, and the data type is the data to be exchanged. For a subscriber to consume data from a publisher, the name and data type of the topic must match. In Figure 6, Node1 and Node 3 publish messages to topic "topic" received by the interested subscriber nodes: Node2 and Node4.

ROS2 interfaces define the structure of the messages to send in the publish-subscribe communication protocol. These interfaces are declared in *.msg* files using a simplified description language. From these files, ROS2 generates the interface code to be used by applications. The data types of these interfaces are primitive data types, strings, lists, arrays, or even other messages. Listing 2.2 shows a simple interface with the fields x, y, and z.

```
float64 x
float64 y
float64 z
```

Listing 2.2: Interface declaration example using ROS2 simplified description language.

Every entity in ROS2 has a QoS that represents the data transport behavior. Each data transaction is configurable via several QoS options: history, depth, reliability, durability, deadline, lifespan, liveliness, and lease duration. Configuring these settings allows the communication to be more suitable for each use case. For exam-

Board	MCU	RAM	Flash
Olimex LTD STM32-E407	STM32F407ZGT6 Cortex-M4F	196 kB	1 MB
Espressif ESP32 DevKitC	ultra-low power dual-core Xtensa LX6	520 kB	4 MB
Teensy 3.2	ARM Cortex-M4 MK20DX256VLH7	64 kB	256 kB
Teensy 4.0	ARM Cortex-M7 iMXRT1062	1024 kB	2048 kB
ROBOTIS OpenCR 1.0	ARM Cortex-M7 STM32F746ZGT6	320 kB	1024 kB
STM32L4	ARM Cortex-M4 STM32L4	128 kB	1 MB

Table 1: List of micro-ROS supported microcontrollers

ple, for an actuator with low power on a lossy network, dropping some messages using the best-effort reliability may be more appropriate.

An embedded system can have a high computer heterogeneity, where some of the components are small sensors and actuators. Unlike powerful computers, some of the smaller components have hardware limitations, so it might not be possible to execute ROS2. An identical middleware to ROS2, namely micro-ROS [26], is designed for this use case. micro-ROS provides several ROS2 features, like nodes, publish-subscribe messaging pattern, and integration into the ROS2 node graph, but designed for microcontrollers. It offers an API for C/C++ programming languages, based on the ROS2 Client Library, and provides some tools and extensions. These two libraries are optimized for real-time applications and embedded systems. micro-ROS nodes integrate the ROS2 graph, allowing ROS2 tools and API to access micro-ROS nodes just like a normal ROS2 node. micro-ROS is implemented on Micro-XRCE-DDS, which allows eXtremely Resource Constrained Environments (XRCE) to communicate with an existing DDS network. micro-ROS supports several boards, listed in Table 1 and several Real-Time Operating System (RTOSes), namely, FreeRTOS, Zephyr, and NuttX.

ROS2 offers a set of tools, making debugging the entire system easier. Some tools have a GUI, and others are accessible through the Command-Line Interface (CLI). ROS2 CLI is usually used to start a node, but ros2launch allows to launch several with just one command, which may be handy for larger systems. The CLI is also used as a debugger since it introspects nodes, topics, and interfaces. In addition, it allows to publish and listen to messages of a specific topic. rqt is a GUI tool that offers several functionalities, e.g., starting nodes, introspecting the ROS2 graph, and logging. Nodes can use log messages to report some work or state. These log messages help to understand the system's state and sometimes to use as a debugger. Finally, ros2bag saves data published to a specific topic into a database by saving the exchanged messages into a file that can be replayed later.

ROS2 offers a special type of nodes with a managed lifecycle that has a state machine and exposes a known interface. This interface allows the node to execute according to its state and transit between states allowing external applications to manage lifecycle nodes. These external applications should provide the initial configuration, when to transit between primary states, and monitor in case of failures. Lifecycle nodes have a state machine with four primary states and six transition states. In a primary state, nodes can execute some tasks, depending on which state the node is in. In transition states, nodes will execute some procedure to change from one primary state to another. The lifecycle nodes state machine is shown in Figure 7.

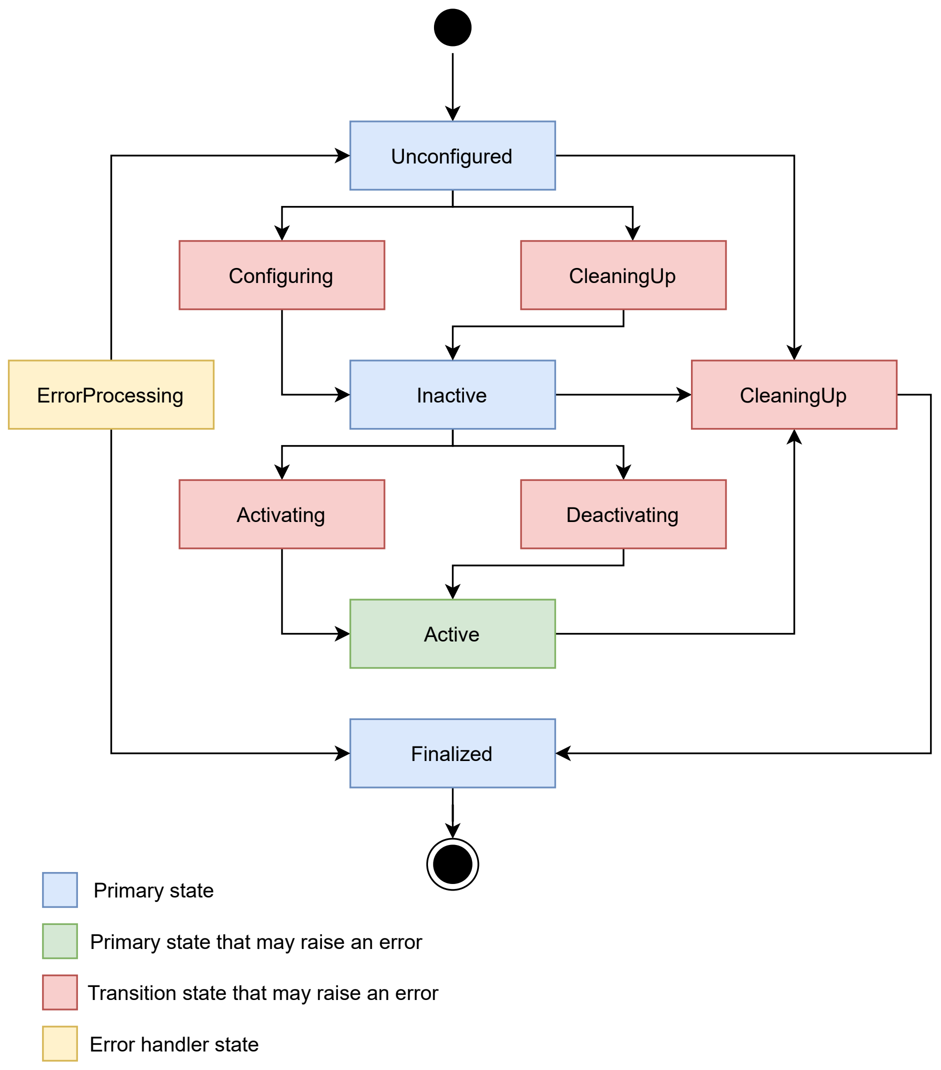

Figure 7: State machine of a lifecycle node.

A node in a primary state can be Unconfigured, Inactive, Active, or Finalized. An Unconfigured node transits to this state right after being instantiated. The Inactive state allows the node to be configured, e.g., to change parameters. The Active is the node's main state, which does the primary processing, from publishing messages to processing data. Finally, the Finalized state is immediately before the node's destruction.

3

SILAB SIMULATOR SYSTEM

This chapter presents the main characteristics of SILAB, its native extension mechanism, and how to develop software modules. After that, the architecture of the motorcycle simulator used in the dissertation is described.

3.1 SILAB OVERVIEW

For this dissertation, in particular, SILAB, a driving simulator software that has been continuously developed by **WIVW** [27] is the best option even because the University of Minho owns a SILAB simulator. An example of an image produced by SILAB is shown in Figure 8.

SILAB has complex physic models to mimic real-world vehicles (cars or motorcycles), has multiple models to control the behavior of other road users, allows to record data during the simulation, and integrate external hardware and software modules to expose the simulator functionalities. SILAB is versatile since it can run in a cluster, or a workstation, have multiple screens, the vehicle can be controlled by a steering wheel, keyboard, or joystick, and has several tools to become more user-friendly.

First, the SILAB's main software is the one used to start simulations. This software has a GUI to inspect information about the running simulation. Such as the computers and the software modules. Other computers that may participate have to run the SILABRemote to be reachable. Since SILAB can run in a cluster, SILABAdmin is used to inspect the entire network of computers and their information. SILABAdmin allows the main computer to check the remote computers' status and information, e.g., hardware and IP addresses. Besides, controlling the entire cluster in this program, e.g., shutting down, or rebooting is possible. While the main computer has SILABAdmin, all the others must run SILABButler to be reachable. To develop new scenarios, SILABAEdit allows to create complex parts, using a rich graphical tool to place the roads, other vehicles, pedestrians, environmental objects, and traffic signs. Besides, this tool allows to define the other vehicles' and pedestrians' behavior.

3.2 SILAB CONFIGURATION FILE

The simulations run in SILAB can have an infinite amount of different configurations. For example, some simulations require SILAB to run a specific scenario, while others require a different one. Instead of programming SILAB

Figure 8: Example of an image produced by SILAB.

to change specific settings, each simulation plan is described in a configuration file. Having these configuration files allows SILAB to be highly configurable.

When users start SILAB, it always asks about a configuration file to describe the simulation plan. From the configuration files, SILAB allows to control a multitude of settings to run the next simulation. These configuration files define, among other things, which computers participate in the simulation, which software modules operate on which computers, visualization settings, in which routes the simulation takes place, and how the other road users and pedestrians should behave. Working with configuration files allows SILAB only to load the specified software modules. Unnecessary software modules are not loaded and do not affect the simulation. Besides, extending a simulation is easy since it is only necessary to instantiate the new software modules in the configuration file. In addition, the configuration files allow SILAB to perform different tasks, from data and video recording to testing advanced driver-assistance systems.

Including all configurations in a single file can generate considerably big files. SILAB has an *include* feature to ease the separation of parts of the configuration and reuse static parts. These static parts remain identical between different simulations, e.g., the graphic settings. So, instead of creating configuration files from scratch, including already defined parts is possible. Besides, SILAB offers a default configuration design that includes the basic settings to run a simulation.

Configuration files are pure text files that follow a specific grammar. The usual structure of these configuration files is shown in Listing 3.1. The *System* section defines parameters that influence the SILAB's behavior on a global level, e.g., the monitoring. The *Configuration* section contains two more subsections that are the core of the configuration file. The *Computerconfiguration* subsection contains a list of computer pools, which define the computers that participate in the simulation. The *DPUConfiguration* subsection contains a list of software modules that participate in the simulation. Both *Computerconfiguration* and *DPUConfiguration* subsections are described in more detail up next. The *TRF* section contains general settings about other traffic participants. Moreover, the *SCN* section contains information about the scenario, namely, the routes, landscapes, and graphical objects. Finally, the *MOP* section contains general information about the pedestrians. An example of a configuration file is shown in Appendix A. A basic configuration file can have multiple lines, so, in this example, some of the details are included in other files.

```
SILAB System
{
    # System configuration
};

SILAB Configuration
{
    Computerconfiguration MyComputerConfiguration
    {
        # Computer configuration
    };

    DPUConfiguration MyDPUConfiguration
    {
        # DPU configuration
    };
};

SILAB TRF
{
    # General information about traffic simulation
};

SILAB SCN
{
    # Scenario definition
};

SILAB MOP
{
    # General information about the pedestrians
};
```

Listing 3.1: SILAB configuration file structure.

3.2.1 *Hardware resources*

One of the main subsections of the configuration file is the *Computerconfiguration* that defines the hardware resources chosen to participate in the next simulation and their detailed information. This information includes the computer's name, IP address, frequency, and aliases. The *Computerconfiguration* subsection is also known as Computer Pool. The configuration file must contain at least one Computer Pool, but when the manager starts a SILAB simulation, the software requests to choose the desired Computer Pool. Each Computer Pool must define the Operator node. The Operator is the one that starts the simulation and watches over all the other remote computers by contacting them periodically.

An example of two Computer Pools is shown in Listing 3.2. The first Computer Pool, called *MySimulator*, instantiates three computers. The *MySimulator* Computer Pool distributes the computation over several computers and can run in a cluster. The second Computer Pool, called *MyWorkstation* has a single computer. The *MyWorkstation* Computer Pool concentrates the computation in a single computer and usually is set to run in a workstation.

The *MySimulator* Computer Pool instantiated the *Operator*, *Traffic*, and *Visualization* computers. These computers are set with a frequency of 60, 120, and 60 Hz, respectively. The computer's frequency indicates how many times per second the software modules assigned to them must execute. For example, the *Traffic* computer runs its software modules' main algorithm 120 times per second. Besides, the *Traffic* and *Visualization* computers have an alias. Tasks can be assigned to computers using the computer's name or one of its aliases. Aliases allow to aggregate a set of tasks. In this way, if a computer is overloaded, a set of tasks can be assigned to another one by changing the computers' alias parameter. For example, if the *Traffic* computer is overloaded, all *SOUND* related tasks can be easily assigned to *Visualization* by changing the *SOUND* alias from the *Traffic* to the *Visualization* computer.

Regarding the *MyWorkstation* Computer Pool, only one computer is instantiated. To this computer, multiple aliases are assigned. In this way, using this Computer Pool automatically assigns the *Operator*, *Traffic* and *Visualization* tasks to the *MyWorkstation* computer.

```
Computerconfiguration MyComputerPool
{
    Pool MySimulator
    {
        Executable = true;

        Computer Operator
        {
            IP = "10.1.1.1";
            Frequency = 60;
            Operator = true;
        };

        Computer Traffic
        {
            IP = "10.1.1.2";
            Frequency = 120;
            Alias = {TRF, SOUND};
        };

        Computer Visualization
        {
            IP = "10.1.1.3";
            Frequency = 60;
            Alias = {VIS};
        };
    };

    Pool MyWorkstation
    {
        Executable = true;

        Computer MyWorkstation
        {
            IP = "127.0.0.1";
            Frequency = 40;
            Operator = true;
            Alias = {Operator, Traffic, TRF, SOUND, Visualization, VIS};
        }
    }
};
```

Listing 3.2: Example of two Computer Pools.

3.2.2 *Software resources*

While the *computerconfiguration* subsection defines hardware resources, the configuration files defines the software resources in the *DPUConfiguration* subsection. The *DPUConfiguration* subsection is also known as Data Processing Unit (DPU) Pool. Like the Computer Pool, the configuration file must contain at least one DPU Pool, but when the manager starts a SILAB simulation, the software requests to choose the desired DPU Pool. A DPU Pool lists multiple software modules that are set to participate in the next simulation. These software modules are called DPUs and run a specific task in a simulation. DPUs can be responsible for a driving task, e.g., vehicle dynamics, or for a generic function, e.g., compare two values. The DPU itself can be seen as the class, and the DPU instance the object. Each DPU instance defines its type, name, which simulation computer runs this instance, and assign the desired values for the class parameters. Every DPU has a set of inputs and outputs. DPUs compute their outputs according to the input data obtained from other DPUs. The DPU outputs are in turn available to other DPUs.

DPU Pools instantiate DPUs and how they are connected, as shown in Listing 3.3 example. The *MyPool* DPU Pool instantiates two DPUs. The first one is an instance of the *DPUController* class and is called *Controller*. The second one is an instance of the *DPUVDyn* class and is called *Vdyn*.

Each DPU instance has to specify its *Computer* parameter. This parameter specifies which computer processes this instance. The *Controller* DPU runs on the *Operator* computer, while the *Vdyn* runs on the *Traffic* computer. Besides, within every simulation step, some DPUs have to run in a specific sequence. In SILAB, this order is defined assigning the *Index* parameter. The DPUs with lower *Index* values are computed first. In the *MyPool*, the *Controller* DPU computes before the *Vdyn* DPU. In this way, the *Vdyn* inputs depend on the outputs of the *Controller* DPU. Finally, each DPU assigns a value to the class parameters. The *Controller* assigns the value 0 to the variable *Flag* and the string 123 to the variable *Value*. Also, DPU Pools have to connect DPUs. In this example, the *Controller* outputs are linked to the *Vdyn* inputs.

```
Pool MyPool
{
    Executable = true;

    DPUController Controller
    {
        Computer = {Operator};
        Index = 10;

        Flag = 0;
        Value = "123";
    };

    DPUVDyn Vdyn
    {
        Computer = {Trafic};
        Index = 20;
    };

    Connections =
    {
        Controller.SteeringWheel -> Vdyn.SteeringWheel,
        Controller.AcceleratorPedal -> Vdyn.AcceleratorPedal,
        Controller.BrakePedal -> Vdyn.BrakePedal
    };
};
```

Listing 3.3: DPU Pool example.

Even though SILAB has a base configuration with the necessary DPUs to run a simulation, some particular cases may need specific DPUs. Instead of creating a new DPU Pool from scratch, SILAB has an inheritance mechanism that allows to extend the base configuration. From the base configuration, changing or adding new information is possible. For example, instantiating more DPUs, edit a parameter of the base DPU Pool, or add more connections. Like the Computer Pools, SILAB asks the user to select the appropriate DPU to execute in the next simulation.

A DPU Pool that inherits the configuration from the Listing 3.3 DPU Pool is shown in Listing 3.4. The *MyPool2* inherits the *MyPool* configurations. Besides, the *ABC* DPU is instantiated from the *DPUABC* class. This DPU runs on the *VIS* computer after the *Controller* and *Vdyn* DPUs (*Index* = 25). Also, the *Flag* parameter of the DPU *Controller* is edited. In the end, the connections between *Vdyn* and *ABC* are defined.

```
Pool MyPool2: MyPool
{
    Executable = true;

    DPUABC ABC
    {
        Computer = {VIS};
        Index = 25;
    };

    Controller.Flag = 1;

    Connections =
    {
        Vdyn.X -> ABC.X,
        Vdyn.Z -> ABC.Z
        Vdyn.Y -> ABC.Y
    };
};
```

Listing 3.4: DPU Pool inheritance example.

3.2.3 *Simulation scenarios*

The last important subsection of a configuration file is the simulation scenario defined in the *SCN* subsection. In SILAB, simulation scenarios are called databases or maps and are composed of road segments, traffic definition, pedestrian definition, flowpoints, and scenario modules.

The smallest unit in a simulation scenario is the road segments, and they are of two types: Courses and Area2s. Courses have constant features along the road, e.g., the road width and the number of lanes. Area2s are the other types of roads, e.g., intersections, freeway entry ramps, and freeway exits.

The definition of Courses and Area2s is done using different methods. Courses road segments are defined in the configuration file, where the road layout, the height profile, the appearance of the landscapes, and cross-sections are specified. For Area2s, SILAB provides a program to edit this type of road segment since it can be very complex to design one in a configuration file. This program is called SILABAedit and produces more complex configuration files offering a GUI.

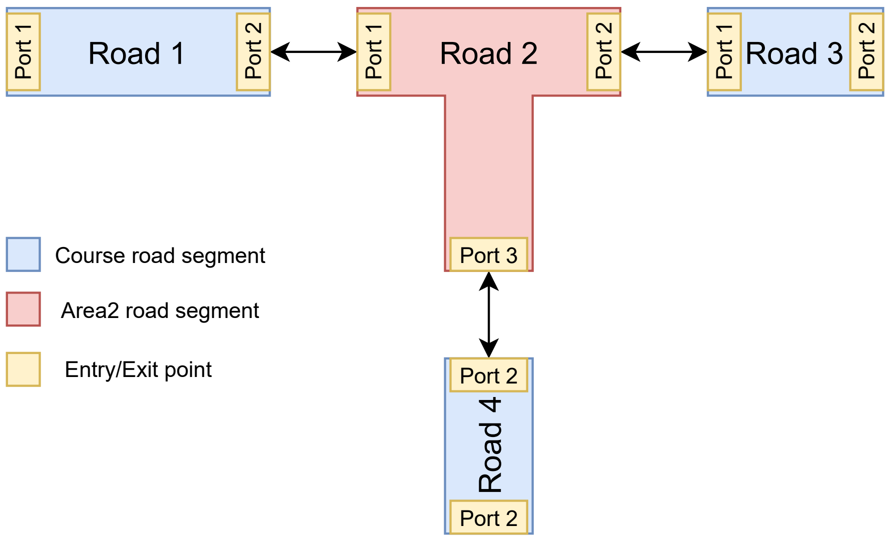

Figure 9: Road network example.

The configuration file must link all these road segments. Courses only have two entry points, while an Area2 road segment can have multiple. A connection example between different road segments is shown in Figure 9. One Area2 and several Courses are linked to form a road network.

In the configuration file, specifying the behavior of the traffic and pedestrians during the simulation is also necessary. SILAB provides several features to control other road users, like choosing the type of vehicle and its behavior. In SILAB, behavior schemes can be used to control the behavior of the users, i.e., a set of behavior models that are hierarchically combined. If more than one behavior model sends some control signal to the road user, only the one with a higher hierarchy is activated. Besides, to control other road users in specific situations, SILAB uses flowpoints. Flowpoints are invisible checkpoints on the road that triggers some event to control a road user, e.g., spawning a vehicle or changing the target velocity and acceleration. These flowpoints are activated when the user passes through the checkpoints.

A scenario module encapsulates road segments, flowpoints, and traffic and pedestrian definitions.

3.3 SILAB SOFTWARE MODULES DEVELOPMENT

SILAB operation is based on a component-based model where DPU are the modules responsible for a specific task. The combination of multiple DPUs allow to create a more complex system and run a complete simulation.

SILAB allows to create new DPUs that are dependent of the SILAB state machine. As shown in Figure 10, SILAB has five different states. When SILAB tries to transit between states, the methods of every simulation DPU associated with each transition are called.

SILAB starts in the *Off* state. When the manager selects the simulation configuration file that contains the plan for the next simulation, SILAB tries to transit to the *Initialized* state. First, the Operator contacts the other simulation computers and sends the simulation plan. Then, the computers load and instantiate the DPUs they set

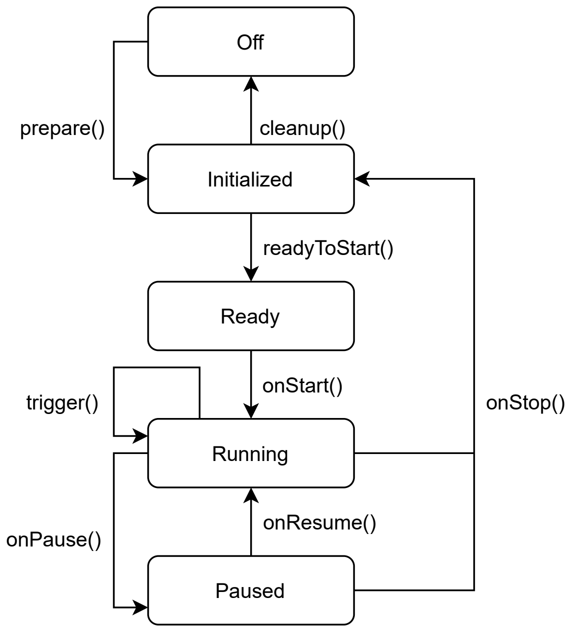

Figure 10: SILAB state machine.

to run by calling the DPU constructor. After that, each computer establishes the necessary network connections. Finally, the method *prepare* of every DPU is called. Only if these steps are successful, SILAB transits to the *Initialized* state.

When the manager requests SILAB to transit to the *Ready* state, the *readyToStart* method of every DPU is called. After all DPUs finish computing the method with success, SILAB can transit to the *Ready* state. Otherwise, if any DPU does not succeed, SILAB stays in the *Initialized* state and requires all DPUs to execute the method *readyToStart* again.

In the *Ready* state, the simulation is ready to start. When the manager requests the simulation to start, SILAB executes the *onStart* method of all DPUs. No matter the result of the *onStart* method, SILAB transits to the *Running* state.

During the *Running* state, each computer running the simulation executes the *trigger* method of the DPUs they are running. Each time all the DPUs of a computer complete this method, SILAB completes a simulation step. The number of times this method is executed depends on the defined computer's frequency that is running the DPU. Before each call, SILAB updates the input variables according to the DPU connections. In the *Running* state, the manager can request SILAB to pause or stop the simulation. If the manager requests a pause, all DPUs execute the *onPause* method and SILAB goes to the *Paused* state. In the *Pause* state, the manager can request SILAB to stop and go to the *Initialized* state (DPUs execute the *OnStop* method) or to resume the simulation and go back to the *Running* state (DPUs execute the *OnResume* method). Finally, when the manager closes the SILAB application, the DPUs execute the *cleanup* method and, then, the application is closed.

SILAB allows to write new DPUs using the C++, Java, and Ruby programming languages. The steps to develop a DPU in each programming language are identical. DPUs are defined as classes of each programming language. The class instance variables define the DPU's inputs, parameters, and outputs. The class methods are the DPU's callbacks that define the task to do between state transitions. These methods are shown in Figure 10. The *prepare*, *Trigger*, and *CleanUp* are the most commonly used methods. The *prepare* method defines the DPU's task between the Off and Initialized states. The *Trigger* method defines the DPU's main algorithm that runs in every simulation step. Finally, the *CleanUp* method releases the allocated resources. For example, a DPU set to compute the distance to the setup implements the distance calculation algorithm in the *Trigger* method that is executed in every simulation step.

In the case of a C++ DPU, first, a specific SILAB software has to be used to generate the DPU project that contains two files. Then, one of the files contains a struct that is used to define the DPU inputs, parameters, and outputs. After that, a different file contains a class that extends the DPU base, class. The extended class has the struct as an instance variable, a class constructor, and some of the methods illustrated in Figure 10. The only methods implemented are the ones that the DPU has some task to do. The undeclared methods have the base class' implementation. Moreover, the files must be compiled as a dynamic library that must be copied to a specific SILAB folder. Finally, to be used in a simulation, the created DPU class needs to be instantiated at least one time. This way, SILAB searches in a specific folder for the generated dynamic library file and loads it to be used in the simulation.

Regarding the implementation of DPUs in the Ruby and Java languages, the process is equivalent to the C++ language but has some differences. Both Ruby and Java are implemented in a single file and do not need to be compiled. Instead, SILAB offers the *DPURuby* and *DPUJJava* to load the Ruby script and the Java bytecode, respectively.

An example of a snippet of a configuration file that uses a C++, Ruby, and Java DPU is shown in Listing 3.5. The *CppOriginDistance* DPU, *RubyOriginDistance*, and *JavaOriginDistance* are written in C++, Ruby, and Java, respectively.

```
DPUOriginDistance CppOriginDistance
{
  Index = 1002;
  Computer = {Operator};
};

DPURuby RubyOriginDistance
{
  Index = 1002;
  Computer = {Operator};
  Script = "RPURubyOriginDistance.rb";
};

DPUJJava JavaOriginDistance
{
    Index = 10;
    Computer = {Operator};
    Class = JavaOriginDistance;
};
```

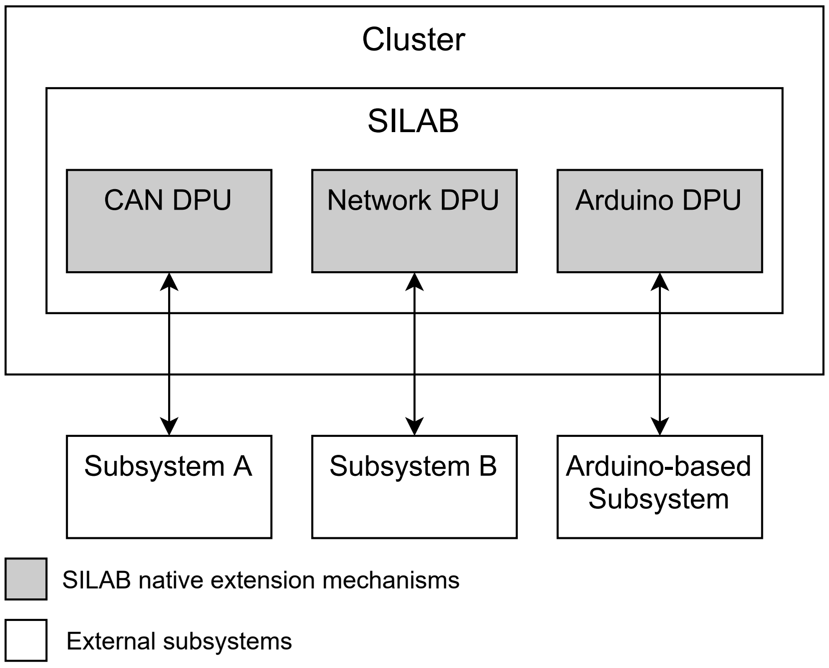

Figure 11: SILAB native extension mechanisms.

```
Connections =
{
  VDyn.X   -> RubyOriginDistance.position_x,
  VDyn.Y   -> RubyOriginDistance.position_y,
  VDyn.Z   -> RubyOriginDistance.position_z,

  VDyn.X   -> JavaOriginDistance.position_x,
  VDyn.Y   -> JavaOriginDistance.position_y,
  VDyn.Z   -> JavaOriginDistance.position_z
};
```

Listing 3.5: Example on how to add a DPU to the configuration file in three different programming languages.

3.4 COMMUNICATION WITH EXTERNAL ENTITIES

SILAB natively offers some extension mechanisms to allow external subsystems, e.g., another application running in a remote computer or a piece of hardware in the simulator's Mock-up, to communicate with the simulation environment. Natively SILAB supports three different extension mechanisms: CAN, network sockets, and Arduino boards, as shown in Figure 11. These extension mechanisms are available using the SILAB script language in the configuration file.

The CAN interface is usually used in cars and motorcycles. So, since multiple Mock-up components are from real vehicles, SILAB natively offers the CAN extension mechanism. More specifically, SILAB offers a DPU to control and manage the CAN bus, called *DPUBUSCAN*. This DPU represents a driver for the CAN bus channel. Besides, the *DPUMSGCAN* is used to send and receive messages periodically. External hardware physically connected to a computer running the *DPUBUSCAN* can use the CAN bus channel to send and receive messages.

As shown in Listing 3.6, configuration files can instantiate the CAN DPUs to use this extension mechanism. The *pCAN* allows the *readCAN* to read messages from the CAN bus. Then, the *readCAN* accesses specific parts of the received messages and outputs those values to other DPUs. In this example, the *DPUMSGCAN* reads messages from the CAN bus. To write messages to the CAN bus, the parameter *Write* needs to be 1.

```
DPUBUSCAN pCAN
{
    Computer = {Operator};
    Index = 10;
    CANMode = 41;
    Baud = 1;
};

DPUMSGCAN readCAN
{
    Computer = {Operator};
    Index = 10;
    CANMode = 41;
    ID = 0x14;
    LEN = 1;
    Write = 0;

    ValueOffset1 = 0;
    DataBegin1 = 8;
    DataLength1 = 4;

    ValueOffset2 = 0;
    DataBegin2 = 12;
    DataLength2 = 4;
};

Connections =
{
    readCAN.Value1  -> SomeDPU.Input1,
    readCAN.Value2  -> SomeDPU.Input2
};
```

Listing 3.6: CAN extension mechanism example.

In addition, SILAB has the DPUNetwork to establish TCP/IP or UDP connections to other computers. This DPU can act as a server or as a client and exchanges packets of binary data. As a server, whenever the DPU receives messages from a client, the DPUNetwork updates the simulation state. As a client, this DPU sends one data packet to the remote computer in every simulation step.

An example of a *DPUNetwork* is shown in Listing 3.7. The *SumNetwork* DPU acts as a client and sends binary packets with two doubles. An application listening in the 8888 port at 192.168.2.2 address is set to receive those binary packets. In return, the application responds with a binary packet containing one double. The received double is an output of the *SumNetwork* DPU.

```
DPUNetwork SumNetwork
{
  Index = 20;
  Computer = {TRF};
  Network_IP = "192.168.2.2";
  Network_IsTCPIP = 0;
  Network_PortSend = 8888;
  Network_PortReceive = 8889;
  Network_IsServer = false;
  SendDefinition = (
    (Sum1, double),
    (Sum2, double)
  );
  ReceiveDefinition = (
    (SumTotal, double)
  );
};

Connections =
{
    Vdyn.X -> SumNetwork.Sum1,
    Vdyn.Y -> SumNetwork.Sum2,
    SumNetwork.SumTotal -> SomeDPU.SumTotal
};
```

Listing 3.7: Network socket extension mechanism example.

Finally, in embedded systems, Arduino boards are commonly used, so SILAB offers the DPUArduinoIO to communicate with this type of board. The DPUArduinoIO communicates with an Arduino-based board over the network to control its analog and digital terminals. First, the IO sketch must be loaded onto an Arduino board. Then, the Arduino board must be configured to be reachable by the SILAB local network. Finally, instantiating the *DPUArduinoIO* is necessary, as shown in Listing 3.8. In this example, the *IO* DPU reads the analog pin number 0. The value read is an output of *IO*. In addition, *IO* controls two digital pins. The first pin is controlled using the analog output mode. The second pin is controlled using the digital output mode. The voltage set for each pin depends on the input data of the *IO* DPU.

```
DPUArduinoIO IO
{
    Computer = {LOCALHOST};
    Index = 1;
    Sys_IP = "10.1.1.68";
    Sys_Port = 8888;

    D3 = (PWMLight, Write, Analog, 0.05);
    D5 = (Light, Write, Digital, 1);

    A0 = (AnalogInput, 0);

    Connections =
    {
        SomeDPU.PWMLight -> IO.PWMLight,
        SomeDPU.Light -> IO.Light,

        IO.AnalogInput -> OtherDPU.AnalogInput
    };
};
```

Listing 3.8: Arduino extension mechanism example.

3.5 DRIVING SIMULATOR MOCKUP 2-WHEELER

Mock-up is the hardware that allows to physically simulate a motorcycle that interacts with the simulation projected onto a screen. The Mock-up and screen interactions are controlled by SILAB simulator. This section shows the overall architecture of the Driving Simulator Mockup 2-Wheeler (DSM-2W) and how all of its various components work together. From a general point of view, as illustrated in Figure 12, the simulator can be divided into back-office and Mock-up areas. The back-office has a cluster of computers and the experimenter computer. The Mock-up area has several interfaces for the rider to interact with the simulation environment.

The cluster executes the simulation and sends the output to the Video Projectors, which project the image onto the screen. The rider thus can view the simulation. Similarly, the cluster sends the simulation's audio to the Sound System, giving the rider an immersive audio experience. The cluster receives control data from the Mock-up, namely, the state from built-in buttons, the pressure from levers and pedals, and the angle from the handlebar's steering. In addition, it sends video and control data to the Mock-up: video to the left and right mirrors, and control, to the handlebar.

3.5.1 *Back-office*

The back-office has a cluster of ten computers running the SILAB software. Each node of the cluster has a specific task. The Master or Operator node runs the SILAB's main software. This node sends the simulation settings to the others and configures the connections within the workstation cluster. The Sound node, also known as Human Machine Interface (HMI), produces the simulation sound, and it is connected to sound amplifiers in

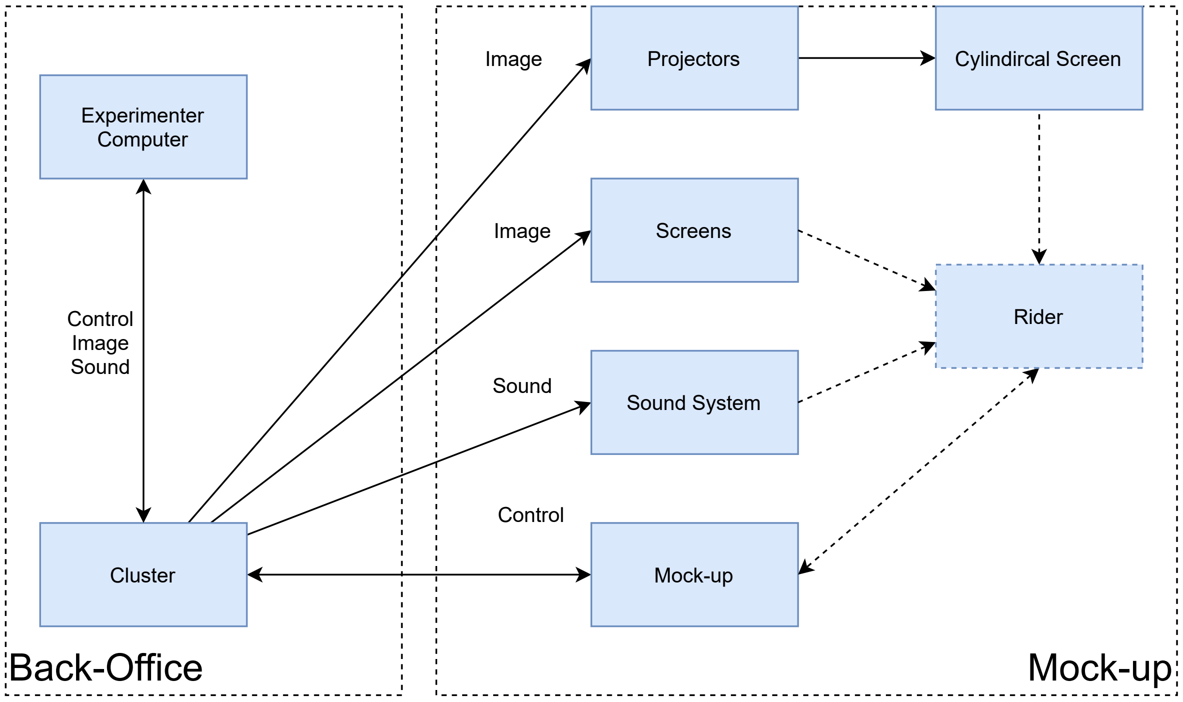

Figure 12: DSM-2W Overall architecture.

the Mock-up area. The VDyn computer is connected to an Electronic Control Unit (ECU) to send and receive control data to and from the Mock-up and computes the vehicle dynamics. The CAN computer records the video captured by several cameras strategically placed in the Mock-up area and allows communication between the experimenter and the rider. Finally, six visualization nodes generate the images for the rider to see the simulation environment. Three of these nodes produce an image to project onto the cylindrical screen. The other two produce an image to be displayed in the left and right rear mirrors. The last visualization node displays an image in the instrument cluster.

Besides, the experimenter's computer is in the back-office. This computer addresses the communication between the rider and the experimenter and remotely controls all software of the back-office computers. This computer is equipped with a dedicated microphone and headphones to improve the communication with the rider.

3.5.2 *Mock-up area*

Regarding the Mock-up area, it has several components for the rider to interact with the simulation environment, as shown in Figure 13. It has the physical structure of a real motorcycle, called Mock-up, and the screen and sound components.

The rider gets visual information from four different screens. The first one is a 180º cylindrical screen to project the simulation environment. The cluster also sends visual information to three more screens in the Mock-up. On the other way around, the Mock-up sends visual information from several cameras to monitor the rider during experiments.

Regarding the sound system, it has several speakers strategically positioned around the rider to reproduce the simulation audio. Besides, the rider receives audio from a helmet equipped with headphones capable of reproducing stereo audio and a microphone to communicate with the experimenter.

The Mock-up is based on a real motorcycle and has several input components: a handlebar, several built-in buttons, rear brake and clutch levers, a gear shifting mechanism, and a front brake pedal. These input components allow the rider to control the motorcycle in the simulation environment and connect to the cluster (and SILAB) through a dedicated ECU to send and receive control messages. The Mock-up sends messages regarding the handlebar's angle, the state of the built-in buttons, and the pressure in the pedals and levers. On the other way around, the cluster sends messages with the force-feedback to the handlebar.

The instrument cluster display is used as HMI enabling the rider to receive information from the simulated vehicle and its status, e.g., velocity and engine rotations per minute, and to provide user input through the touch screen. Moreover, the Mock-up has a vibrator motor to simulate the engine vibration. The motor is placed below the rider's seat and is connected to a power amplifier, which is connected to the cluster via an audio cable. The cluster controls the frequency and amplitude of the vibrations through an audio signal. Lastly, mounting other ECUs on top of the base Mock-up is possible. These can include other instrument clusters, head units, head-up displays, other displays with and without a touch screen, physiological data measurement devices, eye-tracking devices, and so on.

Even though the Mock-up has several mobile joints allowing the rider to use the body weight to roll the motorcycle, the vehicle in the simulation is not affected by the rider's roll movement. Since the only way to control the direction of the motorcycle is through the steering wheel, counter-steering is not available. In addition, the DSM-2W lacks a Stewart motion-based platform and other actuators to provide the rider with roll, yaw, and pitch movement. Unlike other motorcycle simulators referenced, the DSM-2W lacks some functionalities. However, using the DSM-2W for research purposes is still possible.

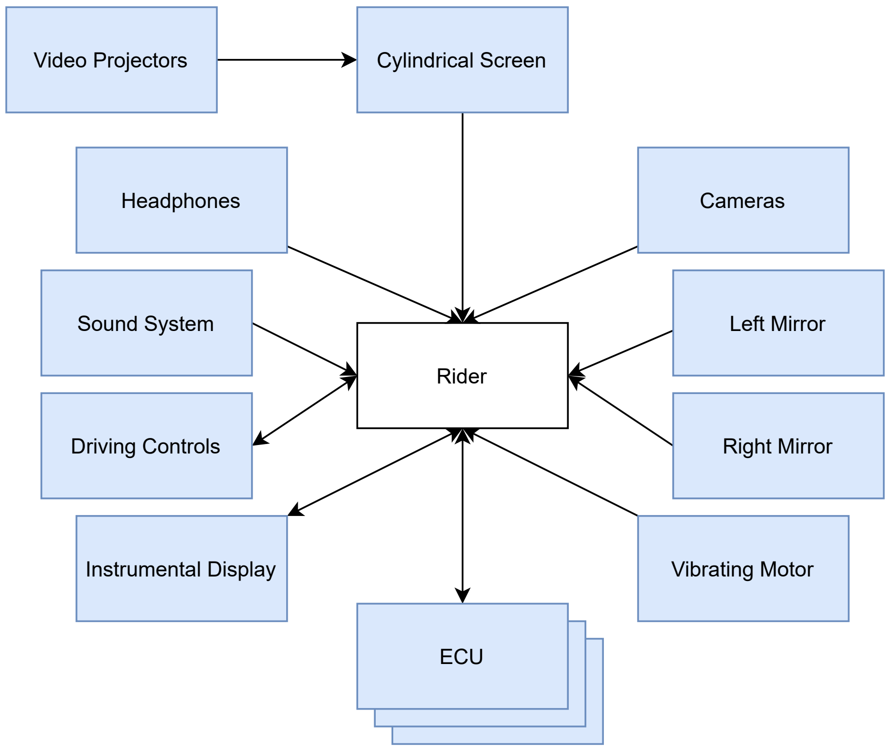

Figure 13: DSM-2W Mock-up main components

4

SILAB ROS2 INTEGRATION

This chapter presents the analysis, the solution, and the design of the integration of ROS2 in SILAB. Also, the main advantages, the potential challenges, and the architecture of a ROS2-based solution to integrate new subsystems onto the simulator is described. Moreover, the structure of a ROS2 DPU, the strategy to access ROS2 from Ruby scripts by taking advantage of several programming language's functionalities, the timeline to compile all code, and a tool to generate the entire code automatically is described.

4.1 INTEGRATION ANALYSIS

This dissertation's main goal is to make SILAB more expandable by using a well-known interface. This interface has two main requirements: Adding more subsystems onto the simulation should be seamless from the point of view of the programmer, i.e., it should be simple to integrate them and at the same time without adding any effort to the infrastructure. Besides, it should be capable of dealing with the hardware and software heterogeneity since the integrated subsystems can be a powerful computer or a microcontroller.

Although the integration of middleware onto the simulation presents some challenges (in this case, ROS2's middleware), this emerged from the fact that using a well-known middleware brings several advantages compared to a solution developed from scratch or based on one of SILAB native extension mechanisms. Both solutions do not have standardized communication, which is harder to deploy, debug, maintain, add new components, and can be hiding further issues for not having been sufficiently tested.

From all solutions analyzed, ROS2, in particular, fulfills all requirements to be the interface of SILAB since it supports several platforms, can deal with the hardware heterogeneity, uses a peer-to-peer architecture, which should guarantee a scalable system. ROS2 nodes are decoupled, making the integration of new subsystems onto the simulation environment seamless. Besides, the use of ROS2 in the automotive context is not new since the Autoware.auto project [3], designed for autonomous driving, uses this interface. The same happens for an automotive simulation context since the CARLA simulator provides a ROS-bridge.

Integrating ROS2's middleware is possible using the SILAB custom DPU ability. These DPUs access ROS2 API allowing subsystems to interact with SILAB, as shown in Figure 14. However, this integration brings several challenges: First, even though SILAB supports several programming languages, due to license restrictions, only the Ruby API is available to write custom DPUs. So bridging Ruby DPUs with the C++ or Python ROS2 libraries is necessary. Second, combining SILAB's API to retrieve the simulation data with ROS2 API to publish messages is

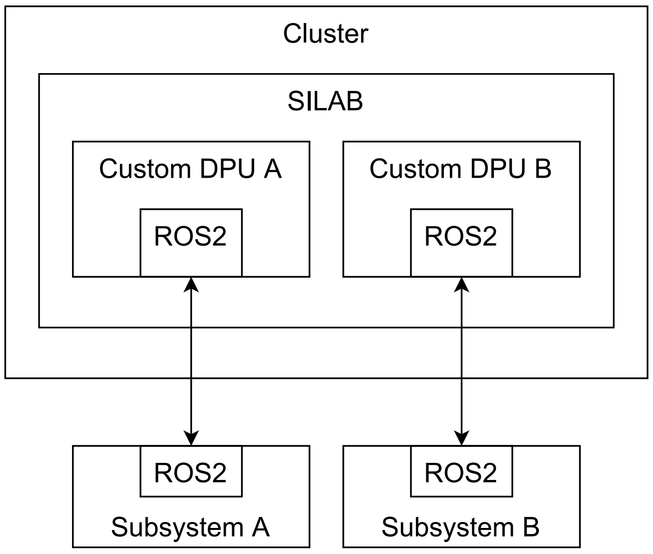

Figure 14: ROS2 extension mechanism.

needed. Also, some DPUs have to retrieve data from ROS2 topics to manipulate the simulation data. Lastly, the DSM-2W has components of several types in which some have low power, memory, and processing capacity. So, the use of ROS2 in some of these components might not be the best solution because ROS2 introduces some overhead and, these components can become overloaded, even when using more efficient QoS configurations. In these cases, replacing ROS2 with a middleware especially developed for embedded systems, like micro-ROS, might be necessary.

Even if a simulator does not support custom software modules, integrating the simulator with ROS2's environment is possible. For that, creating one or multiple ROS2 bridge nodes is necessary. These ROS2 bridges have to deliver ROS2 published messages from the subsystems to the simulation environment through a supported extension mechanism. In addition, they also have to receive data from the simulation environment to publish ROS2 messages. This workaround allows the integration of the simulation with the ROS2 environment but increases the system's complexity. This integration is shown in Figure 15.

4.2 INTEGRATION DESIGN

As can be seen from the previous sections, ROS2 fulfills the requirements and, considering its popularity, is naturally the chosen solution. In this solution, integrating both SILAB and ROS2 APIs is necessary, e.g., to publish the current velocity of the vehicle to a display that exhibits the rider's velocity.

According to the ROS2 mechanism, subsystems interested in data from the simulation subscribes to a ROS2 topic. For these subsystems to integrate with the simulation, another node or nodes have to publish data on the same topic, as shown in Figure 16. For example, a collision detection DPU receives information about the rider's surroundings in the simulation environment and detects possible collisions with other road users. Whenever a collision is detected, this DPU publishes a message to the warning subsystems to warn the rider.

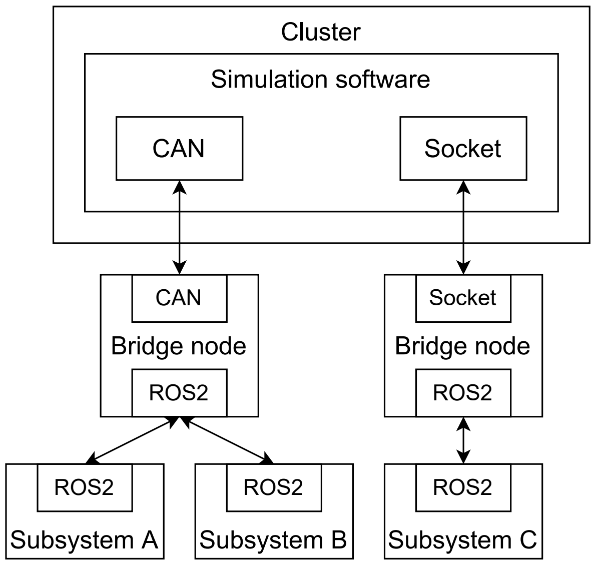

Figure 15: ROS2 integration onto a simulator that does not support custom software modules.

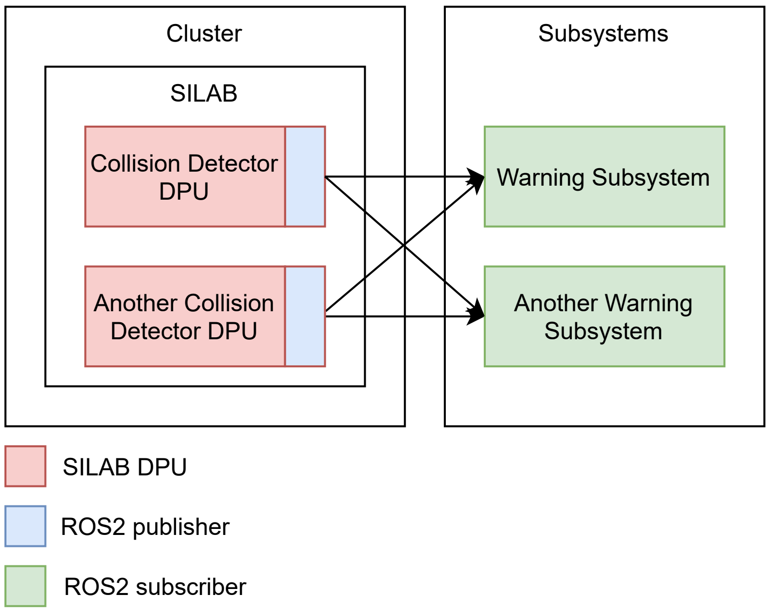

Figure 16: SILAB integration with other subsystems when the data flows from the simulation.

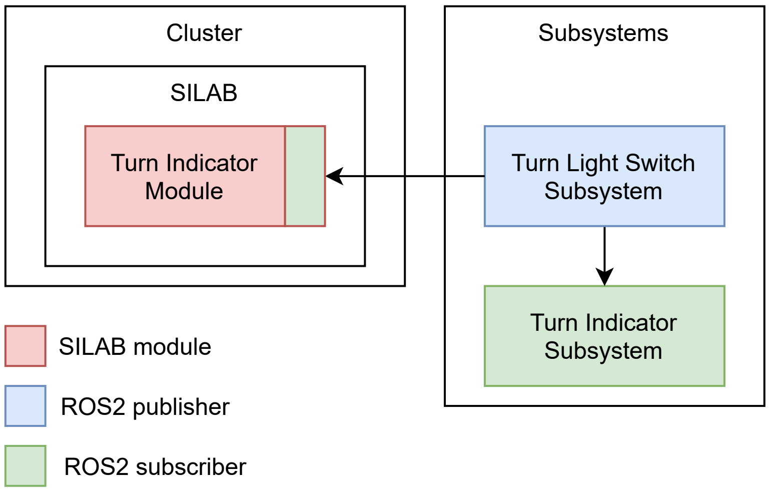

Figure 17: SILAB integration with other subsystems when the data flows from the real world.

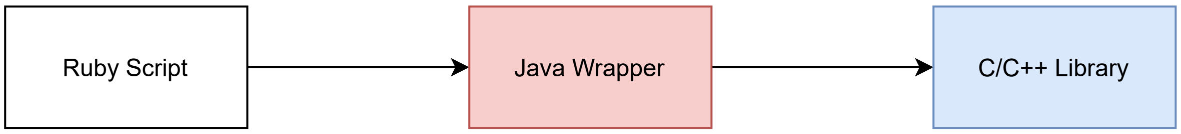

Figure 18: Access of Ruby DPUs to the ROS2 library.

On the other hand, some subsystems retrieve data from the real world and send data to the simulation. These applications use ROS2 API to publish data to a topic that another subsystem or SILAB DPU can subscribe to. For example, an application that detects the state of the motorcycle's turning indicator button in the Mock-up. Whenever the state of the button changes, the application publishes a message. A SILAB DPU receives this message and updates the motorcycle's turning light state in the simulation environment, as illustrated in Figure 17.

ROS2 offers C++ and Python APIs, and SILAB allows the creation of DPUs with C++, Java, and Ruby. The current SILAB license owned by the University of Minho, however, only allows the development of Ruby DPUs, an issue that had to be addressed, increasing the solution's complexity. Using ROS2's C++ API directly would have been possible with an upgraded version of SILAB's license.

Regarding the license issue, the goal was to use Ruby's C/C++ native interface functionality to access C/C++ libraries, more specifically, to access ROS2 libraries. However, SILAB uses JRuby, a version based on the Java Virtual Machine. In this specific Ruby version, C/C++ native interfaces are not accessible, unlike the original C-based implementation of the Ruby programming language. Instead, JRuby allows accessing Java bytecode. Since Java, like the original Ruby implementation, can access C/C++ native interfaces using the Java Native Interface (JNI), ROS2 is accessible from Ruby scripts with a bridge based on Java. JNI is a foreign function interface programming framework that enables Java code to invoke libraries written in C/C++ by loading dynamic library files. In the end, these bridges increase the complexity of a ROS2 DPU because creating Ruby scripts that access a Java wrapper is necessary, which in turn accesses C/C++ libraries, as shown in Figure 18.

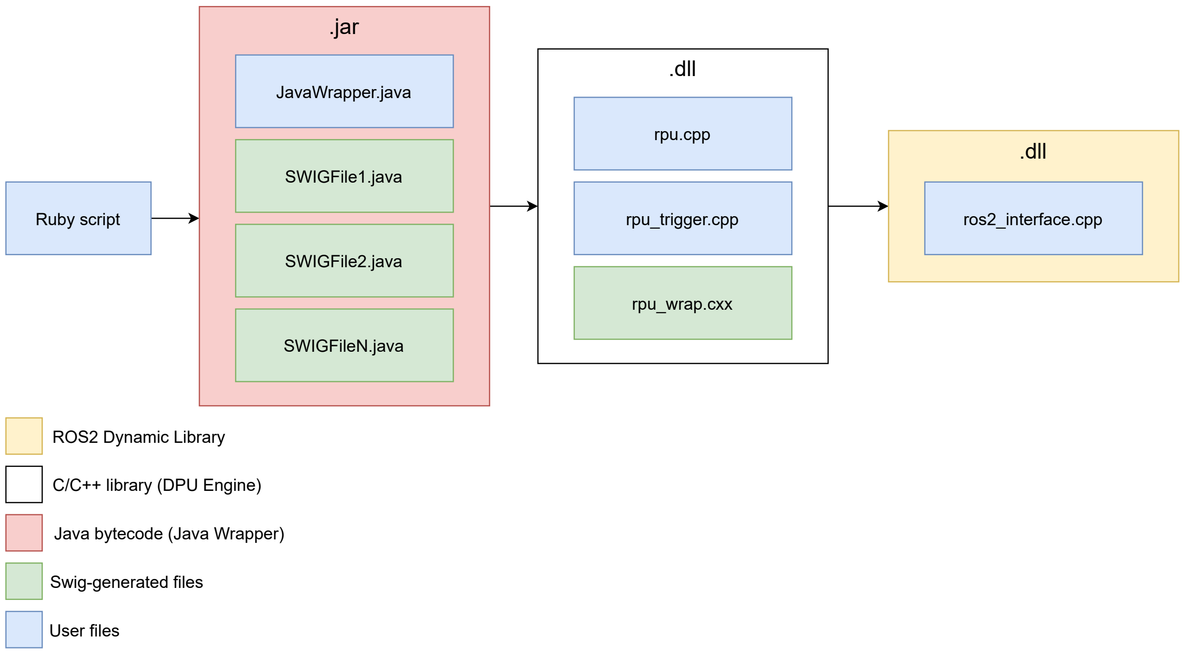

Figure 19: User and SWIG generated files.

As mentioned above, built-in functions in JRuby allow to access Java bytecode. However, the bridge between Java and C/C++ native interfaces is more complex since JNI needs regular Java files and some wrappers. Instead of creating and writing these wrapper files, a tool like the Simplified Wrapper and Interface Generator (SWIG) [28] can generate them automatically. SWIG is open-source software that generates wrappers allowing other languages, like Java or Ruby, to access C/C++ functions and data types. SWIG requires an interface file that contains the list of functions to export in order to generate a C/C++ wrapper file and the target programming language code. In short, SWIG generates the source code to glue C/C++ native interfaces with the desired target language.

Besides the Ruby-Java and Java-C/C++ native interfaces bridges, another one is introduced. All ROS2 related code is separated from the C/C++ original library. In this way, the computation to publish a message is separated from the DPU code. The Ruby, Java, and C/C++ layers and their required files are shown in Figure 19.

4.3 INTEGRATION IMPLEMENTATION

The development of a ROS2 DPU has several building blocks. First, developing the ROS2 Dynamic Library is necessary. The ROS2 Dynamic Library is a regular ROS2 package containing all ROS2 related code: node, publisher and subscription creation, message publication, and message handling. This package is compiled like a regular ROS2 package and exported as a C++ dynamic library.

After that, creating the DPU Engine that contains the logic to transit between SILAB states is necessary. The DPU Engine logic access ROS2 Dynamic Library to use its functionalities. The DPU Engine is the heart of a

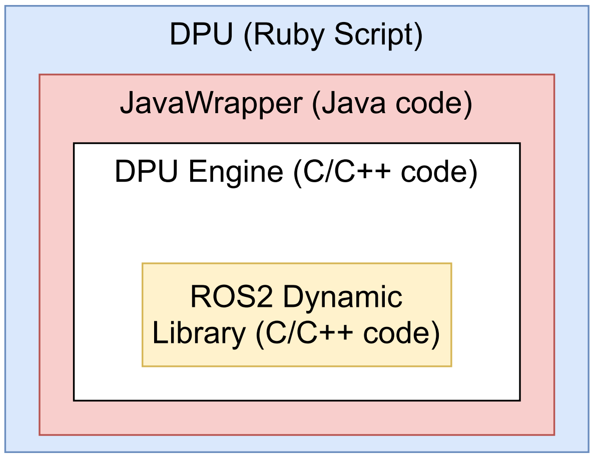

Figure 20: Layers- to access ROS2 from Ruby scripts.

ROS2 DPU. In this layer, the logic of the DPU to transit between the *Off* to the int*Initialized* states and its behavior during the *Running* state is implemented.

The next building block involves SWIG. It starts by creating a SWIG interface file from which SWIG generates one C/C++ and several Java Wrapper files. With all C/C++ files created, the DPU Engine is compiled into a dynamic library.

Then, creating a specific JavaWrapper.java file that will access the C/C++ dynamic library using JNI is required. After that, all the Java SWIG generated files and the JavaWrapper.java file are compiled into Java bytecode. Finally, the Ruby script accesses the Java bytecode, which in turn accesses the DPU Engine, which in turn accesses the ROS2 Dynamic Library, as shown in Figure 20. In this strategy, Ruby and Java are merely used to access this DPU Engine. It should be noted that for each new DPU all the layers and bridges have to be created.

Altogether, ROS2 DPUs are designed to contain a single node with only one publisher or one subscriber. If a regular node must have a subscription and a publisher, the node can be split into two different DPUs. The first DPU only has one node with one subscription that updates its outputs according to the received messages. The second DPU connects its inputs to the subscriber DPU's outputs to detect when and what messages to publish.

As stated before, SILAB has a main state machine that dictates the DPUs logic between state transitions and in the Running state. With this in mind, ROS2 DPUs initialize nodes, publishers, and subscriptions during the Off to the Initialized state transition. Depending on the DPU type, i.e., it should act as a publisher or as a subscriber, the Running state implementation is different. On the one hand, each ROS2 publisher DPU implements a simple algorithm to determine if it has to publish a message. On the other hand, each ROS2 subscriber DPU handles the received messages and updates the DPU's outputs. Since ROS2's middleware requires global resources per process, each simulation must run a special ROS2 DPU for this specific task. This DPU initializes the required global resources in the Off to the Initialized state transition. Besides, this special DPU releases and destroys any global resources going to the Off state. The process executed by each DPU type between state transitions is shown in Figure 21.

All use cases that require ROS2 should separate the use case's main algorithm from the message exchange part. ROS2 DPUs should simply implement the logic to publish a message or what to do when receiving one.

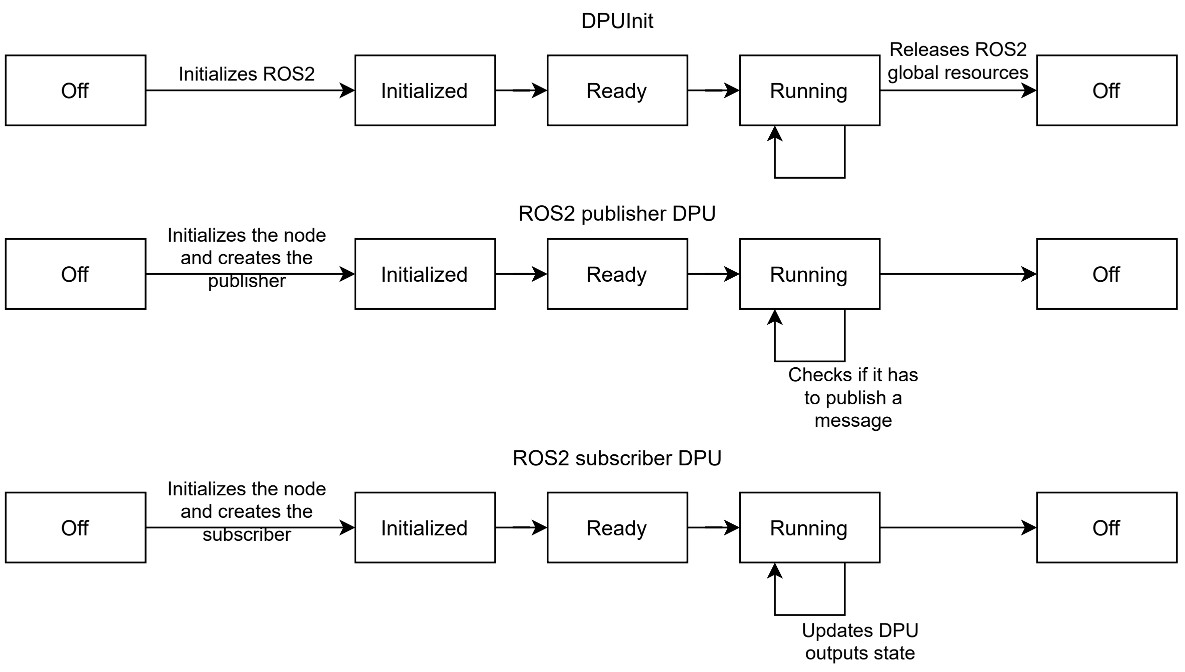
DPUInit
Off
Initializes ROS2
Initialized
Ready
Running
Releases ROS2 global resources
Off
ROS2 publisher DPU
Off
Initializes the node and creates the publisher
Initialized
Ready
Running
Off
Checks if it has to publish a message
ROS2 subscriber DPU
Off
Initializes the node and creates the subscriber
Initialized
Ready
Running
Off
Updates DPU outputs state

Figure 21: ROS2 DPUs responsibilities between state transitions.

ROS2 publisher DPUs should have as input the use case's DPU output. On the other hand, ROS2 subcriber DPUs should output their state to the use case's DPU. For example, a use case that requires computing the danger for several specific situations. All the danger detection algorithms are implemented in Hazard Detector DPUs. The Hazard Detector DPUs output their computation to a Danger Publisher DPU. From their input values, the Danger Publisher DPUs publish a message. Since the Hazard Detector DPUs belong to the same use case, instead of creating multiple ROS2 DPU classes, instantiate multiple times the same class is possible, as shown in Figure 22. This strategy allows to separate ROS2 from the hazard detection algorithms. In addition, this separation makes the replacement of the current middleware easier.

4.4 CROSSOVER SOFTWARE MODULES GENERATOR

Even though ROS2 DPUs offer several advantages, bridging Ruby scripts with ROS2 is much more complex than other extension mechanisms. To avoid creating every file for each DPU (the user files shown in Figure 20), which is error-prone, a tool was developed. This tool generates the necessary ROS2 code, boilerplate code to bridge SILAB and ROS2, and SILAB's scripts and configuration files. These files can be generated automatically since part of the code follows a pattern. The tool has two main components, the Parser and the Generator.

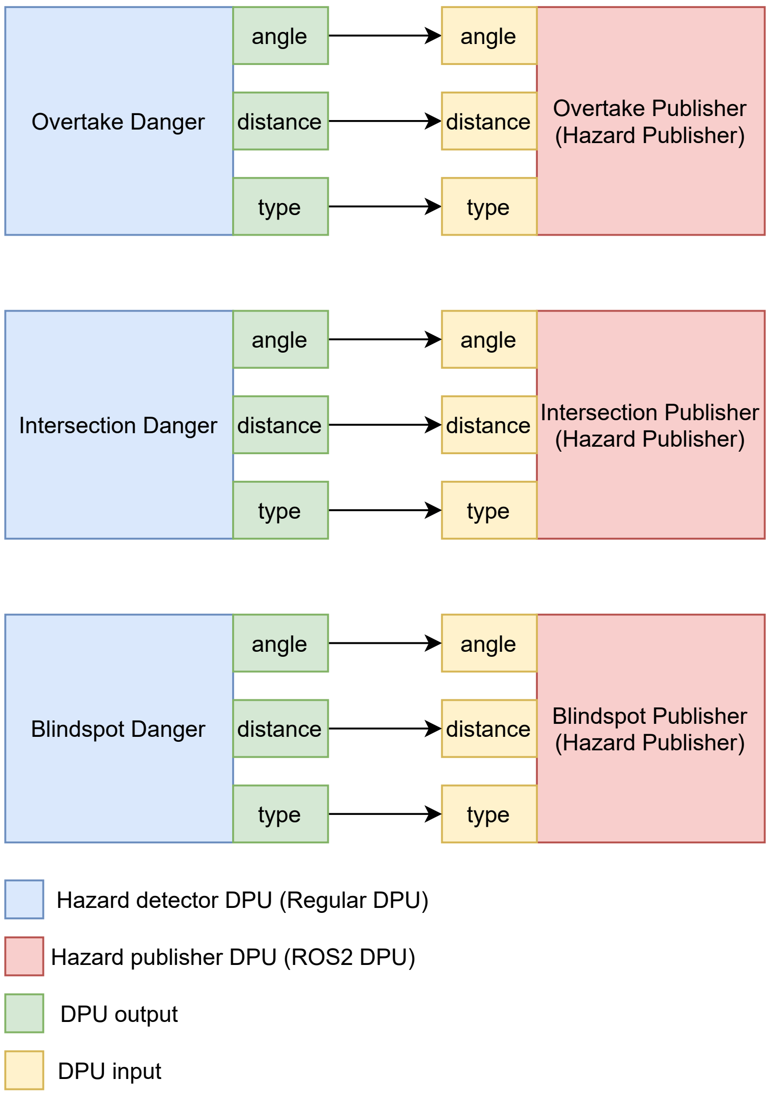
Overtake Danger
angle
distance
type
angle
distance
type
Overtake Publisher (Hazard Publisher)
Intersection Danger
angle
distance
type
angle
distance
type
Intersection Publisher (Hazard Publisher)
Blindspot Danger
angle
distance
type
angle
distance
type
Blindspot Publisher (Hazard Publisher)
Hazard detector DPU (Regular DPU)
Hazard publisher DPU (ROS2 DPU)
DPU output
DPU input

Figure 22: Example of a ROS2 publisher DPU use case.

4.4.1 *Parser*

The Parser reads a descriptive file to create, organize, and structure the data for the Generator. Listing 4.1 has a snippet of an example of a descriptive file. This descriptive file should contain the information of all publishers and subscribers DPUs to generate. In this example, the subscribers field is empty, but a subscriber is identical to a publisher.

```
publishers:
  - hazard_detection_city:
      description: 'DPU that detects any hazard in the city road segment'
      license: 'MIT License'
      index: 100
      computers:
        - Operator
      message: std_msgs/msg/int32
      node_name: hazard_detection_city
      topic_name: hazard_detection_topic
      fields:
        - data:
            type: float
            default: 0
      inputs:
        - hazard_x:
            type: float
            default: 0
            connection: DataSingle.X1
        - hazard_y:
            type: float
            default: 0
            connection: DataSingle.Y1
        - ego_x:
            type: float
            default: 0
            connection: VDyn.X
        - ego_y:
            type: float
            default: 0
            connection: VDyn.Y
      parameters:
      outputs:
        - hazard_angle:
            type: float
            default: -1
            connections:
              - XYZ.x
              - XYZ.y
        - hazard_distance:
            type: float
            default: -1
            connections:
              - XYZ.a
              - XYZ.b
subscribers:
```

Listing 4.1: Example of a descriptive file to be read by the Parser.

The tool uses each element of the publishers and subscribers fields to create a different ROS2 package and the necessary files to bridge ROS2 with SILAB DPUs. Each publisher and subscriber have multiple key-value pairs to describe the characteristics of each package.

Regarding the ROS2 information, the descriptive file should contain the message type and the topic name the node should publish or subscribe to. Also, it should include the description and license fields to update the package's meta-information. Besides, defining the message's fields and the node's name is necessary.

Finally, the last three fields are the inputs, parameters, and outputs of the DPU. All of them have type and default values, which tell the C/C++ type and the default value of each variable. The difference between the inputs, parameters, and outputs is their connection field. Since the parameters are static values, they do not have any input or output connections. DPU inputs can not have more than one connection, so this key is only a string. Unlike inputs, outputs can have multiple connections.

4.4.2 *Generator*

With the information provided by the Parser, the Generator has two different tasks. It starts by creating the folder structure of the extended ROS2 packages. Then, it generates the DPUs files and the compilation scripts.

Each file to generate has a Jinja2 [29] template, like the one shown in Listing 4.2. Jinja2 is a templating engine that allows to write template files with special placeholders. For example, Jinja2 allows to replace parts of the template with one value. The Generator combines the Parser data with Jinja2 templates to generate the final files. Besides, the Generator renders three scripts that are placed in the root of the workspace. These scripts have the goal to save time and reduce the risk of errors during the compilation sequence of all DPUs. The first script sets some necessary environment variables and compiles the ROS2 Dynamic Library. The second script compiles the DPU Native Interface using Cygwin and Mingw-w64. Finally, the third script compiles the Java Wrapper library.

```
#include "rpu_auxiliary_variables_{{ package.package_name }}.h"

struct Rpu{{ package.package_class_name }}Variables
{
    {% for in_ in io.inputs %}
    {{ in_.input_type_c }} {{ in_.input_name }};
    {% endfor %}
    {% for parameter in io.parameters %}
    {{ parameter.parameter_type }} {{ parameter.parameter_name }};
    {% endfor %}

    {% for output in io.outputs %}
    {{ output.output_type_c }} {{ output.output_name }};
    {% endfor %}
};

class Rpu{{ package.package_class_name }}
{
public:
    Rpu{{ package.package_class_name }}();
    ~Rpu{{ package.package_class_name }}();
    void start({% if func.start_definition_c %}{{ func.start_definition_c }}, {%
        endif %}char* node_name, char* topic_name);
    void update({{ func.update_definition_c }});
    void trigger(double time, double time_error);
    void release();
    {% for output in io.outputs %}
    {{ output.output_type_c }} get_{{ output.output_name }}();
    {% endfor %}

    Rpu{{ package.package_class_name }}Variables rpu_variables;
    void* ros2_interface;
    Rpu{{ package.package_class_name }}AuxiliaryVariables aux_variables;
};
```

Listing 4.2: Jinja2 template used by the Generator to render one of the files.

5

VALIDATION

In this chapter, the subsystems incorporated onto the simulator, the scenarios developed, and the implemented DPUs in order to validate the concept are described. The incorporated subsystems include the headphones, the haptic jacket, the smart glasses, the jacket red lights, the turn indicator gesture detector, the turn indicator wrist lights, and the haptic gloves. The subsystems are aggregated and described according to their use case. First, the headphones, the haptic jacket, and the smart glasses provide warnings related to hazards. Second, the jacket red lights is related to the motorcycle's brakes. Then, the turn indicator gesture detector and the turn indicator wrist lights belong to the turn indicator use case. After that, the orientation use case is only composed of the haptic gloves subsystem. Finally, the main advantages perceived of using middleware like ROS2 is shown and how the integrated subsystems, in the end, validate the solution proposed in this dissertation.

5.1 HAZARD DETECTION

The first set of subsystems belongs to the hazard detection use case. This use case contains the headphones, the haptic jacket, and the smart glasses subsystems. These subsystems warn the rider of hazards when facing specific situations in a SILAB scenario. These situations include the blindspot, the intersection, and the overtake. In the blindspot, the rider has to enter a highway, and a car is in their blindspot. In the intersection, the rider passes an intersection, and a car dangerously comes from the right. In the overtake, the rider has to risk a dangerous overtaking maneuver on the opposite lane.

All components related to the hazard detection use case are shown in Figure 23. On the SILAB side, this use case has three regular DPUs that implement the hazard detection algorithms and three ROS2 publisher DPUs. The subsystems side has three controller subscribers and three warning applications that warn the rider according to the hazards detected on the SILAB side.

5.1.1 *SILAB*

In this subsection, first, how the regular DPUs compute the distance and the azimuth angle to the hazard is described. Then, the ROS2 publisher DPUs are characterized. Finally, each situation the rider has to face is described.

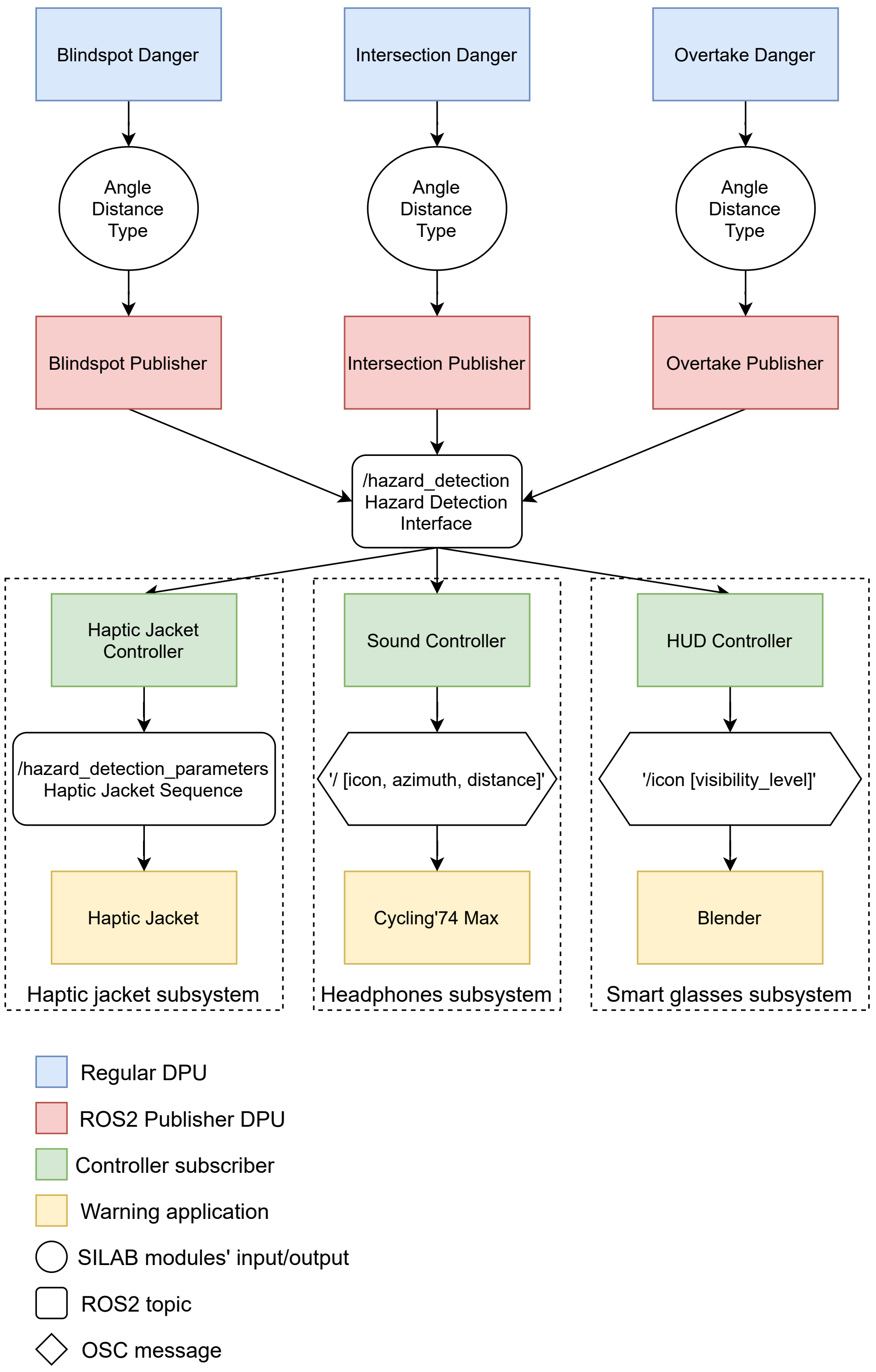

Figure 23: Hazard detection use case: complete diagram.

For each situation the rider has to face, regular DPUs compute the distance and the azimuth angle to the hazardous vehicle. These DPUs receive the rider's and the hazardous vehicle's current coordinates to compute the distance to the hazard using the Euclidean equation

$$distance = \sqrt{(r_x - h_x)^2 + (r_y - h_y)^2} \tag{1}$$

where r_x, r_y, h_x, and h_y are the rider's and the hazard's x and y coordinates in the simulation world, respectively. Besides, they receive the rider's yaw rotation to compute the azimuth angle of the hazard

$$\begin{aligned} \vec{v} &= \cos(yaw) \cdot \vec{i} + \sin(yaw) \cdot \vec{j} \\ \vec{u} &= (h_x - r_x) \cdot \vec{i} + (h_y - r_y) \cdot \vec{j} \\ \gamma &= \text{atan2}(det(v, u), \vec{v} \cdot \vec{u}) \end{aligned} \tag{2}$$

where $\vec{v}$ is the rider's orientation, $\vec{u}$ is the hazard's direction from the rider perspective, yaw is the orientation of the rider, r_x, r_y, h_x, and h_y are the rider's and the hazard's x and y coordinates in the simulation world, respectively. These DPUs output the result of the distance and azimuth angle computation and which particular situation is causing the danger, i.e., if the dangerous event is caused by an overtake, intersection, or blindspot.

In the designed SILAB scenario, three different regular DPUs compute the hazardous situations: the Blindspot Danger, the Intersection Danger, and the Overtake Danger. Each one of these regular DPUs is connected to one ROS2 publisher DPU. These ROS2 publisher DPUs have only one function: to publish a specific ROS2 message with the data provided by the hazard detection DPUs. Only when the regular DPUs output specific values, the ROS2 publisher DPUs publish hazard-related messages.

Blindspot

The first situation designed to test the rider consisted of the blindspot event. In this event, the rider enters an acceleration lane to access a highway, as shown in Figure 24 from the birds-eye perspective. When the rider enters the acceleration lane, a vehicle is strategically positioned on the highway to be in the rider's blindspot. This vehicle is programmed with a behavior model that synchronizes its velocity with the rider's and follows its blindspot, making a potential collision likely to happen when the rider moves from the acceleration ramp to the highway. The vehicle's velocity is programmed to work only if the rider's velocity is above a specific threshold. So, if the rider decreases its velocity in advance, it immediately reduces the risk of a collision. At the start of the acceleration lane, in order to guarantee that the rider does not spot the hazardous vehicle beforehand, the acceleration lane is placed near a traffic island with several bushes, trees, and other objects.

For this situation, the Blindspot Danger DPU, as shown in Figure 23, is the one responsible to compute the blindspot hazard. Within the blindspot scenario module, two invisible checkpoints in the road delimit the dangerous zone. These two points were placed using the SILAB hedgehog feature. When the rider passes through the first checkpoint, placed at the start of the acceleration lane, the Blindspot Danger DPU knows the rider is in the dangerous zone. Being the rider in the dangerous zone, the Blindspot Danger DPU continuously computes the azimuth angle to the hazardous vehicle. If this azimuth angle is between two threshold values,

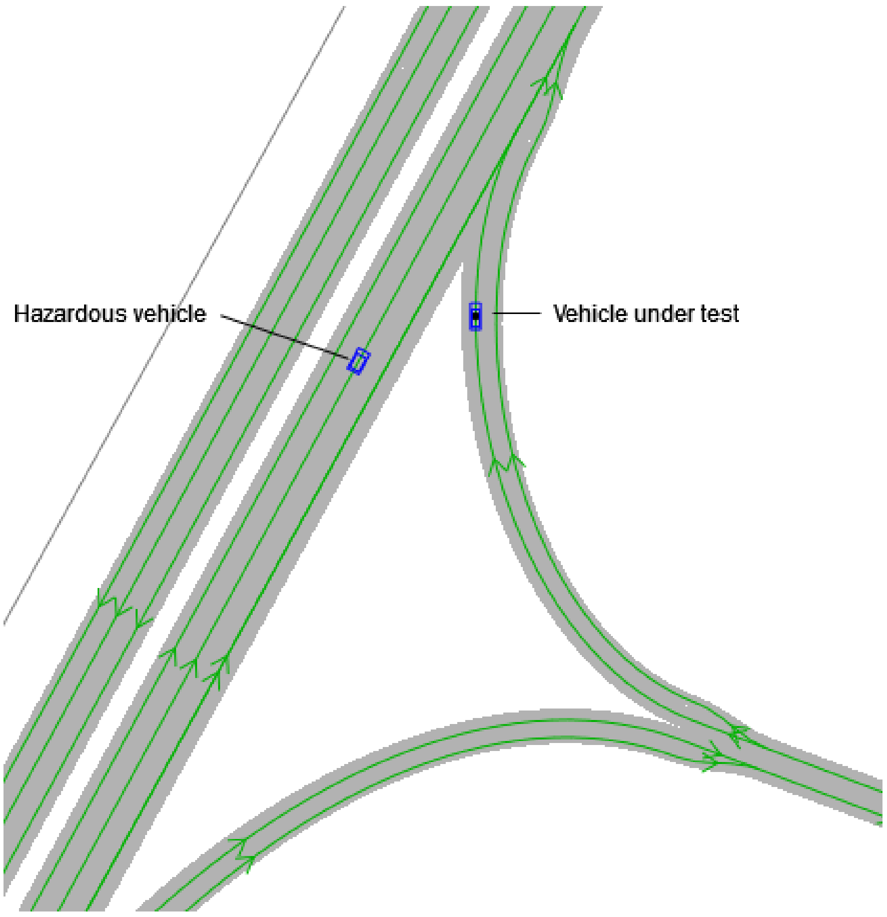

Figure 24: Blindspot scenario module from the birds eye perspective.

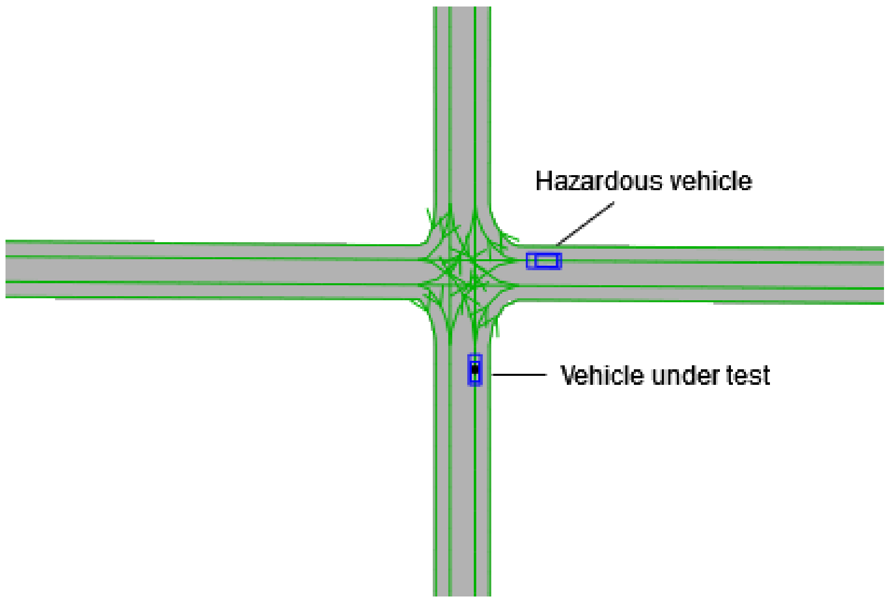

Figure 25: Intersection scenario module from the birds eye perspective.

the Blindspot Danger DPU detects that the vehicle is in the rider's blindspot. When the rider passes through the second checkpoint, which is placed right after the ending of the acceleration lane, the Blindspot Danger DPU knows the rider left the dangerous zone and stops detecting the danger.

Intersection

In the second event, the rider has to pass an intersection, as shown in Figure 25 from the birds-eye perspective. In this event, even though the rider has priority (a traffic signal indicating this is placed a couple of hundred meters from the intersection's center), a vehicle, ignoring the traffic rules, comes dangerously from the right side and does not stop. As in the blindspot, this vehicle has a behavior model that allows synchronizing its velocity with the rider's to increase the probability of a collision. Also, SILAB's flowpoint feature is used to increase the chances of a collision. This feature allows invisible checkpoints to be placed on the road that will command the hazardous vehicle to update its behavior, e.g., its velocity, when the rider passes through these points. In this specific event, the flowpoint commands the hazardous vehicle to abruptly reduce the velocity just before the center of the intersection, further increasing the probability of a collision. Even though the hazardous vehicle is programmed to follow the rider's velocity, the rider can avoid the hazard by reducing beforehand its velocity. The hazardous vehicle is hidden from the rider since the scenario has several objects on the rider's right side, e.g., houses.

For this particular situation, the Intersection Danger DPU is the one responsible to detect when the rider is in danger. First, the Intersection Danger DPU estimates the time for the rider to reach the center of the intersection. If the time is below a specific threshold, in this case, six seconds, the hazard is detected. During this period, the Intersection Danger DPU computes the distance and azimuth angle to the hazardous vehicle.

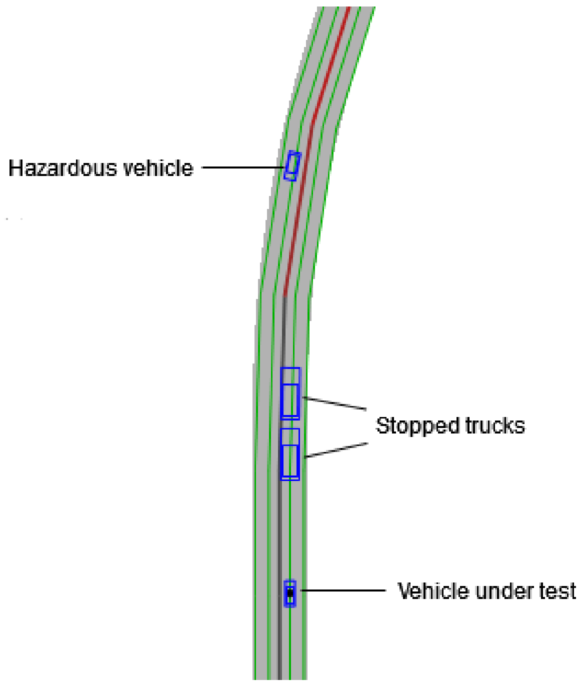

Figure 26: Overtake scenario module from the birds eye perspective.

The Intersection Danger DPU stops detecting the hazard when the rider or the hazardous vehicle passes the center of the intersection.

Overtake

For the last situation, the rider is subjected to overtake a couple of semi-trailer trucks stopped on the road (e.g., the trucks have broken down) occupying his lane, as shown in Figure 26. Since the road only has two lanes, the rider has to risk a maneuver on the opposite lane. Having a couple of trucks allows to hide a vehicle that will emerge on the opposite lane and endanger the situation. Unlike the other two events, this one uses a DPU to control the vehicle's behavior by commanding it to accelerate. This DPU acts like a flowpoint, but instead of being activated by passing through some invisible checkpoint, it is triggered by the Overtake Danger DPU.

The Overtake Danger DPU is the one responsible to compute the hazard in this particular situation. It starts by estimating the time for the rider to reach the trucks. Just like the intersection, when the time is below a specific threshold, in this case, six seconds, this DPU detects a dangerous situation. The Overtake Danger DPU computes the distance and azimuth angle to the hazardous vehicle during the dangerous period. The Overtake Danger DPU stops detecting the hazard when the hazardous vehicle crosses the rider.

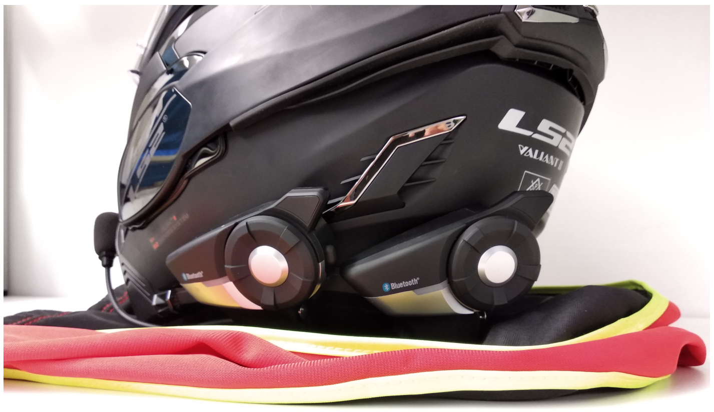

Figure 27: Headphones and microphone modules placed on the helmet's left side.

5.1.2 *Subsystems*

Three different warning modalities are set to transmit information to the rider. Each warning modality has its own subsystem that receives hazard-related messages, processes them, and actuate in order to be perceived by the rider. The sources of the hazard-related messages are the Overtake Publisher, the Intersection Publisher, and the Blindspot Publisher ROS2 publisher DPUs.

Headphones

The headphones subsystem warns the rider using an audio modality. This subsystem is implemented with the help of Cycling'74 Max [30]. This software allows an audio track to be played with a specific azimuth angle for spatialization and a scale to control the audio volume. Having these abilities, from the values detected by the hazard detection DPUs, Cycling'74 Max can warn the rider of the distance to the hazard and where it is coming from through audio cues.

The rider listens to the audio from a module [31] placed on the helmet's left side, as shown in Figure 27. This module can reproduce stereo audio (the speakers are installed inside the helmet and are not visible). The audio comes from the computer in the Back-office connected to the module using Bluetooth, as shown in Figure 12.

Cycling'74 Max does not support ROS2's middleware. However, it offers a UDP listener to interface with external applications. This UDP listener is used to receive and process Open Sound Control (OSC) messages. So, a controller application, called Sound Controller, had to be developed. The Sound Controller has two responsibilities. First, to translate hazard-related ROS2 messages to the OSC protocol. Second, the audio is designed to play with a specific melody, managed by the Sound Controller. While the hazard detection DPUs detect a dangerous situation, the Sound Controller requests the Cycling'74 Max to play the audio track in a loop. Within this

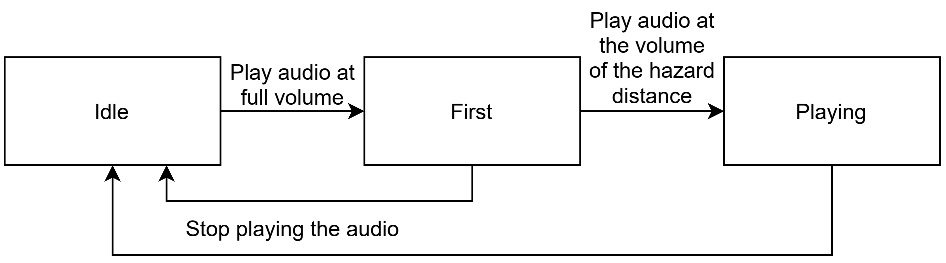
Idle
Play audio at full volume
First
Play audio at the volume of the hazard distance
Playing
Stop playing the audio

Figure 28: Sound Controller state machine.

loop, the first cycle is played with the maximum volume, while the next ones are played with a volume according to the hazard distance. Regarding the azimuth angle, the entire loop follows the direction of the hazard.

The Sound Controller uses a state machine to control the volume to play the audio track, as shown in Figure 28. It starts in the Idle state. When the Sound Controller receives the first hazard-related message, it changes its state to First. After receiving the first message, the Sound Controller requests Cycling'74 Max to play the first cycle of the loop with the maximum volume and with the hazard azimuth angle. In this state, the Sound Controller requests Cycling'74 Max to update the playing azimuth angle every time a hazard-related message is received. After the first cycle, the Sound Controller goes to the Playing state. In this state, the Sound Controller requests Cycling'74 Max to play the audio track with the azimuth and volume according to the hazard angle and distance, respectively. After receiving hazard-related messages in this state, the Sound Controller requests Cycling'74 Max to update the playing azimuth angle and volume. After the DPUs stop detecting hazards, the Sound Controller goes to the Idle state, and Cycling'74 Max stops playing the audio cue.

Haptic jacket

The haptic jacket consists of 4 motors placed in strategic positions in the rider's jacket, as shown in Figure 29, that will vibrate as a warning of a hazard occurring in the direction of the vibration.

A 3D printed mold is designed to hold one battery, a Khadas Edge-V [32] board, and a Texas Instrument DRV2605LEVM-MD [33] board (with 8 DRV2605L motor drivers of which only four are used), as shown in Figure 30. This mold fits the necessary components into the rider's jacket. On the 3D printed mold, a switch button is connected to the battery to turn on and off the entire system. Regarding the physical connections between components, the battery powers the Khadas Edge-V board, which, in turn, powers the DRV2605LEVM-MD board. Moreover, two wires are used to connect the Edge-V to the DRV2605LEVM-MD that are used to transmit I2C signals to control the motor drivers, which in turn control the vibration of the motors. Finally, the DRV2605LEVM-MD is connected to four motors. These connections are shown in Figure 31.

A ROS2 package was created to implement this use case, where a subscriber node, called Haptic Jacket, is set to listen for hazard-related messages. Besides ROS2, another API is used for the control of the motors. This API uses the wiringPi [34] library to communicate with the DRV2605LEVM-MD board.

Between the Haptic Jacket and the ROS2 publisher DPUs, a controller application, called Haptic Jacket Controller, processes the distance and azimuth angles to the hazard. Even though the Haptic Jacket application

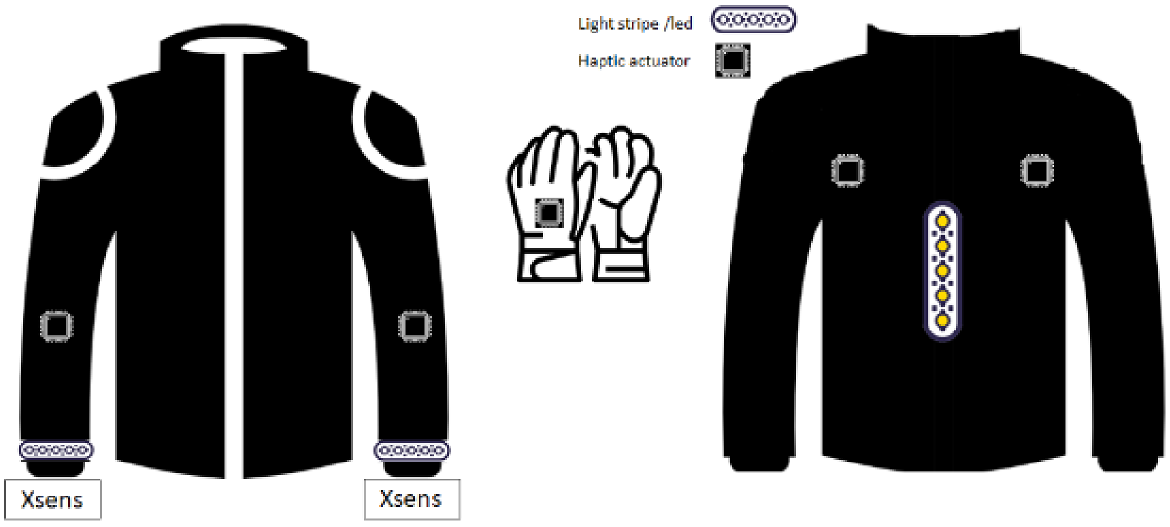

Figure 29: Placement of concepts in the rider's jacket and gloves.

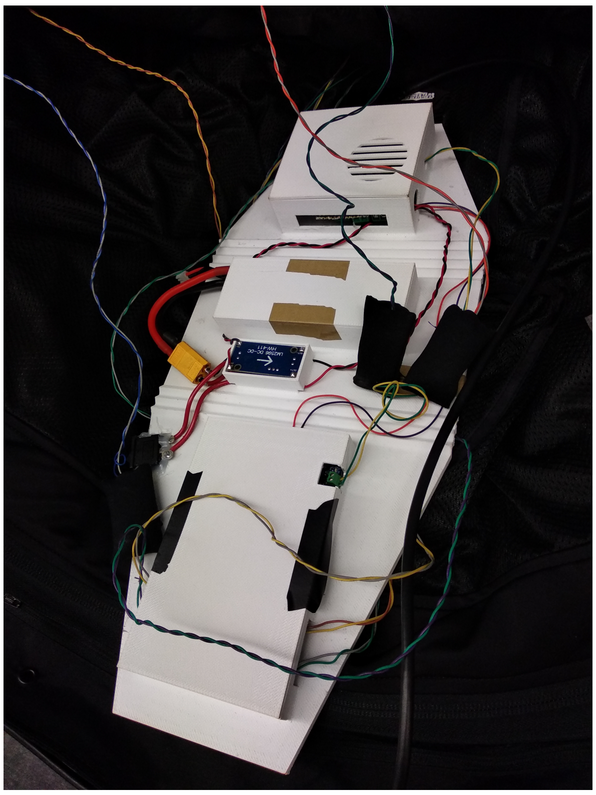

Figure 30: 3D printed mold with the haptic jacket components.

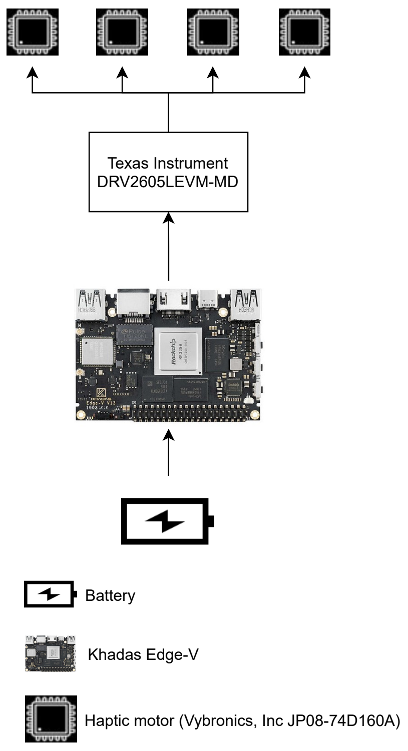

Figure 31: Hapitc jacket hardware connections.

receives ROS2 messages, it also has a controller node to keep a coherent architecture between all modalities. From the azimuth angle, distance to the hazard, and the type of situation, the Haptic Jacket Controller indicates the duration and intensity of the motors by publishing this data to the Haptic Jacket application. According to the situation type, the Haptic Jacket Controller dictates which motor vibrates.

Besides, the Haptic Jacket Controller has a state machine to control the motors' behavior, as shown in Figure 32. The Haptic Jacket Controller starts in the Idle state. After receiving a hazard-related message, it goes to the Vibrating state. In this state, the Haptic Jacket Controller computes the motors' vibration time using the hazard distance as a reference. After the vibration time, the Haptic Jacket Controller goes to the Stopped state. In this state, the Haptic Jacket Controller computes the amount of time the motors should be stopped, also using the hazard distance as a reference. After the stopped time, the application goes to the Vibrating state, and the process is repeated. Using this state machine allows the Haptic Jacket Controller to control the vibration and pause times to transmit at which distance the danger is. When the hazard detection DPUs no longer detect any hazard, the application goes to the Idle state requesting the haptic motors to stop vibrating.

Smart glasses

The last subsystem of this use case is the smart glasses [35]. Smart glasses provide a visual warning with augmented reality to the rider. These glasses are used to display icons of warning signs, the ones shown in Figure 33.

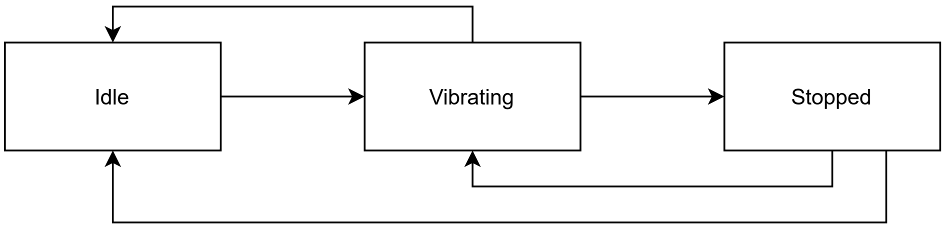

Figure 32: Haptic Jacket Controller state machine.

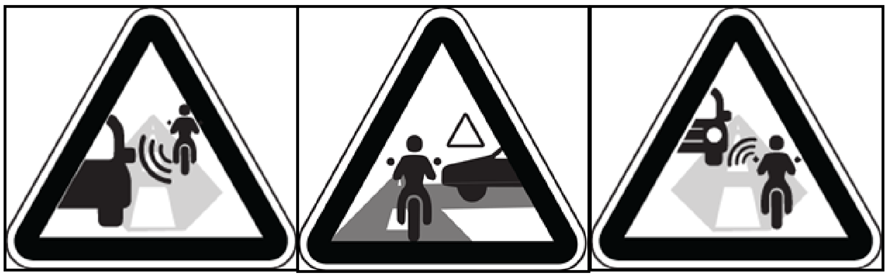

Figure 33: Displayed icons in the smart glasses.

A 3D printed mold is designed to fit in the helmet and to support the smart glasses, as shown in Figure 34. The only connection in this system is a USB-C cable, which is connected to the Khadas Edge-V board, shown in Figure 31, for power and to display a Blender application.

Blender 2.79 [36] is open-source software that allows to create animated movies, 3D models, interactive applications, and computer games using the Blender Game Engine. The Blender application is interactive and aims to provide visual cues to the rider whenever a hazard is detected. Besides, the icons are displayed with a specific amount of red level that depends on how threatening the situation is. To make interactive applications, Blender provides a Python API. However, Blender only supports Python 3.6, which is not compatible with ROS2 Foxy Fitzroy. From this setback, a controller called HUD Controller to translate ROS2 messages into the OSC protocol was developed.

The HUD Controller is designed to process the hazard-related messages published by the ROS2 publisher DPUs. From this processing, the HUD Controller requests the Blender application to change the displayed icon and the amount of red level. The HUD Controller follows the state machine shown in Figure 35.

More specifically, as soon as any ROS2 publisher DPUs publishes a hazard-related message, the HUD Controller processes the type of the hazard and requests the Blender application to display the icon with the lowest/no red level. Only after at least two seconds and when the situation hazard level increased, the HUD Controller requests the Blender application to change to the second red level. This level operates in the same way as the first one since only after at least two seconds, and when the hazard level increases, does the application requests the third and highest red level. Finally, the smart glasses display the highest red level until the hazard detection DPUs no longer detect any hazard.

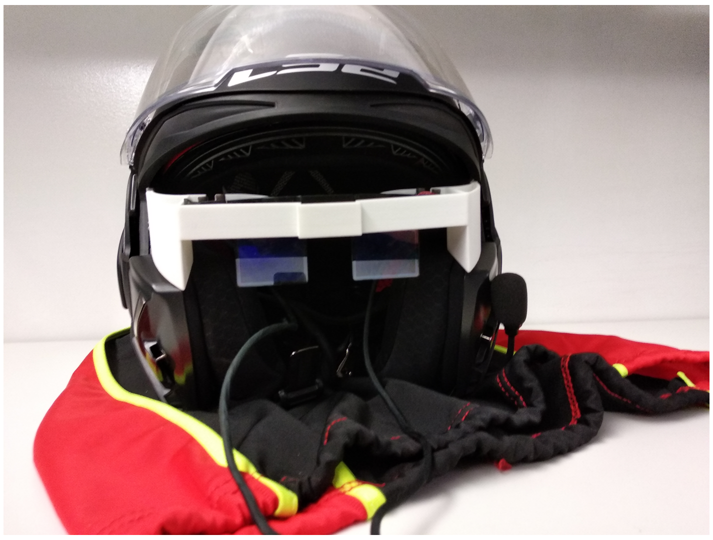
Figure 34: 3D printed mold with the smart glasses placed inside the helmet.

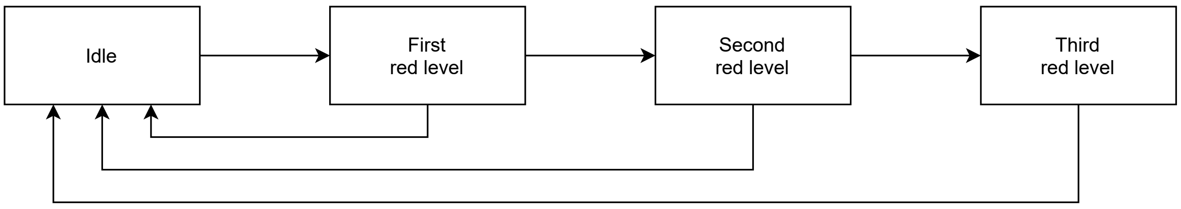

Figure 35: HUD Controller state machine.

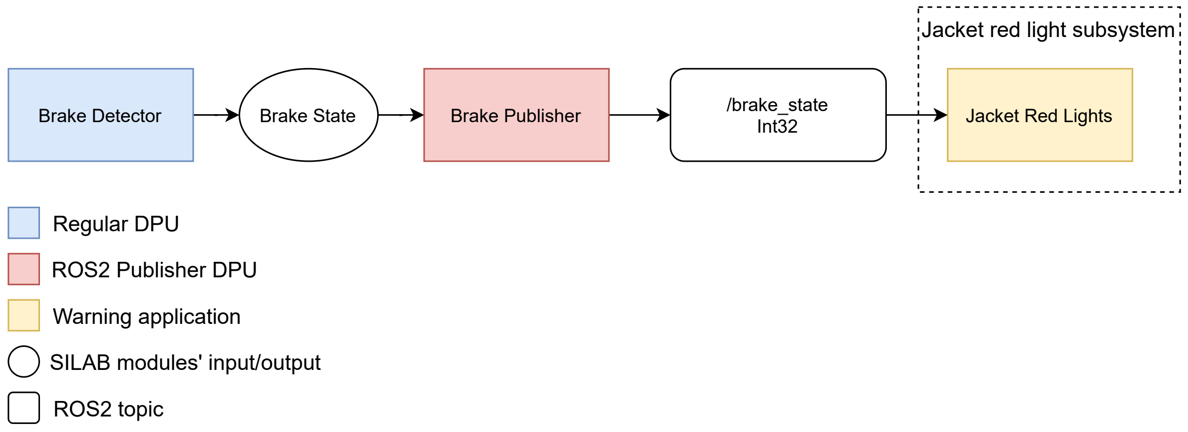

Figure 36: Jacket brake light use case: complete diagram.

5.2 JACKET BRAKE LIGHT

The second set of subsystems belongs to the jacket brake light use case. This use case only contains the jacket red light subsystem. The jacket red light subsystem is not intended for the rider to receive warnings but rather for the ones behind him. The jacket red light subsystem has a set of LEDs on the jacket back, as shown in Figure 29.

The components of this use case are shown in Figure 36. On the SILAB side, this use case has a regular DPU that implements the algorithm to detect when the rider is braking and a ROS2 publisher DPU. The subsystems side is composed of a ROS2 node that warns the ones behind the rider when the brakes are being pressed.

On the SILAB side, first, the Brake Detector DPU computes the state of the brakes. This DPU computes the brake's state by receiving, as input, the amount of pressure the rider applies to both the foot brake pedal and the hand brake lever. If the rider is pressing the pedal or the lever, the DPU outputs the rider is pressing the brakes. The Brake Detector DPU is connected to the Brake Publisher DPU, which receives the brake's state as input. Whenever the brake's state changes, the Brake Publisher DPU publishes a simple ROS2 message.

On the subsystems side, a ROS2 subscriber called Jacket Red Lights (running in the Khadas Edge-V board referenced in Section 5.1) is set to receive the published messages by the Brake Publisher. The Jacket Red Lights application lights up a set of LEDs in the rider's jacket when the rider is braking.

5.3 TURN INDICATOR LIGHTS

The third set of subsystems belongs to the jacket turn indicator lights use case. This use case contains the turn indicator gesture detector and the turn indicator wrist lights subsystems. The subsystems of this use case allow the rider to make gestures to signal a maneuver instead of the traditional way (i.e., using a lever) and make the rider more visible to others (the LEDs on the gesture side light up).

The components for this use case are shown in Figure 37. On the subsystems side, a node detects the gestures the rider is signaling. Besides, a different node controls the state of the LEDs according to the rider's

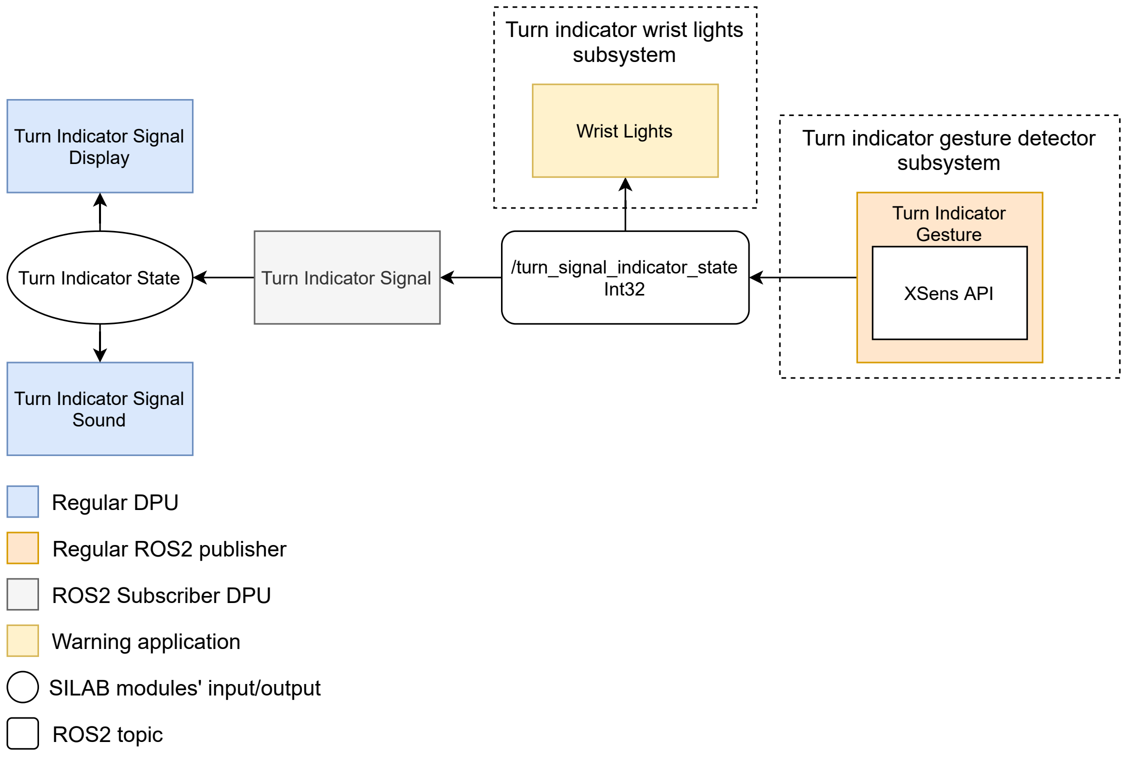
Turn indicator wrist lights subsystem
Wrist Lights
Turn indicator gesture detector subsystem
Turn Indicator Gesture
XSens API
Turn Indicator Signal Display
Turn Indicator State
Turn Indicator Signal
/turn_signal_indicator_state Int32
Turn Indicator Signal Sound
Regular DPU
Regular ROS2 publisher
ROS2 Subscriber DPU
Warning application
SILAB modules' input/output
ROS2 topic

Figure 37: Turn indicator lights use case: complete diagram.

gesture. SILAB side has a ROS2 subscriber DPU that updates its state according to the gesture detected by the turn indicator gesture detector subsystem and a couple of regular DPUs.

This use case, unlike all the others, has a subsystem that obtains data from the real world. To do so, a ROS2 package with an Xsens Dot [37] API was developed. This API allows to interface with the Xsens Dot accelerometers for data extraction. Using this data, the Turn Indicator Gesture (running in the Khadas Edge-V board referenced in Section 5.1) calculates if a user is performing a manual turning gesture and outputs the result as a ROS2 publication. Also, in the subsystems side, a ROS2 subscriber (running in the Khadas Edge-V board referenced in Section 5.1) is set to listen to the Turn Indicator Gesture publications. This node is called Wrist Lights and controls the state of the LEDs on each arm. When the rider signals the left turn indicator gesture, it lights up the left arm LEDs. On the other hand, when the rider signals the right turn indicator gesture, it lights up the right arm LEDs.

On the SILAB side, the Turn Indicator Signal is a ROS2 subscriber DPU that outputs to other regular DPUs the gesture the rider is signaling. The Turn Indicator Signal updates its outputs every time the Turn Indicator Gesture publishes a message. This ROS2 subscriber DPU is connected to a couple of regular DPUs, namely, the Turn Indicator Signal Display and the Turn Indicator Signal Sound. The Turn Indicator Signal Display displays in the instrument cluster a turn indicator icon. The Turn Indicator Signal Sound can play an audio file. Whenever the rider is making a gesture the icon and the sound cues are exhibited.

5.4 ORIENTATION

Finally, the last set of subsystems belongs to the orientation use case. This use case is only composed of the haptic gloves subsystem. The haptic gloves subsystem contains a couple of gloves with a haptic motor placed inside each one, as shown in Figure 29. Each time the rider must change direction, one of the haptic motors inside the gloves vibrates, indicating the path to take. More specifically, when the rider has to take a left exit, the left glove motor vibrates. On the other hand, when the rider has to take a right exit, the right glove motor vibrates.

The components of this use case are shown in Figure 38. The SILAB side has one regular DPU that implements the algorithms to verify when the rider has to change of direction and a ROS2 publisher DPU. The subsystems side has one controller subscriber and two micro-ROS nodes that warn the rider when he has to change direction according to the SILAB side.

On the SILAB side, first, the SILAB hedgehogs are used to specify where and when the rider is warned to change direction. The hedgehog feature allows placing invisible checkpoints in the scenario, and when the rider passes through it, the output value of the Orientation regular DPU are updated. Updating the Orientation output value makes the Orientation Publisher DPU to publish a ROS2 orientation-related message.

On the subsystems side, the Haptic Gloves Controller (running in the Khadas Edge-V board referenced in Section 5.1) receives the orientation-related messages indicating the direction the rider has to take. From these messages, the Haptic Gloves Controller defines how long and which haptic motor has to vibrate by publishing one ROS2 message to the correct topic. The message contains a melody that interpolates different vibration intensities with pauses like the Haptic Jacket Controller described in Section 5.1. The message published by the Haptic Gloves Controller is listened to by the Left Glove or the Right Glove applications. The glove applications follow the melody set by the controller and apply the defined vibration to a haptic motor. Unlike the other subsystems, the haptic gloves are set to run on an Espressif ESP32 microcontroller (on a SparkFun ESP32 Thing board). For this reason, the Left Glove and Right Glove are not a regular ROS2 node but a micro-ros node.

5.5 VALIDATION APPROACH

Two stages of development and testing were followed to implement the solutions mentioned earlier, having ROS2 an important role in both stages. In the first stage, the subsystems were developed and tested without any SILAB DPUs. They were tested using the ROS2 CLI and other tools to simulate the integration onto the simulation environment. In the same way, SILAB DPUs development is independent of any subsystem. Since both subsystems and SILAB DPUs were already validated, their integration is simple and is only necessary to make small adjustments. This approach shows that development can proceed on each side independently, with the advantages inherent to it, namely faster development and testing.

Besides, the portability and reusability have improved using ROS2 as the simulator interface. Such that, would not be necessary any change regarding the subsystems to integrate them onto a real motorcycle that offered a ROS2 middleware interface.

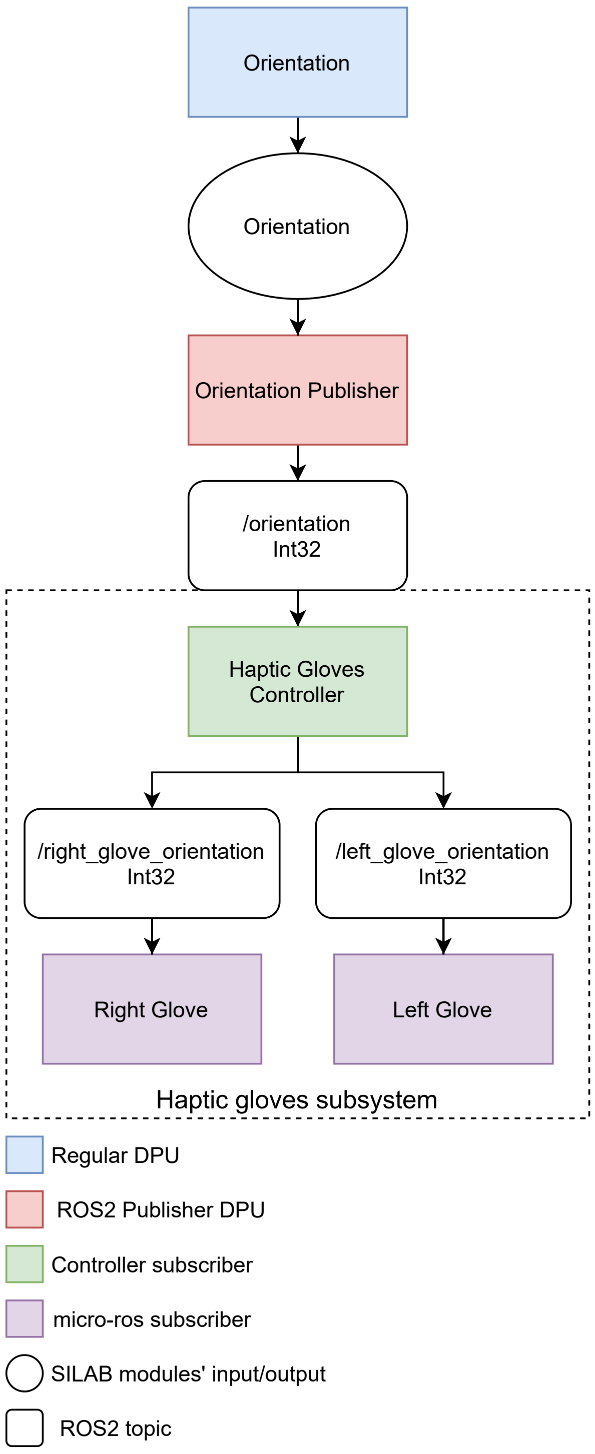

Figure 38: Orientation use case: complete diagram.

Finally, the simulator's expandability improved. So that subsystems using ROS2 interface are ready to be integrated onto the simulator, being only necessary to implement the algorithms to trigger those subsystems with the ROS2 extension mechanism.

6 CONCLUSION

The evolution of technology allows vehicles to have more and more interfaces for riders. Therefore, before using these interfaces in a real environment, testing them in a simulation environment can be useful, even though it is not the same as testing in the real world. Simulation environments typically include extension mechanisms for the simulation environment to be expandable, making it possible to integrate subsystems under test. Using well-known interfaces for these subsystem integration creates a coherent ecosystem for expandability and reusability.

6.1 DISSERTATION FINAL REMARKS

Simulation environments usually offer extension mechanisms to integrate new subsystems. These mechanisms are usually available in the form of APIs or TCP/IP sockets. The problem with these interfaces is that they are specific for each application. This dissertation presents a flexible and expandable simulation environment based on ROS2's middleware.

Several use cases were developed to validate this approach enabling ROS2's middleware to be responsible for the communication with different subsystems under test, bringing also several advantages: the subsystems are more portable, ready to be integrated onto a different simulator or even to be integrated into a real vehicle. besides, the simulator's expandability improved, reducing the difficulty to integrate new subsystems. Although the ROS2 SILAB DPU crossover is not so easy to implement, this is because of the available SILAB license. In the end, the advantages surpass this question even because a tool was developed to mitigate this problem. Also, the complexity in creating DPUs is in the computation itself and not on ROS2 integration. As a result, this approach is expected to be used in other projects.

A solution based on ROS2's middleware is better compared to the alternatives natively offered by SILAB, namely, CAN, network sockets, and Arduino extension mechanisms. First, by having the publish-subscribe messaging pattern, ROS2 allows to exchange messages and removes the need to manage the connections between subsystems and DPUs. The development of DPUs and subsystems is faster and more independent since ROS2 hides some of the complexities mentioned above and offers the CLI with multiple functionalities for testing and debugging. Subsystems can easily exchange messages considering that ROS2 messages are well-defined structures, unlike network sockets or CAN that usually work at the byte level. Using a well-known and tested tool reduces the risk of having hidden problems. ROS2, unlike the Arduino interface, is usable by more generic-purpose hardware and, unlike the CAN interface, does not need to be physically connected to the cluster. In addition, ROS2 is already being used in automotive contexts, e.g., for autonomous vehicles.

6.2 FUTURE WORK

The current solution complexity can be reduced if the University of Minho upgrades its SILAB license, allowing native C++ development. In this case, a simpler and cleaner solution can be implemented using directly ROS2 C++ API. In the most sophisticated approach, the ROS2 extension mechanism could be implemented by only using SILAB configuration files, just like SILAB native extensions. In the current implementation, the tool developed to ease the integration of ROS2 onto the simulation still has some margin for improvement. For example, some inputs from the user can be automated, and the tool can not generate the code for ROS2 messages that contain fields of other messages.

BIBLIOGRAPHY

[1] H.-P. Krüger, M. Grein, A. Kaussner, C. Mark, and others, “SILAB-A task-oriented driving simulation,” in *Proceedings of the Driving Simulator Conference (DSC) North America*, (Orlando, Florida), p. 9, 2005.

[2] Open Robotics, “ROS Index.” Available at: `https://index.ros.org`. Accessed: 2021-12-22.

[3] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, “A Self-Driving Car Architecture in ROS2,” in *2020 International SAUPEC/RobMech/PRASA Conference*, pp. 1–6, Jan. 2020.

[4] D. Basacik, N. Reed, R. Robbins, *et al.*, “Smartphone use while driving: a simulator study,” Mar. 2012.

[5] D. V. M. Nguyen, V. Ross, A. T. Vu, T. Brijs, G. Wets, and K. Brijs, “Exploring psychological factors of mobile phone use while riding among motorcyclists in vietnam,” *Transportation research part F: traffic psychology and behaviour*, vol. 73, pp. 292–306, 2020.

[6] R. Talbot, H. Fagerlind, and A. Morris, “Exploring inattention and distraction in the safetynet accident causation database,” *Accident Analysis & Prevention*, vol. 60, pp. 445–455, 2013.

[7] A. Werle and F. Diermeyer, “An investigation of smart glasses for motorcyclists as a head-up-display device-performed on a riding simulator,” in *International Conference on Applied Human Factors and Ergonomics*, pp. 226–237, Springer, 2021.

[8] V. Huth, F. Biral, Ó. Martín, and R. Lot, “Comparison of two warning concepts of an intelligent curve warning system for motorcyclists in a simulator study,” *Accident Analysis & Prevention*, vol. 44, no. 1, pp. 118–125, 2012.

[9] C. Lee and M. Abdel-Aty, “Testing Effects of Warning Messages and Variable Speed Limits on Driver Behavior Using Driving Simulator,” *Transportation Research Record*, vol. 2069, no. 1, pp. 55–64, 2008.

[10] M. M. Scheunemann and S. G. van Dijk, “ROS 2 for RoboCup,” in *RoboCup 2019: Robot World Cup XXIII* (S. Chalup, T. Niemueller, J. Suthakorn, and M.-A. Williams, eds.), (Cham), pp. 429–438, Springer International Publishing, 2019.

[11] G. Pinelli, P. Rajan, L. Berzi, and G. Savino, “The influence of body lean on the realism of a motorcycle riding simulator adopting counter-steering approach,” Sept. 2018.

[12] D. Ferrazzin, F. Salsedo, F. Barbagli, C. Avizzano, G. Di Pietro, A. Brogni, M. Vignoni, M. Bergamasco, L. Arnone, M. Marcacci, *et al.*, “The moris motorcycle simulator: an overview,” *SAE Transactions*, pp. 199–210, 2002.

[13] V. Cossalter, R. Lot, M. Massaro, and R. Sartori, "Development and validation of an advanced motorcycle riding simulator," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 225, no. 6, pp. 705–720, 2011.

[14] A. Stedmon, E. Brickell, M. Hancox, J. Noble, and D. Rice, "Motorcyclesim: a user-centred approach in developing a simulator for motorcycle ergonomics and rider human factors research.," *Advances in Transportation Studies*, no. 27, 2012.

[15] B. Westerhof, E. de Vries, R. Happee, and A. Schwab, "Evaluation of a motorcycle simulator," Symposium on the Dynamics and Control of Single Track Vehicles, 2020.

[16] E. Zeeb, "Daimler's new full-scale, high-dynamic driving simulator–a technical overview," *Actes INRETS*, pp. 157–165, 2010.

[17] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning* (S. Levine, V. Vanhoucke, and K. Goldberg, eds.), vol. 78 of *Proceedings of Machine Learning Research*, pp. 1–16, PMLR, 13–15 Nov 2017.

[18] Solace, "Advanced Event Broker. An event mesh for connected enterprises | Solace." Available at: `https://solace.com`. Accessed: 2021-11-29.

[19] Apache Software Foundation, "Apache Kafka." Available at: `https://kafka.apache.org`. Accessed: 2021-12-21.

[20] "Erlang – Erlang/OTP." Available at: `https://www.erlang.org`. Accessed: 2021-11-29.

[21] H. Bruyninckx, "Open robot control software: the orocos project," in *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, vol. 3, pp. 2523–2528, IEEE, 2001.

[22] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[23] Open Source Robotics Foundation, Inc., "Changes between ROS 1 and ROS 2." Available at: `http://design.ros2.org/articles/changes.html`. Accessed: 2020-10-26.

[24] Open Source Robotics Foundation, Inc., "ROS on DDS." Available at: `https://design.ros2.org/articles/ros_on_dds.html`. Accessed: 2021-06-24.

[25] Wikimedia Foundation, Inc., "Data distribution service." Available at: `https://en.wikipedia.org/wiki/Data_Distribution_Service`. Accessed: 2021-12-26.

[26] "micro-ROS." Available at: `https://micro.ros.org`. Accessed: 2021-12-21.

[27] WIVW, "Driving simulation and silab." Available at: `https://wivw.de/en/silab`. Accessed: 2021-12-21.

[28] "Simplified Wrapper and Interface Generator." Available at: http://www.swig.org. Accessed: 2021-11-29.

[29] "Jinja — Jinja Documentation (3.0.x)." Available at: https://jinja.palletsprojects.com/en/3.0.x/. Accessed: 2021-11-29.

[30] Cycling'74, "Cycling'74." Available at: https://cycling74.com/. Accessed: 2021-12-28.

[31] Sena Technology, Inc., "20S EVO - Motorcycle Bluetooth Headset with Built-In Intercom | Sena." Available at: https://www.sena.com/product/20s-evo. Accessed: 2021-12-09.

[32] Khadas Technology Co., Ltd., "Edge-V | Khadas." Available at: https://www.khadas.com/product-page/edge-v. Accessed: 2021-12-09.

[33] Texas Instruments Incorporated, "DRV2605LEVM-MD Evaluation board | TI.com." Available at: https://www.ti.com/tool/DRV2605LEVM-MD. Accessed: 2021-12-09.

[34] G. Henderson, "WiringPi." Available at: http://wiringpi.com. Accessed: 2021-12-09.

[35] WaveOptics Ltd, "Modules « WaveOptics." Available at: https://enhancedworld.com/products/modules/. Accessed: 2021-12-09.

[36] Blender Foundation, "blender.org - Home of the Blender project - Free and Open 3D Creation Software." Available at: https://www.blender.org. Accessed: 2021-12-09.

[37] Xsens, "Xsens DOT Wearable Sensor Platform." Available at: https://www.xsens.com/xsens-dot. Accessed: 2021-12-09.

ACRONYMS

API Application Program Interface. 3, 8, 13, 14, 16, 36, 37, 39, 54, 57, 60, 64, 65

CAN Controller Area Network. 6, 7, 29, 30, 33, 64, 65
CLI Command-Line Interface. 16, 61, 64

DDS Data Distribution Service. 12, 13, 16
DPU Data Processing Unit. v, viii, 23–31, 36, 37, 39–45, 47, 49, 51–54, 56, 57, 59–61, 64
DSM-2W Driving Simulator Mockup 2-Wheeler. v, c, d, 3, 4, 32–35, 37

ECU Electronic Control Unit. 33, 34

GUI Graphical User Interface. 7, 16, 19, 25

HMI Human Machine Interface. 32, 34

IT Information Technology. 8

JNI Java Native Interface. 39–41

MOM Message-Oriented Middleware. 8, 10

ORB Object Request Broker. 8
OROCOS Open Robot Control Software. 10
OSC Open Sound Control. 53, 57, 74

QoS Quality of Service. 12, 15, 37

ROS Robot Operating System. 8, 10, 12
ROS2 Robot Operating System 2. v, viii, c, d, 4, 5, 8, 12–16, 36–43, 45, 47, 49, 53, 54, 56, 57, 59–61, 63–65
RPC Remote Procedure Call. 8
RTOS Real-Time Operating System. 16

SWIG Simplified Wrapper and Interface Generator. v, 40, 41

XRCE eXtremely Resource Constrained Environments. 16

A

SILAB CONFIGURATION FILE EXAMPLE

```
SILAB System
{
    WatchdogPeriod = 10000;
};

include system/SYSBase.inc

SILAB Configuration
{
    Computerconfiguration VeraSimulator
    {
        Pool Simulator
    {
      Executable = true;
      Computer MyComputer
      {
        IP = "127.0.0.1";
        Frequency = 60;
        Alias = {ALL};
        Operator = true;
      };
    };
    };

    DPUConfiguration DPUs
    {
        include System\DPUBase.inc

        Pool P27Base : Base
        {
            Executable = true;

            include config\overtake_danger.inc
            include config\intersection_danger.inc
            include config\blindspot_danger.inc
        };

        Pool P27Tests : P27Base
        {
            Executable = true;

            include config\DPUInitConfig.inc
            include config\overtake_danger_publisher.inc
            include config\intersection_danger_publisher.inc
            include config\blindspot_danger_publisher.inc
        };
    };
};

includeif SGE/SGE_700.cfg
include scnx/SCNXSGE_700.cfg
includeif SNDX/SNDX.cfg

SILAB TRF
{
    RoadUserTable Fahrzeuge
  {
    Car =
    {
      Vehicles.V600.Hyundai(blue, 0.1),
      Vehicles.V600.FiatPunto(red, 0.2)
    };

    Truck =
    {
        Vehicles.V600.MAN_TGL(blue, 0.2)
    };
  };

  BehaviourScheme Standard
  {
    Behaviour =
    {
      (Standard)
    };
  };
};

SILAB SCN
{
    include SCNX/SCNX_700.cfg
    include includes\cross_sections\NationalRoadCrossSection.inc
    include includes\cross_sections\MotorwayCrossSection.inc
    include includes\intersection\Intersection.inc
    include includes\national_road\NationalRoad.inc
    include includes\motorway\Motorway.inc
    include parts\Parking.cfg
    include parts\Start.cfg


    # map
    Map Map1
    {
        Intersection Intersection;
        Motorway Motorway;
        NationalRoad NationalRoad;

        Parking parking;
        Start start;

        Connections =
        {
            start.Port1 <-> Intersection.Port1,
            Intersection.Port2 <-> Motorway.Port1,
            Motorway.Port2 <-> NationalRoad.Port1,
            NationalRoad.Port2 <-> parking.Port1
        };

        SetupPoints =
        {
            ("StartPoint", start.Port2),
            ("Intersection", Intersection.Port1),
            ("NationalRoad", NationalRoad.Port1),
            ("Motorway", Motorway.Port1)
        };
    };
};

SILAB MOP
{
  include MOV/MOPModels_600.inc
  include MOV/MOPAnimations.inc

  MOV Params {

    WallDist = 0.5;
    PeopleDist = 0.7;
    StandingPeopleDist = 0.4;
    ExtraVehicleDist = 0.3;
  };
};
```

Listing A.1: SILAB configuration file example.

B

OPEN SOUND CONTROL

Open Sound Control (OSC) is a message-based protocol developed for communication among computers and multimedia devices that works over UDP. The unit transmitted by two different endpoints is called an OSCPacket, in which the sender is the OSCClient and the receiver the OSCServer. An OSCPacket consists of its content, a block of binary data, and its size. The content of an OSCPacket must be either an OSCMessage or an OSCBundle.

For this case, the OSCClients only send packages with OSCMessages. An OSCMessage has three different parts, namely, the OSCAddressPattern, followed by the OSCTypeTag, followed by the OSCArguments. The OSCAddressPattern is a simple string that starts with a '/' character. An OSCTypeTag is a string that starts with a comma and is followed by a character that represents the data type of the OSCArguments, which can be i, f, or s, which match an integer, a float, or a string data type, respectively. Finally, the OSCArguments are a contiguous sequence of binary data, in which the number of arguments in this part of the message must be equal to the OSCTypeTag.

Regarding an OSCServer, they must specify several OSCMethods. These OSCMethods are invoked by OSCClients through the OSCAddressPattern. Invoking OSCMethods is identical to a procedure call, in which the OSCClients call some specific method with a set of arguments.

The OSCMethods of an OSCServer is arranged in a tree structure, in which the leaves are the OSCMethods and the branches OSCContainers. This tree structure is called an OSCAddressSpace. The OSCAddress of an OSCMethod is the full path to the OSCMethod in the OSCAddressSpace, starting from the tree's root. An OSCMethod's OSCAddressPattern begins with a '/' followed by the names of all the containers, in order, along the path from the root of the tree to the OSCMethod, separated by '/' characters, followed by the name of the OSCMethod.

When an OSCServer receives an OSCMessage, it invokes the proper OSCMethods in its OSCAddressSpace. These OSCMethods must match the OSCAddressPattern of the OSCMessages, in which all the matching OSCMethods are invoked with the same OSCArguments. This process is called Dispatching. The parts of an OSCAddressPattern are the strings between '/' characters and the string after the last '/' character. An OSCMessage is dispatched to an OSCMethod if the number of parts is equal and each part matches. The OSCAddressPattern can have several special symbols to form regular expressions to invoke several methods with a single call to the OSCServer.

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 039334; Funding Reference: POCI-01-0247-FEDER-039334].