D5.4 FINAL VERSION OF CPSOSAWARE INTEGRATED PLATFORM

Authors	Pavlos Kosmides, Christina Michailidou (CTL), Konstantina Papachristopoulou (8Bells), Antonio Alvarez Romero, Miguel Martin Perez (ATOS), Pekka Jääskeläinen (TAU), Gerasimos Arvanitis (UPAT), Jordi Casademont, Javier Fernandez (i2CAT), UoP, Nikos Piperigkos, Christos Anagnostopoulos, Aris Lalos (ISI), Wojciech Jaworski (RTC), (IBM)
Work Package	WP5 – CPSoSaware Integration and Cross-layer Optimization supporting design-operation continuum

Abstract

This document presents the updated and final report on CPSoSaware components and their integrations with respect to the demonstrator scenarios of the use cases. It presents how each component serves the purpose of the demonstrator along with the required interactions with other components in terms of integrations interfaces, APIs and data structures that are exchanged.

Funded by the Horizon 2020 Framework Programme

of the European Union

Deliverable Information

Work Package	WP5 CPSoSAWARE Integration and Cross-layer Optimization supporting design-operation continuum
Task	T5.2 Integration, Cross-level Optimizations for CPSoS Maintenance and CPSoS lifecycle Design Operation Continuum
Deliverable title	D5.4 Final Version of CPSoSAWARE Integrated Platform
Dissemination Level	PU
Status	F: Final
Version Number	1.00
Due date	M36

Project Information

Project start and duration	01/01/2020 – 31/12/2022, 36 months
Project Coordinator	Industrial Systems Institute, ATHENA Research and Innovation Center 26504, Rio-Patras, Greece
Partners	1. ATHINA-EREVNITIKO KENTRO KAINOTOMIAS STIS TECHNOLOGIES TIS PLIROFORIAS, TON EPIKOINONION KAI TIS GNOSIS (ISI) the Coordinator 2. FUNDACIO PRIVADA I2CAT, INTERNET I INNOVACIO DIGITAL A CATALUNYA (I2CAT), 3. IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD (IBM ISRAEL 4. ATOS SPAIN SA (ATOS), 5. PANASONIC AUTOMOTIVE SYSTEMS EUROPE GMBH (PASEU) 6. EIGHT BELLS LTD (8BELLS) 7. UNIVERSITA DELLA SVIZZERA ITALIANA (USI), 8. TAMPEREEN KORKEAKOULUSAATIO SR (TAU) 9. UNIVERSITY OF PELOPONNESE (UoP) 10. CATALINK LIMITED (CATALINK) 11. ROBOTEC.AI SPOLKA Z OGRANICZONA ODPOWIEDZIALNOSCIA (RTC) 12. CENTRO RICERCHE FIAT SCPA (CRF) 13. PANEPISTIMIO PATRON (UPAT)
Website	www.CPSoSAWARE.eu

Control Sheet

VERSION	DATE	SUMMARY OF CHANGES	AUTHOR
0.1	01/11/2022	Structure	UOP
0.5	15/12/2022	First Round of Contributions	All
0.8	24/12/2022	Final Round of Contributions	All
0.9	12/01/2023	Review of Del. Completed	IBM, I2CAT
1.0	18/01/2023	Final version of the Del.	UoP

	NAME
Prepared by	UOP
Reviewed by	IBM, I2CAT
Authorised by	ISI

DATE	RECIPIENT
12/01/2023	Project Consotium
19/01/2023	European Commission

Table of contents

Figures

Tables

Definitions and acronyms

Acronym / Term	Definition
AR	Augmented Reality
ASIP	Application-specific Instruction-set Processor
CL	Cooperative Localization
CNN	Convolutional Neural Network
CPS	Cyber-Physical System
CPHS	Cyber-Physical-Human System
CPSoS	Cyber-Physical System of Systems
CSV	Comma-separated Values
CSAIE	Cognitive System AI Engine
DCNN	Deep Convolutional Neural Network
DMS	Driver Monitoring System
DoF	Depth of Field
FPGA	Field-Programmable Gate Array
HLS	High-Level Synthesis
HW	Hardware
LiDAR	Light Detection And Ranging
MRE	Modelling and Redesign Engine
MQTT	Message Queuing Telemetry Transport
OWL	Web Ontology Language
OSM	Operator State Monitoring
PoCL	Portable Computing Language
RDF	Resource Description Framework
RDMA	Remote Direct Memory Access
RTL	Register Transfer Level
SAT	Storage And Transformation
SCM	Source Code Management

SHACL	Shapes Constraint Language
SRMM	Security Runtime Monitoring and Management
SW	Software
TCE	TTA-Based Co-design Environment
UML	Unified Modeling Language
XR	Extended Reality
XRT	Xilinx Run-time

1 Introduction

Cyber-Physical Systems (CPSs) are designed using a model-based design approach, thus accurate modelling and simulation plays an important role in the design outcome. This approach is similar for Cyber-Physical System of Systems (CPSoS) although we must consider the fact that CPSoS have a continuous evolution that involve the continuous addition, removal, and modification of hardware and software CPS components over the CPSoS complete life cycle. This poses a considerable CPSoS challenge since the CPSoS design phase and operation phase are not separated but rather coexist through time thus forming a design operation continuum that must be supported. This continuum leads to a need for System-wide dynamic reconfigurability and adaptability of CPS resources and CPS process lifecycles. The CPSoS must include a mechanism able to reconfigure its CPS components according to its evolving physical and cyber environment, possibly commission new components or decommission/replace old ones. However, the complexity and autonomy of the CPSoS makes it very hard to identify when a reconfiguration is needed, thus highlighting the need for introducing CPSoS self-awareness through a CPSoS cognitive mechanism. The cognitive CPSoS must be able to provide situational awareness in a decentralized manner (matching the decentralized way CPS operate within the system) and aid both CPSoS operators and users in order to reduce the complexity management burden. CPSoSAWARE architecture as already presented thoroughly in the respective deliverable D1.4 [1], delivers these requirements through the tight integration of various components that operates in the CPSoS System Layer and CPS/CPHS Layer.

In essence, CPSoSAWARE project's main objectives are to deliver an architecture formed in a modular way able to adapt to various scenarios and apply to all layers of data flows, from CPS and edge layer to cloud system layer. In the context of the project, several components have been designed and implemented to address challenges that are presented in two very demanding application domains, the automotive and manufacturing. The evaluation of the CPSoSAWARE architecture will be performed through respective demonstration scenarios as described thoroughly in D6.5 [2]. Since the operational environment of these components span to all layers of the architecture, it is expected highly heterogeneity in terms of interfaces and data structures introducing significant integration challenges.

Deliverable D5.4 is the final version of a series of 2 deliverables that describe the integration activities performed in T5.2. T5.2 focuses on the integrations and cross level optimizations for CPSoS Maintenance and CPSoS lifecycle design operation continuum. In this task, the various CPSoSAWARE blocks that provide support for the CPSoS Design Operation Continuum are integrated. This action performs integration of the Modelling and Redesign Engine (MRE), the Cognitive System AI Engine (CSAIE), the (Security Runtime Monitoring and Management) SRMM and the Storage And Transformation (SAT) blocks of the CPSo-SAWARE System Layer with the Cyber-Physical-Human System (CPHS) layer and the actual generation of test vectors to be used for the validation of the Design Operation Continuum support mechanism. In this task, the evaluation process that is going to be conducted on the tow use cases in WP6 will be used as feedback in order to provide optimization to the cross – layer integrated components. The evaluation process is meant to highlight possible Requirements KPI misalignments due to integration of the various CPSoSAWARE blocks and components and provide possible solutions to mitigate the risk. The cross-layer communication will be optimized in order to support the required Key Performance Indicators (KPI) , thus

focusing on providing fast response time and small communication latency. The evaluation process will also be extended to the level of provided security in the Design Operation Continuum Support Mechanism. The task is associated with all WP5 and WP4 tasks and the evaluation process of WP6.

This deliverable presents the final integration designs and implementations of the CPSoSAWARE components with regards to the two CPSoSAWARE application domain pillars and their demonstrators.

Section 2 summarizes the architecture of the CPSoSAWARE project and presents the main interactions between the components along with their interfaces. Section 3 presents the approach followed to automate and support the software integration development. Section 4 presents the core technology components that were developed in CPSoSAWARE and can be applied to both use cases. Finally, Section 0 is dedicated on presenting the components integrations that will serve the demonstrations for each use case.

2 Architecture

CPSoSAWARE system, as of the latest version of the system architecture, consists of 3 main layers. 1)CPSoS System Layer, 2) CPS/CPHS Layer, 3) Simulation and Training Layer (Figure 1). The distribution of the technical components of these layers is presented from Figure 2 to Figure 4. T5.2 is strongly related to T1.3 where the dependencies, interactions and finally interfaces of the various components have been detected. These interactions are depicted in Figure 5. More insights and details on the description/specifications of the system architecture and components is given in "D1.4: Second Version of CPSoSAWARE System Architecture" from which these figures where excerpted [1].

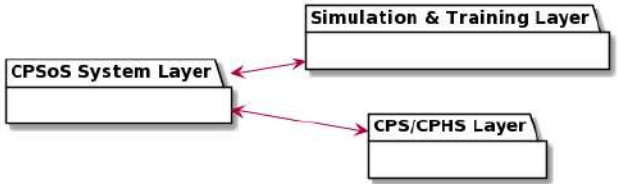


Figure 1: CPSoSAWARE Layers

Moreover, the outcome of these integration activities as performed in T5.2 and reported in the two respective deliverables will be realized in WP6 for the execution of the pilots. During the initial phases of the CPSoSAWARE project, two use cases have been defined and described in detail. In this definition phase, the use cases are outlined, and the main components have been identified. These developments are to be integrated on the two pilot demonstrators and tested/validated in specific testing scenarios as reported in D6.4 [3].

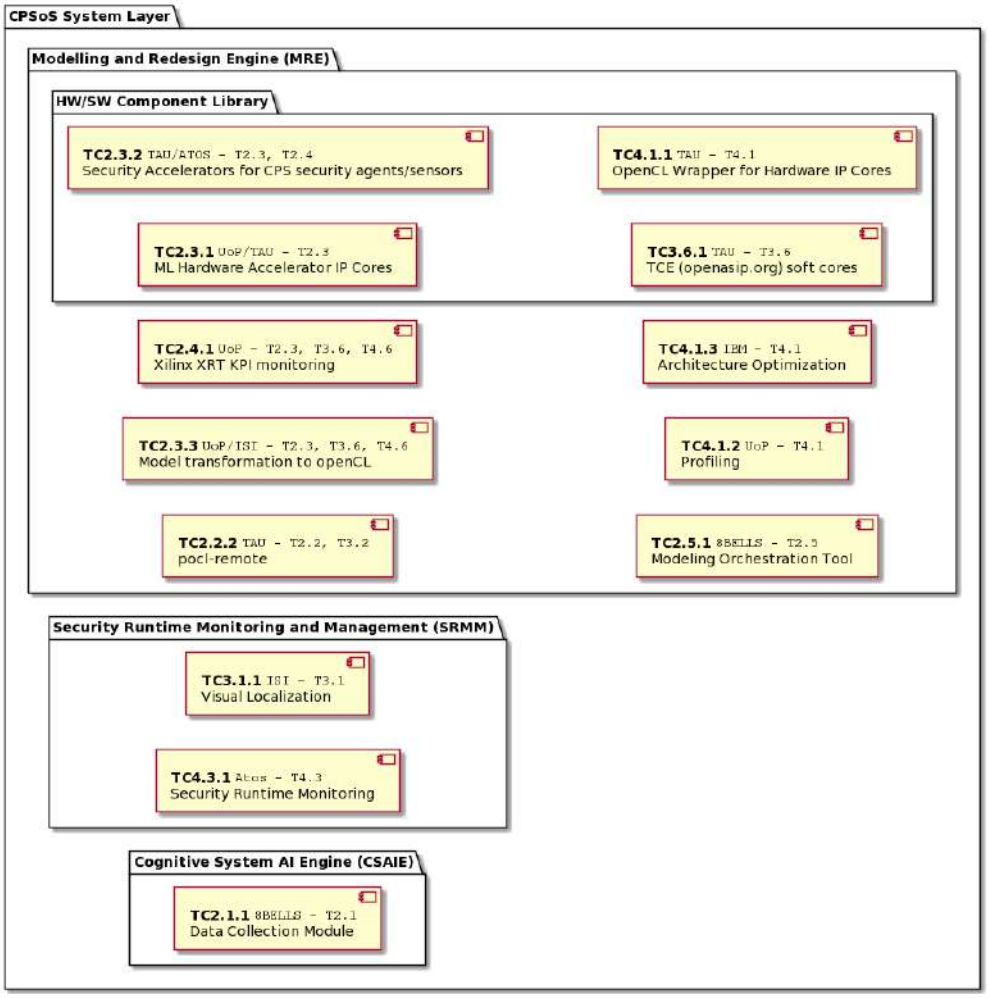


Figure 2: CPSoS layer and sub-blocks

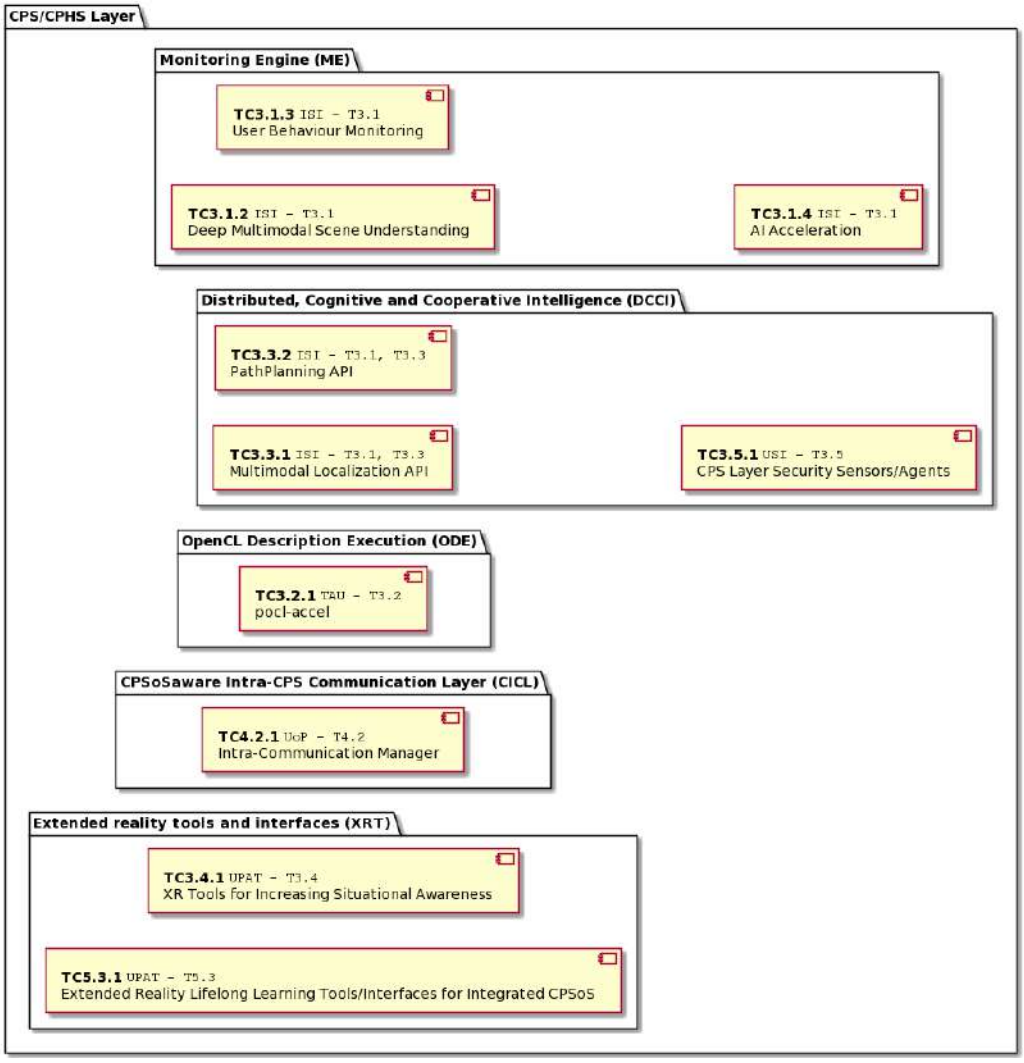


Figure 3: CPS/CPHS layer and sub-blocks

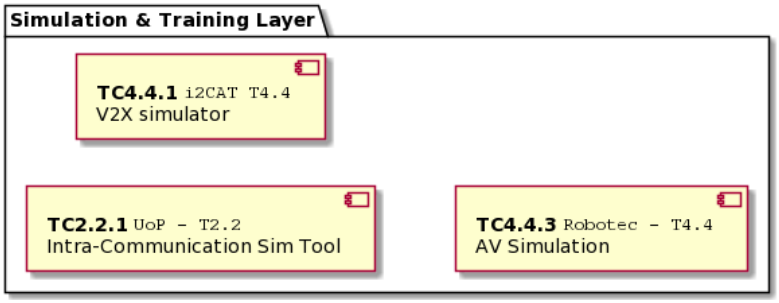


Figure 4: Simulation and Training layer and sub-blocks

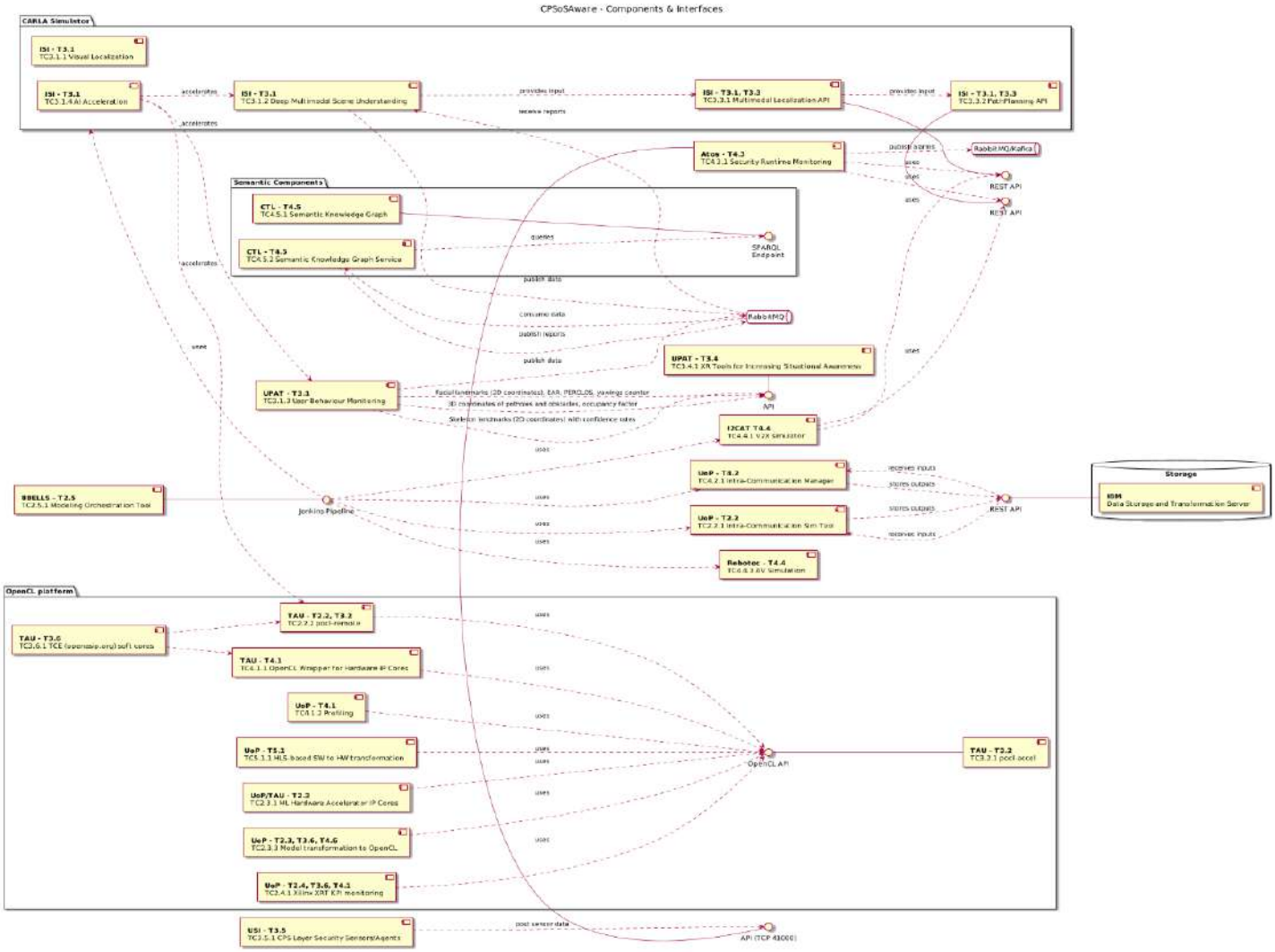


Figure 5: Overview of system interfaces

3 Integration & Deployment Framework

To facilitate a more formal and automated way to perform integration testing and deployment, CPSoSAWARE adopted the use of Continuous Integration / Continuous Deployment automation servers. In CPSoSAWARE, automations on integration testing where these are applicable, are based on Jenkins [4], an open source and free software that implements an automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous deployment. It is a server-based system that runs in servlet containers such as Apache Tomcat and it supports several version control tools (e.g. CVS [5], Subversion [6], Git [7], Mercurial [8], etc.) and can execute various build tools commands as well as arbitrary shell scripts and Windows batch commands.

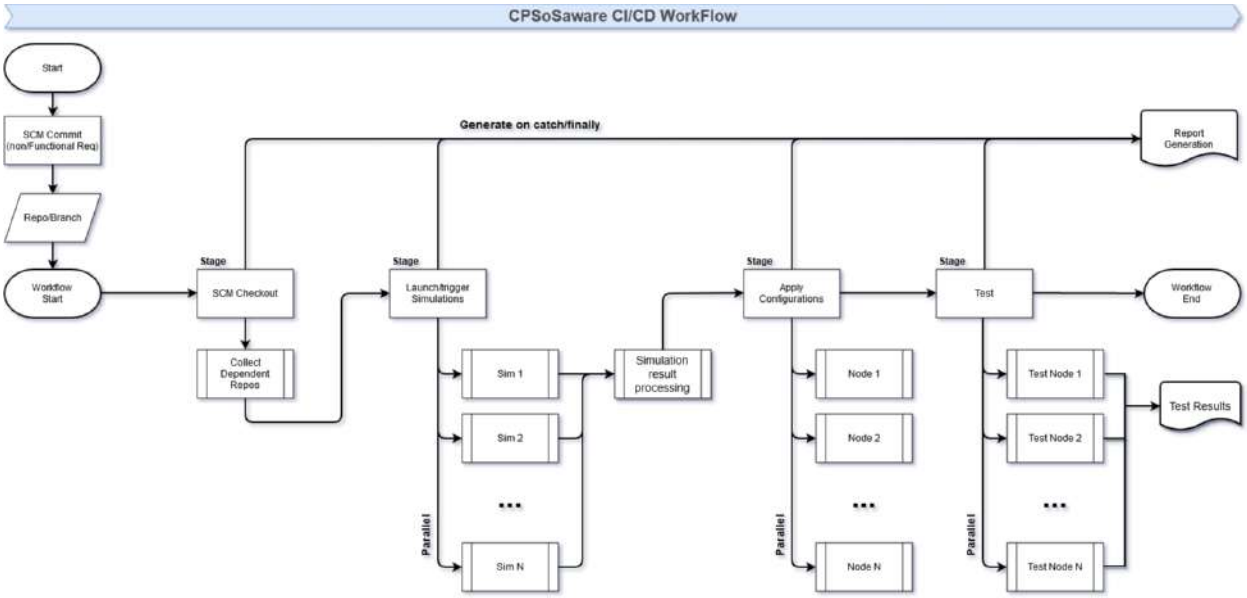


Figure 6: CPSoSAWARE CI/CD workflow

The workflow proposed in the CPSoSAWARE project is presented in Figure 6. This workflow is designed based on Jenkins Pipelines [9] and there will be configured with a source code management (SCM) polling trigger.

The SCM system adopted by the CPSoSAWARE is Git. Git is a distributed version-control system for tracking changes in any set of files, originally designed for coordinating work among programmers cooperating on source code during software development. Its design goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems).

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. A continuous delivery (CD) pipeline is an automated expression of your process for getting software from version control right through to the users. Every change to the software (committed

in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline domain-specific language (DSL) syntax. The definition of a Jenkins Pipeline is written into a text file (called a Jenkinsfile) which in turn can be committed to a project's source control repository. This is the foundation of "Pipeline-as-code"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

As already imposed, all the involved components in the CPSoSAWARE platform will be version controlled and stored in Git Repositories. These components will be:

- Functional/non-Functional requirements
- Simulation suite code
- Components configurations (raspberry, FPGA, etc.)
- Components codes:
    - Bitstreams codes
    - Service codes
    - Scripts
- Test automation scripts: The testing scripts will verify that the configurations are applied/deployed successfully in the components and there is communication between them.

Also, a binary repository manager (also known as artifactory) will be configured to store 3rd party libraries and/or the outcome of the build process. This repository will store binaries such as:

- Customized OS images
- FPGA bitstreams
- Simulation suite binaries

It must be noted, CPSoSAWARE components present a heterogeneity that does not allow in the context of the project, to configure pipelines where end – to – end workflows will be able to be automated through the CI/CD automation server. However, individual integration paths have been already tested through Jenkins pipelines while automated deployment/commissioning tasks are to be executed by Jenkins delivering the required functionality of the TC4.61 as described in D1.4 [1]. The details of the automation server maintained from UOP along with the Storage And Transformation (SAT) engine developed from IBM that is used for persisting configuration data and evaluation results, are detailed in more details in D4.4 [10].

4 Core Technology Component

Core technology components regard CPSoSaware components that were designed and developed without being restricted to the two CPSoSaware use cases. Thus, they are demonstrated in separate scenarios that serve as partial demonstrators for technologies that could be used in similar purposes in future implementation cases and are not tailored to a specific use case.

4.1 Distributed and Reliable Edge Execution Environment Technology Demos

In this section we describe technology demonstrators and baseline measurements of the CPSoSAWARE components related to the distributed and reliable execution environment developed in the project. The components were developed mostly within WP2 and WP3 contexts with the final optimization, technology integration demonstration work and documenting done with the Task 5.2's scope. These demonstrators were done separately from the final demonstrators. The following components were tested in the described tech demonstrators.

Pocl-remote (TC2.2.2): Scalable distributed OpenCL runtime layer with P2P event synchronisation capabilities.

ML Hardware Accelerator IP Cores (TC2.3.1): FPGA-based IP core components (interfaces) focused on Machine Learning / Deep Neuronal Networks (ML/DNN) computations. The IP cores are seamlessly integrated in the Portable Computing Language (PoCL) based OpenCL run-time system.

PoCL-accel (TC3.2.1): A Generic OpenCL driver for PoCL to interface with custom devices (hardware accelerators) from the OpenCL API.

OpenASIP (TCE, openasip.org) Soft Cores (TC3.6.1): Customised processors designed using TTA-Based Co-design Environment (TCE), an ope-source application-specific instruction set toolset based on the Transport-Triggered Architecture (TTA). Various hardening features can be added via replication of functionality and special instructions.

OpenCL Wrapper for Hardware IP Cores (TC4.1.1): OpenCL kernel description interface to associate Hardware IP cores with the OpenCL models.

Profiling (TC4.1.2): Profiling for a highly heterogeneous platform consisting of multicore ARM processor, ASIP processors as well as FPGA fixed logic IP. FPGA logic is a "morphable" computation resource without predefined computational capabilities. All software nodes will be handled by PoCL remotely enabling dynamic remapping and re-scheduling opportunities.

In addition to these components, initial support for OpenVX programming was added to expand the work started in WP3 with a domain specific programming layer, as planned. Also automated command queue redundancy support via an FPGA-based voting mechanism which was started in WP3 was integrated to functional demonstrators as described in the following text. Finally, a large part of the last months in the

project of TAU was spent on optimization different aspects of the execution environment. This was done, for example, by participating in the standardization of OpenCL command buffer mechanism [11] in the Khronos group to which TAU provided an example implementation of in the PoCL framework in addition to various comments that helped shaped up the specification. The command buffering, after taking fully into use is expected to dramatically lower the latency of remote offloaded task graphs in repeated execution.

There was also work on producing a pruned version of the PoCL-R client, named Nano-PoCL, which is a runtime library meant to be serving a host program in low-end near-sensor devices such as distributed camera processors in an intelligent car. In this case, we demonstrated the work by using a nano-drone which has limited processing capabilities on board based on a SoC with a RISC-V CPU.

4.1.1 Portable Hardware Acceleration Abstraction

The built-in kernel mechanism that was initially reported in D3.2 [12] was developed further in the context of this deliverable. To demonstrate the easy integration of the proposed built-in kernel registry abstraction to higher level programming models, a proof-of-concept OpenVX implementation was created on top of the built-in kernel abstraction layer.

The OpenVX implementation lowers the program consisting of OpenVX nodes to an OpenCL task graph composed of commands submitted to one or more command queues, which are then passed on to the OpenCL implementation for execution. The OpenCL task graph can consist of both built-in and software kernels, depending on what built-in kernels the device supports. The principle behind this lowering is shown in Figure 7.

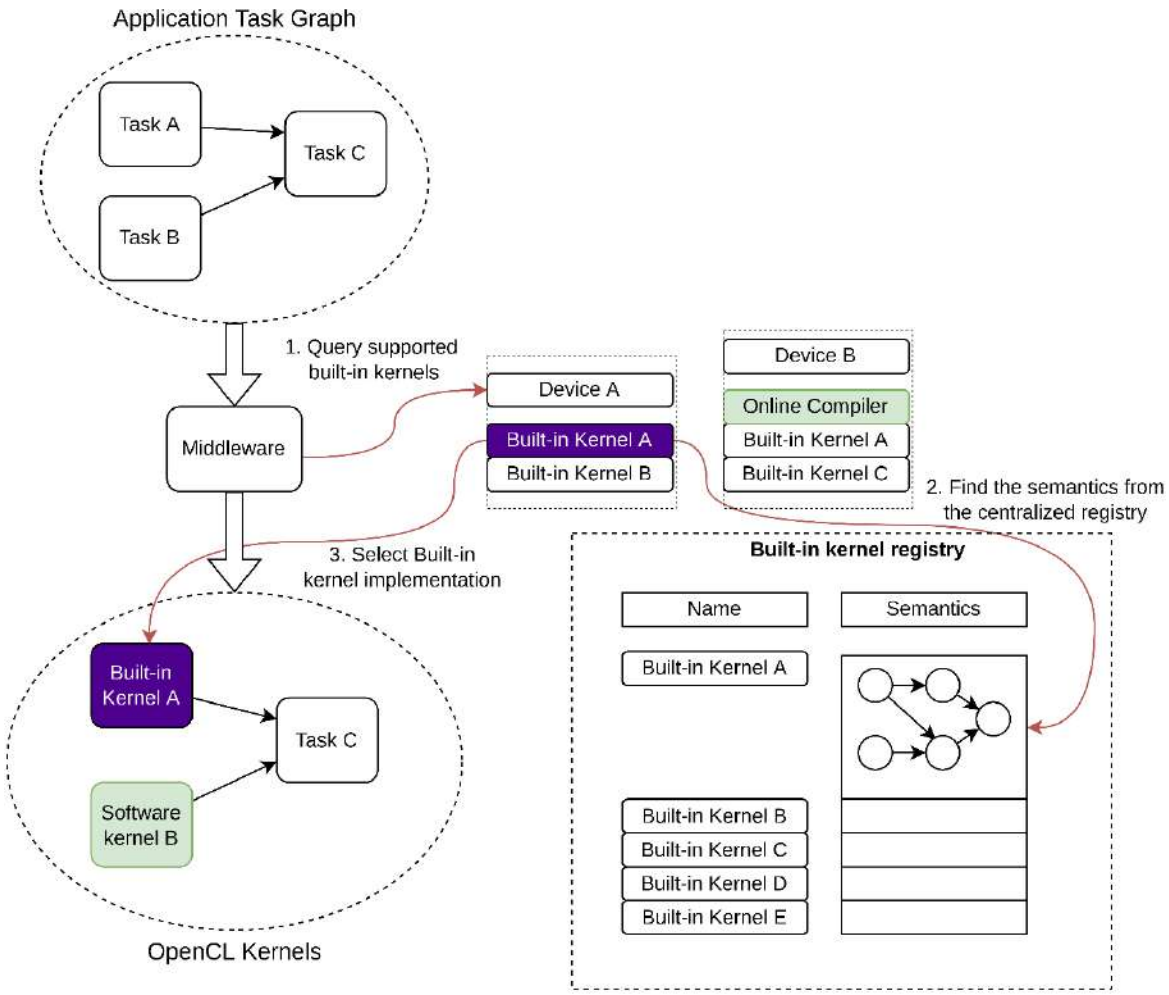


Figure 7: Task graph lowering by OpenVX. The OpenVX implementation queries the available devices for built-in kernel implementations and lowers the task graph to an OpenCL task graph described with kernels and event dependencies.

The built-in kernel abstraction makes it possible to have specialized versions of OpenVX specified nodes, which allows for more specialization in the kernel, which can be utilized to increase the efficiency. The specialization parameters are hardened in the built-in kernel specification. It is possible to add support for other OpenVX datatypes and specializations inside the OpenVX implementation at the point when the node is being lowered, and the built-in kernel is being chosen.

The same OpenCL platform can also be simultaneously targeted by other higher-level frameworks. To demonstrate this, an example Python application was created, which uses both OpenVX and TVM APIs which internally both utilize a common OpenCL context. This demonstrator is further described in a publication [13].

The application chosen for this technology demonstrator was an object detection network YOLOv3-tiny. The network finds bounding boxes for objects in an image and classifies them to belong to a specific category. The complete demonstration application also included an image preprocessing step to show how the proposed abstraction enables efficiently adding pre- and postprocessing kernels to neural network-based applications.

The pre-processing step was created as a OpenVX program. A minimal OpenVX implementation was created that implements the API calls needed in the demonstrator. For this demonstrator, two new built-in kernels for image pre-processing were added to the built-in kernel registry. The semantics of the built-in kernels were chosen to be specialized versions of the OpenVX nodes.

TVM was used for the YOLOv3-tiny's neural network operations. There the convolution operations were replaced with the built-in kernel **pocl.dnn.conv.2d.nchw.f32**. The built-in kernel abstraction enabled the backend to convert the built-in kernel functions into calls to optimized DNN acceleration library cuDNN.

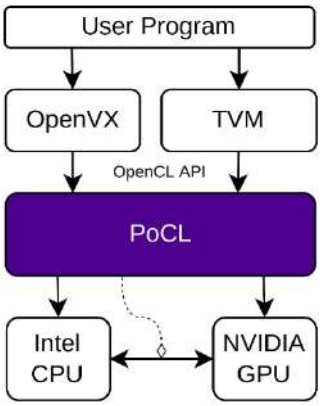


Figure 8: Object detection demonstration setup. The user program defines the application in standard OpenVX and TVM library calls, which internally share the OpenCL context with each other.

The built-in kernel abstraction layer and its connection to the AlmaIF FPGA abstraction that was started in WP2 and initially reported in D2.3 [14] is published as open-source[1] to create a common vendor-independent portability layer for both software and hardware developers.

[1] http://code.portablecl.org/

4.1.2 The Demonstrator Application

The demonstrator application was chosen to be a representative workload that utilizes algorithms in classical computer vision. For this, we adapted a Khronos sample project using the OpenVX API[2]. The sample project demonstrates an OpenVX pipeline for detecting pixels that are potentially human skin and one that implements the Canny edge detection algorithm. These are then combined with some application provided OpenVX nodes to create a live application that allows a user to "pop" virtual bubbles with their hand by detecting the hand position from an image or live video feed.

Since support for application-provided nodes is not relevant to our current work, so instead of the bubble popping application, we simply combined the skin tone and edge detector pipelines and displayed their combined results as is.

The skin tone detector splits the image in red, green and blue channels and applies a threshold to each channel separately. The thresholded boolean images are then combined with a series of logic operations to produce a binary estimate of whether each pixel is skin coloured or not.

The edge detector part converts the input image into YUV colour, splits out the luma channel and applies a Canny edge detection filter to it. The single-channel results from both parts are combined into a new RGB image with the binary skin colour mask in the red channel and the edge mask in the red and blue channels.

[2] https://github.com/KhronosGroup/openvx-samples

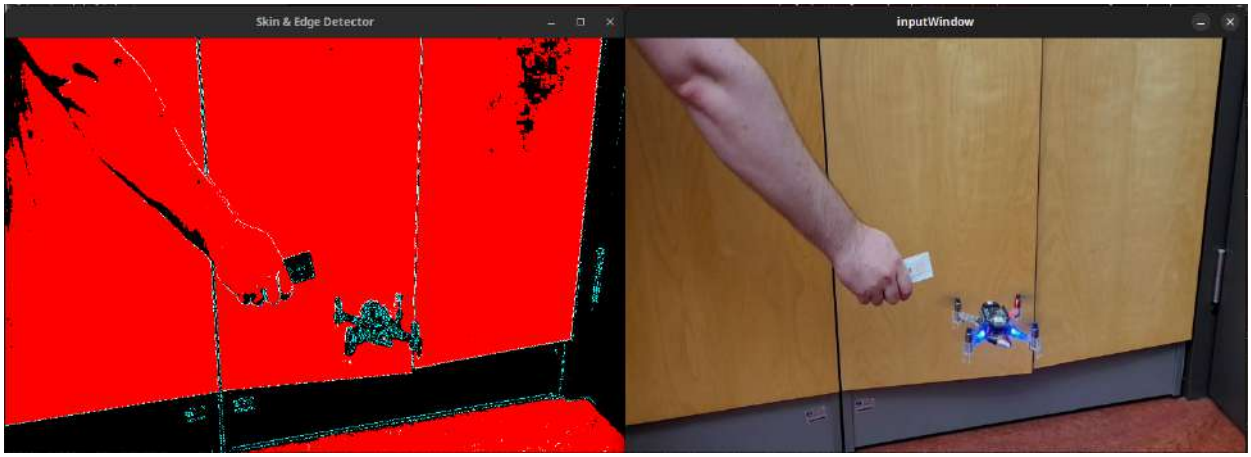


Figure 9: Left: pixels that are detected as skin coloured highlighted in red, edges are highlighted turquoise. Right: original input image.

4.1.3 Latency Lowering with Command Buffering

The latency of the application was improved with the provisional command buffer extension of OpenCL which allows recording command sequences and repeatedly launching them. Commands within a command buffer are synchronized with a mechanism that is separate from the events that is used with commands outside a buffer. This gives implementations more freedom to optimize execution within a buffer. Since command arguments are validated at the time of recording a command to a buffer, such checks can be omitted when enqueuing the buffer to a command queue for execution. For synchronizing with other OpenCL commands, enqueuing a command buffer yields an event representing completion of the recorded command sequence as a whole. Events yielded by prior commands can also be given as dependencies that have to complete before any of the recorded commands can be started.

For the purposes of command buffers, PoCL is split into two parts: the API layer, that provides the OpenCL functions to applications and the backend layer that contains a variety of implementations of the actual computing facilities such as (but not limited to) one built on LLVM and CPU threads, one that translates OpenCL commands to CUDA and one that forwards commands to a different machine (PoCL-Remote). In our proof-of-concept implementation, all command buffer logic is implemented in the API layer, making it usable with any backend without the backends having to even be aware of command buffers.

For the traditional style of immediately enqueuing commands the API layer validates command parameters and constructs a data structure describing the command corresponding to the OpenCL functions that the linked application calls. These structures are then passed on to the backend implementation that corresponds to the OpenCL device associated with the command queue that the command is being enqueued to.

When a command is recorded to a command buffer, the parameters are validated and the same data structure is constructed in the API layer, but without event dependencies. Instead of calling into the backend implementation, however, the command is simply stored in the buffer for later. When the buffer is enqueued, the API layer creates copies of the stored commands, replaces the internal sync points with proper events and then passes the final command structure to the backend implementation.

In the demo application, the whole OpenVX pipeline is recorded to a single command buffer that is enqueued once for each video frame. Launching the recorded buffer exhibits a performance degradation in our overlay implementation compared to enqueuing each recorded separately using the traditional immediate API. The difference becomes more pronounced in a synthetic benchmark that generates frames (I.e. command buffers) with around 1800 commands: the traditional immediate enqueuing API takes around 15ms per frame to process these commands while the command buffer takes a whopping 60ms per frame. This is because of an unoptimized initial implementation, which will be our focus in next projects where the command buffering will be developed further.

Creating backend-specific implementations of the command buffer functionality so that the API layer of the library can request the backend to execute the entire buffer in a single call would likely yield great performance improvements over the overlay implementation.

4.1.4 Multi-Device Edge Scalability for PoCL-R via RDMA

Data center-grade hardware frequently has specialized high-speed network interfaces that support offloading the actual data transfer tasks from the CPU to the network interface itself via Remote Direct Memory Access (RDMA).

With RDMA the CPU can register memory regions for direct memory access from the network interface. These regions can then be used to request an asynchronous data transfer from a given region on one machine to a registered region on a different machine. The registered memory regions can be in RAM or on a device that shares e.g., a PCIe bus with the network interface and can make portions of its memory directly accessible from the bus, such as a GPU. Since the network interface can directly access the source and destination memory regions, the CPU does not have to be involved in the transfers beyond the initial registration and setting up the transfer request. There are different modes of operation, some of which additionally require the responding side to set up a request for the network interface to accept transfers. Requests can also be chained, allowing applications to submit multiple requests to the network interface in a single API call to batch the CPU work efficiently.

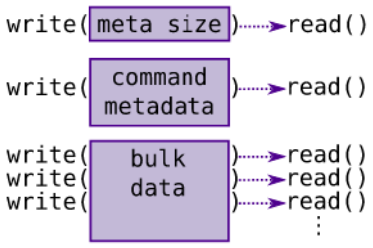


Figure 10: With TCP sockets, individual parts of messages have to be sent separately and long messages may need to be split up into multiple write calls to the networking driver.

Pocl-remote (TC2.2.2) was extended to make use of RDMA when transferring OpenCL buffer contents between remote servers. RDMA being a message-based protocol ended up simplifying the communication logic a fair bit compared to the stream-based TCP implementation. With TCP multiple read and write calls were needed to first transfer the message size and then read from the stream the exact number of bytes needed for the message. With RDMA this is instead handled by the network interface as part of the protocol, avoiding the extra round-trip through application logic and the kernel networking driver. Similarly, with TCP, large writes must be split up into multiple smaller ones. RDMA instead handles the entire transfer with a single request regardless of size.

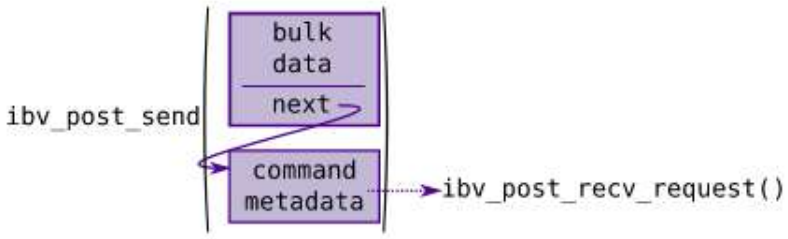


Figure 11: RDMA allows sending arbitrarily long chains of messages in a single call to the driver.

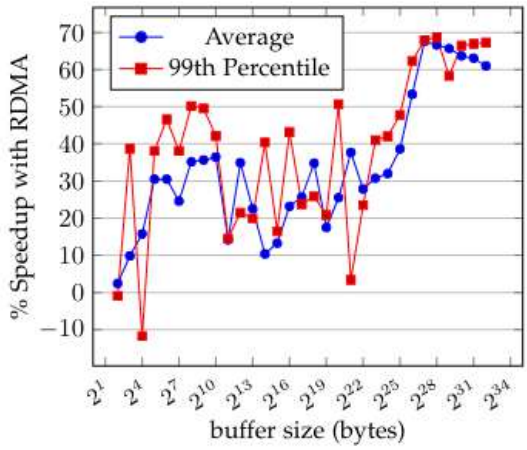


Figure 12: Performance improvement in server-to-server buffer transfers when RDMA is used instead of TCP

In a synthetic benchmark that increments a single number in a buffer before transferring it to a different device on a different machine the move to RDMA proved to be on average 30% faster than the classic TCP code path. With extremely large buffers the difference further increased to over 60% on average.

These speedups were also reflected in a basic matrix multiplication benchmark where each device was assigned a subsection of the matrix to compute, and the partial results were transferred to a single device for combining into the final matrix. As the matrix size was increased, performance increased by around 60% compared to the TCP code path, matching the results of the synthetic transfer benchmark. Notably, the performance increase does not occur if enough separate machines are added that the portion calculated and transferred by a single machine stays below the size where the performance improvement in the synthetic transfer benchmark suddenly climbs from 30% to 60%. Additionally, if the size of the per-machine portion is reduced the overhead of managing memory regions for each peer becomes apparent as a reduction in overall performance.

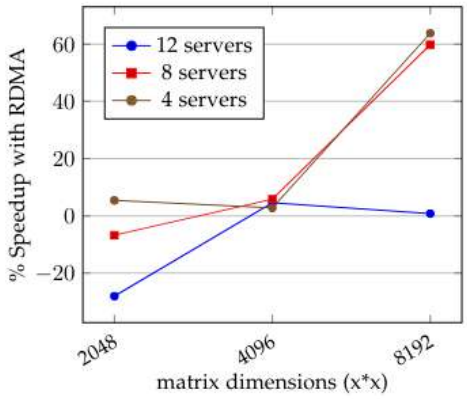


Figure 13: Performance improvement in naïve matrix multiplication when server-to-server buffer transfers are handled with RDMA instead of TCP

4.1.5 Reliability via Redundant Command Queue Execution

The purpose of this demonstration is to show the triple modular redundant voting in a situation where this is useful. In this hypothetical situation, wrong results can lead to damage to the demonstrator and/or its environment. An example could be a robot in a hazardous environment where it is possible due to outside interference that a processing system or communication interface returns the wrong data, causing it to hit objects in the environment. The setup will be a computer vision scenario with edge detection replicated to three devices on a CPU and the voting device on an FPGA. To one of the replicated devices interferences will be introduced, such as values in the input buffers being randomly flipped. The voting device will take this erroneous data and the correct data from the other two devices and return the correct result. To help visualize the correcting behaviour, the input buffers to the voting device will be shown.

4.1.5.1 Voter device

In D3.6 [15], a scenario was demonstrated using a PYNQ-Z1 development board which had multiple HLS devices with the AlmaIFV2 interface doing redundant computation with one device voting on the results. It was noted back then that there was room for improvement and, in this section, it will be shown what has been done to address this.

The previous implementation made use of block RAM (BRAM) to store buffers of each device. This type of RAM is on the FPGA itself and while fast in scenarios where data needs to be read and written to often by the IPs (functional blocks of hardware on a FPGA are often called Intellectual Property), is not fast when transfers need to be done from the host. In the task of voting, this is not the case at all, data is read and written to once, negating any benefits of using BRAM. On top of that, voting generally requires more memory than the computation being executed redundantly (e.g. in a case of TMR, three input buffers and one output buffer), which makes BRAM, with its limited size problematic.

To address this, a solution was implemented that makes use of the PYNQ-Z1's DDR3 RAM. This RAM consists of 512 MB of which 128 MB is dedicated for contiguous memory allocation (CMA) by default[3]. This is a huge improvement compared to BRAM which is 630 KB in total. DMA controller IPs are used to access this memory in a streaming fashion. This set up would also work.

Some simple tests were done to determine how much data could be moved in this way. Using the PYNQ python environment with two DMA controllers reading and writing in a FIFO manner, the average throughput was 946 MB/s. To compare the performance from the CPU to this, the STREAM[4] benchmark was also run. This benchmark was compiled with openMP enabled and the benchmark where a value from one array is copied to another array reached a maximum speed of 1026 MB/s and 604 MB/s without openMP. It is possible to bring the performance of the DMA controllers closer to what was achieved with the CPU benchmark by increasing the clock speeds. The benchmark was run at 100 MHz, the default clock speed on the Z1, but it is possible to generate bitstreams up to 140 MHz for the subsection that writes to RAM of the DMA controller, and it is possible to go even higher for the reading subsection. These results should be taken with a grain of salt since the max frequencies are also dependent on the physical placement which in turn is dependent on the other IPs in the design. Nevertheless, these results lead us to conclude that it is possible to get high performance from the DDR RAM with the FPGA.

Another design choice different from the previous implementation was to make the HLS design more modular. And while useful in demonstrating the technology, the previous design left performance on the table. Previously, the entire AlmaIF core was one IP block, which would handle all aspects from communication with the host to executing the required built-in kernels. This made the core rather complex, and the HLS tools were not able to pipeline the actual execution of the kernels. Another downside to this approach was that all aspects would be tied to the same clock, capping it to the slowest part of the entire design.

In this implementation, the design consists of multiple IP blocks: a control IP and an IP for every different built-in kernel, an example of this can be seen in Figure 14.

[3] https://pynq.readthedocs.io/en/v2.0/_modules/pynq/xlnk.html

[4] https://pynq.readthedocs.io/en/v2.0/_modules/pynq/xlnk.html

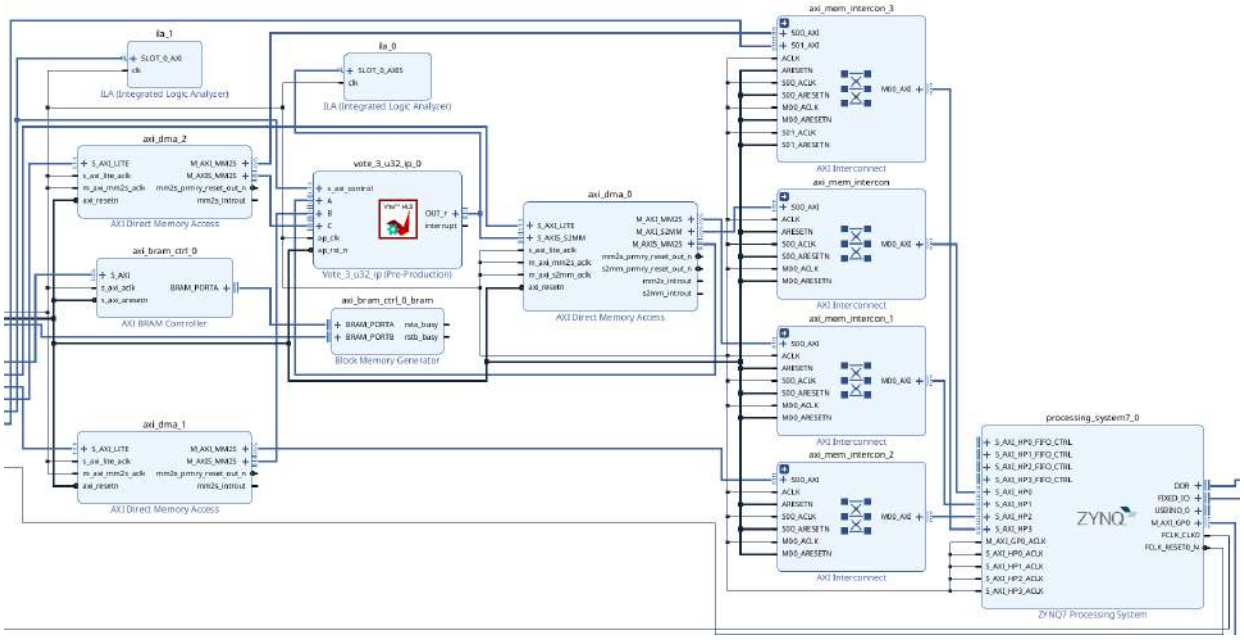


Figure 14: Block diagram of different IPs. In the center is IP that provides the voter kernel functionality. Not shown is the control IP

The control IP handles communication with the host, tells the DMA controllers which data to read and write to and controls the voting IP. Communication with the other IPs is done over AXI and pipelining is not required. This leaves the voting IP to be simplified and specialized in voting. In FPGA design it is recommended to turn the work dimensions into one dimension [5]. And this fits well with the contiguous access patterns of the DMA controllers and allows the HLS tools to pipeline the operations. The result of this pipelining is that the IP can vote every clock cycle with a delay of 3 cycles. This is shown in Figure 15And since the clocks are now separate, it is possible run the IPs at different clocks. The Vitis HLS CLI estimates that the control IP can run at a maximum of approximately 130 MHz while the voting IP can run at just over 1000 MHz, although at that point it will be capped by the maximum frequency that the FPGA can generate.

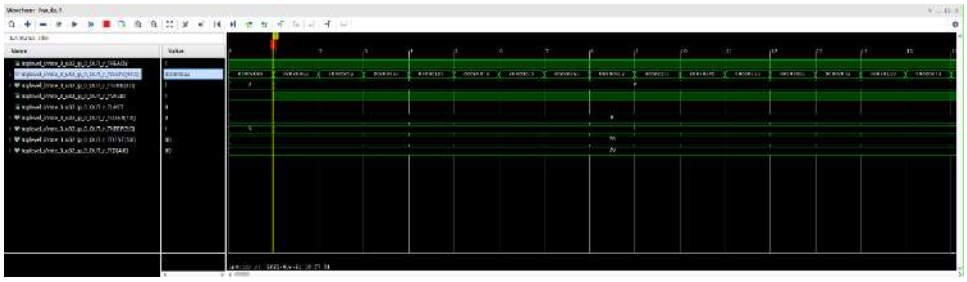


Figure 15: Example of the pipelining in action, the voter IP can output results every cycle

There are several numbers that can be used to bring the new performance into perspective. The voting kernel works on packets of 32 bits and is running on a clock of 100 MHz, that puts the maximum throughput at 381.5 MiB per second. To keep the IP fed, 1144.4 MiB is read every second and 381.5 MiB is written. That puts the total data transferred every second at 1525 MiB. This is far above the transfer speeds achieved in the STREAM benchmark.

Unfortunately, much of this performance is lost due to driver overhead. Looking at the traces of execution, on average the time spent on the kernel is 0.297 milliseconds for an input size of 1024, this puts the throughput on 13.2 MiB per second. However, this is still much faster than the previous implementation which had an estimated throughput of 4.9 MiB per second on an input of 128. It becomes clear that this new implementation is faster than the previous implementation.

One might argue that this implementation is too tailored to the PYNQ-Z1 by making use of the RAM also used by the ARM CPU. But fortunately, it is not uncommon to see large DDR RAM banks on larger (PCIe) FPGAs, such as for example the Xilinx UltraScale and Alveo series. And something similar can be done as what Ruan et al. [16] did and prefetch data from the host to the FPGA RAM before it is needed by the voter device. Another possible solution to the limited memory of the FPGA that is often seen in other OpenCL libraries would be to make use of OpenCL pipes. But since it would require the programmer to rewrite the kernels to use pipes, which reduces the portability, this was left out for now.

4.1.5.2 AlmaIFv2 HLS

In order for Programmers to quickly start implementing their own custom built-in kernels with the AlmaIFv2 interface, an example HLS template was created. The High Level Synthesis (HLS) allows one to write IP functionality in the more familiar C language and have it compiled to a Hardware Description Language (HDL) like Verilog or VHDL. The template contains source code and scripts to generate a working bitstream that can be programmed onto the FPGA.

Included with the template is also a testbench file written in C. This file can be used in two different forms of testing. The form is a C simulation: the code is compiled with a conventional compiler and the results of executing the functions describing the IP blocks are checked. This is quick and can be done without using HLS tools such as Vitis HLS. Many logical mistakes will manifest in such a simulation. The other form

is a behavioral simulation in which the C code is compiled to a Register Transfer Level (RTL) level and the results are verified. This is a more accurate simulation, but it takes longer. Being able to catch these bugs early on is beneficial for productivity since there is a considerable amount of time spent compiling and synthesizing before the code written is deployed on to the FPGA.

Included with the template is also a CMake file which can be used by many C IDEs to configure a build environment. This allows programmers to write code in a program they are more familiar with, lowering the barrier even more. On top of this, they can easily use their IDE's debugging environment to solve bugs encountered. The Vitis HLS IDE does also provide the possibility to insert breakpoints, however for unknown reasons, when using Vitis 2022.1.2, these were not being hit for the template.

4.1.5.3 Erroneous device behavior simulation

Previously in D3.6, an API function in PoCL was shown that could allow one to run OpenCL kernels in a redundant way to improve reliability. The report of D3.6 mainly focused on the overhead and since then, work has gone into also demonstrating the correctness of the implementation. In this section, a demonstrator is described, and the results are shown.

Under normal circumstances, the failure of a computational device does not happen often. The exact Mean Time Between Failure (MTBF) varies widely from one system to another but in demonstrative purposes is often on the scale of decades [17]. To speed things up, a system can be exposed to radiation, however this could potentially permanently damage the system. Therefore, a less permanent demonstration was devised.

By inserting a second kernel to the queue of a redundant device, erroneous behavior can be simulated. This kernel takes the output buffer of the kernel a programmer wants to be run redundantly and randomly changes bits before then passing it on to the voting kernel. This implementation has a number of benefits. First off, since the kernel is used to simulate the behavior, this can be used to simulate any erroneous device supported by PoCL. Secondly, the error rate can be tuned to a desirable percentage of errors. This can be useful if the input size is small and therefore the chance of an error manifesting is smaller. Thirdly, there is no need to modify the redundant kernels in order to simulate any faults. Going from normal usage to a simulation can be done by setting a compiler flag.

As a demonstration, an example setup has been made which highlights the results of executing kernels redundantly with one device being faulty. In this demonstration, a sobel edge detection kernel is applied to an 800 by 600 pixel gray-scale image (Figure 16). The faulty device will randomly set black pixels to white as can be seen in below (Figure 17).

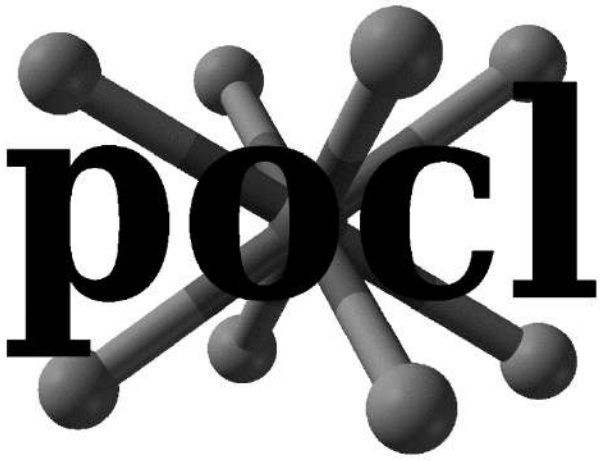


Figure 16: input gray-scale image for the faulty device demonstrator

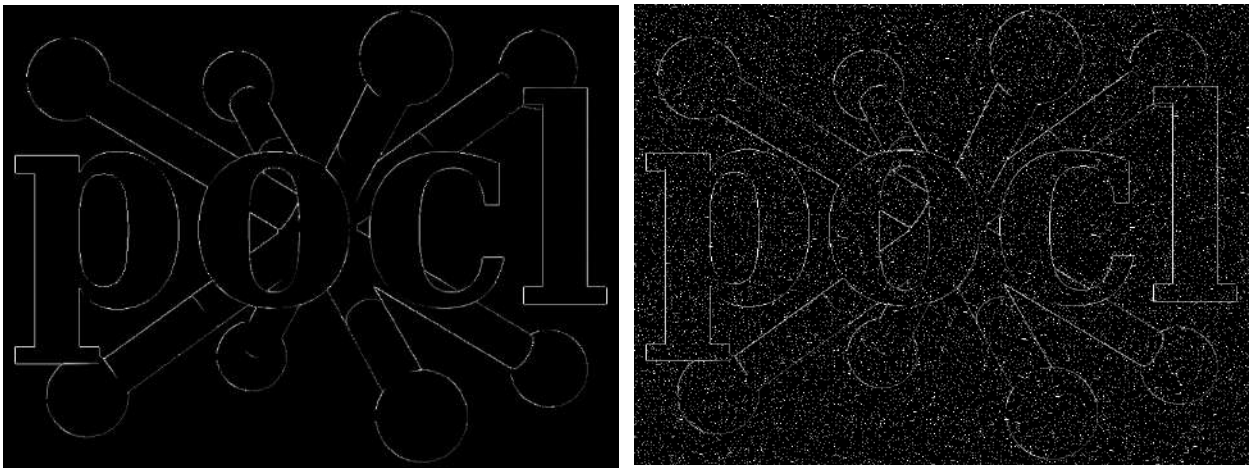


Figure 17: corrected resulting image.

Looking at the normally corrected image (Figure 17), one can see that the white pixels are regions of interest and that there are relatively fewer white than black pixels. Under normal circumstances, subsequent image processing could further propagate this error and make the result unusable. However, the voting device is able to correct this error and output the proper image.

A new API call has been added to PoCL that allows one to retrieve a redundant output buffer. It is intended for testing purposes, but being to inspect a manual buffer allows the end programmer to see the correcting behaviour in action and not blindly believe in the fault tolerance capabilities.

4.1.6 Nano-PoCL: Edge OpenCL Client for Near-Sensor Offloading

Nano-PoCL is a minified deployment of PoCL that can be executed in very low-end network connected edge devices to perform offloading to a centralized server using PoCL-R. A part of this activity was also lightweight video compression running on the edge device reducing the data size and transmission time. The video is then consumed and processed by a deep learning—based computer vision algorithm running in a centralized computer in the car.

Nano-PoCL work was implemented by using a RISC-V—based GAP8 system-on-chip (SoC) on an "AI Deck" add-on board for a Crazyflie nano-drone (Figure 18). This work consisted of two parts: reducing the amount of resources used by the PoCL code base itself and implementing the required POSIX functionality in the FreeRTOS environment provided by the GAP8 SDK.

The resource consumption of PoCL itself was reduced with a combination of compiler and toolchain configuration changes, removing functionality that is unnecessary for a remote-only implementation or nonsensical in a bare-metal environment, and shortening various initially rather conservative static allocations around the code base. Large static allocations were found frequently in code dealing with name or log strings provided by or meant for users and application developers. Since an embedded environment like a nano-drone or a car can work simply with a set of fixed IP addresses and device indices, much of this memory and logic was deemed unnecessary and removed.

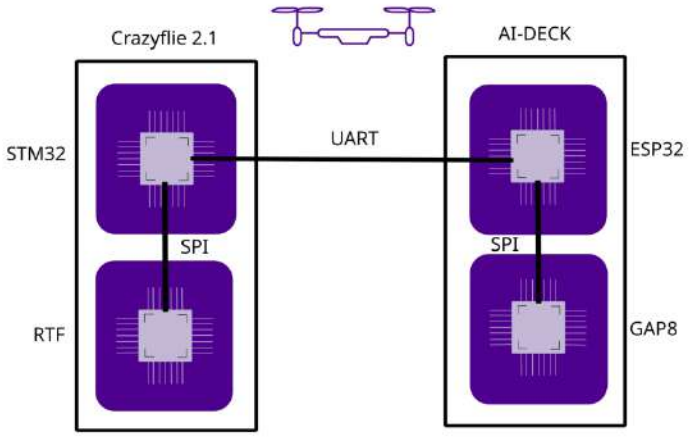


Figure 18: Hardware layout of the Crazyflie nano drone. The STM32 chip is the main control unit of the drone body. It communicates with the proprietary radio RTF chip via SPI and with the WiFi-enabled ESP32 on the AI Deck via UART. The main CPU of the AI Deck is the GAP8 that communicates with the ESP32 via SPI. The ESP32 also routes messages between the STM32 and the GAP8

Implementing the required POSIX networking APIs proved to be tricky since the GAP8 SoC itself does not have any network connectivity. This is instead handled by a separate ESP32 SoC with integrated Wi-Fi on the same circuit board. The ESP32 can additionally communicate with the main board of the drone.

Due to this board layout the networking APIs were implemented on the GAP8 as stubs that send short messages to the ESP32, which provides the actual functionality and exposes a POSIX-like API in its SDK. The ESP then replies to the GAP8 with a message containing the output data from the respective API call.

In order to demonstrate the functionality in a practical control loop scenario, a simple application was devised to run on the GAP8 SoC. The application takes a picture using the camera integrated in the AI Deck and finds the position of the brightest pixel in it. The task of finding the brightest pixel is offloaded to a different machine on the network via PoCL-Remote. This information is then used to send a command via the ESP32 to the STM32 control SoC on the drone's main board to turn the drone to face the position of said pixel and adjust its hovering altitude to match.

A number of problems were found while testing with this application. Most notably, the ESP32 SoC in charge of the Wi-Fi connection turned out to have rather aggressive power saving measures, causing it to constantly drop to a lower power state. Waking the chip up again for the next request added an unpredictable amount of latency to the next command, ranging anywhere from a few milliseconds to hundreds of milliseconds.

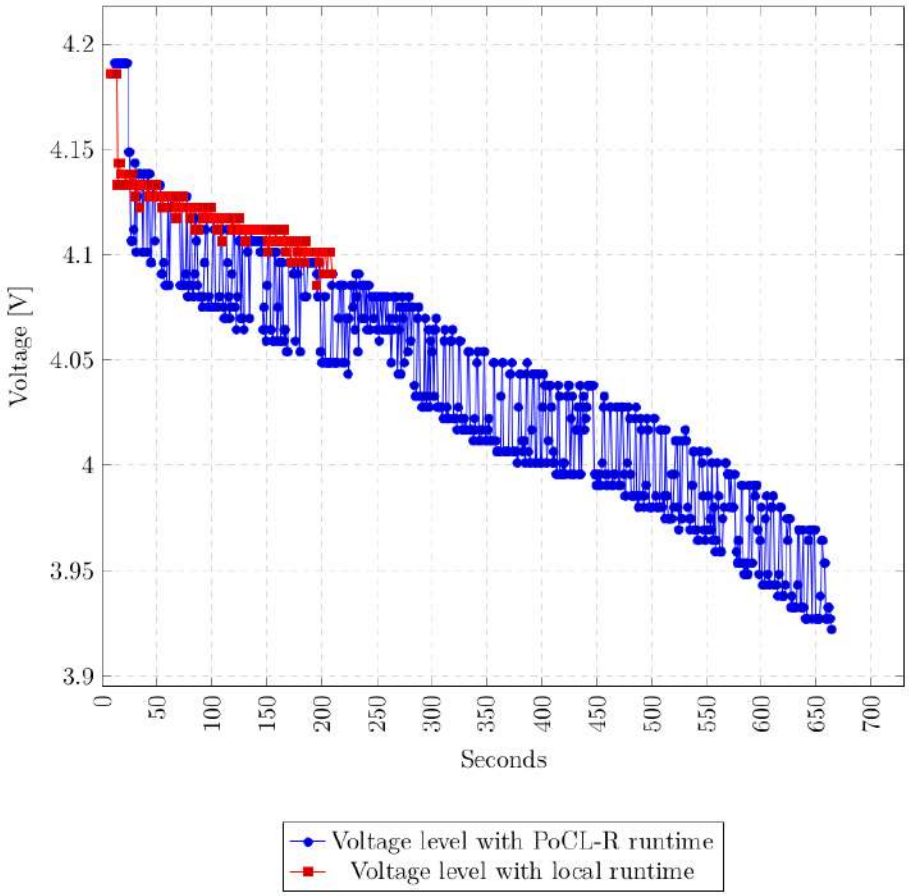


Figure 19: Battery voltage over time during the run time of the demonstrator application.

Second, the OpenCL kernel for finding the brightest pixel in the image turned out to be so lightweight that the additional delay and power consumption of powering on the Wi-Fi chip outweighed any benefits from offloading the kernel instead of performing the computations on the GAP8.

For the video compression part, a full demo is not integrated yet, however, we studied several compression algorithms and gathered preliminary results that are going to drive the eventual realization:

- ASTC texture compression was shown to achieve 2.3x the encoding speed of JPEG on a mobile device at the cost of about 1.7-4.4 percentage points (pp) of semantic segmentation accuracy (measured as mean intersection over union, mIoU) after retraining the model with decompressed images.
- We identified potential improvements of 22-23% of the encoding time of JPEG XS reference encoder at the cost of 0.3-0.4 pp mIoU after retraining. Nevertheless, the JPEG XS reference encoder is still too slow for real-time deployment and needs more optimizations.

- As compressed data is expected to be transmitted via a wireless link, we also studied Linear Video Coding (LVC) and its effect on computer vision task accuracy. LVC is a joined source-channel coding scheme consisting of a simple compression based on Discrete Cosine Transform (DCT) and additional linear transforms aimed to enhance the resilience against channel noise and packet losses. The object detection task showed high resilience against channel impairments while semantic segmentation showed a high resilience against discarding DCT coefficients.

The preliminary results justify the use of ASTC as a lightweight compression method and LVC as a method for both compression and hardening against wireless channel impairments.

5 Connected and Autonomous Vehicles

The idea of cooperation has been introduced to self-driving cars about a decade ago with the aim to reduce the occlusion caused by other users or the scene. More recently, the research efforts turned toward cooperative infrastructure bringing a new kind of the point of view as well as more processing.

High-quality localization, the ability for agents to estimate their poses reliably and accurately (i.e., positions and orientations) with respect to the surrounding environment or to a geographic coordinate system, is crucial. The GNSS, as a traditional solution, is not always available or reliable, due to reasons such as signal blockages, multipath reflection, and jamming [18]. One potential solution for localization in GNSS-denied environments is to utilise map matching techniques, given a prior map represented as a scalar field. Scalar fields associate a scalar value with every point in space, and applications include gravity anomaly [19], magnetic anomaly [19] [20], topographic [21], and olfaction, to name a few. Methods utilising scalar fields for localization regulate agents' dead-reckoning error growth through matching the information measured by on-board sensors with the prior given scalar field maps, such as terrain-aid navigation and magnetic anomaly–based navigation. CPSoSAWARE has engaged a multitude of modules for environmental modelling, self-localization, vehicle control, Driver System Monitoring, and cyber-attack immunization. All the modules above are involved in Level-3 and level-4 autonomous. Such modules synergy was also considered in CPSoSAWARE. Figure 20 illustrates all the modules developed in the automotive pillar of CPSoSAWARE as well as the synergies and the associations.

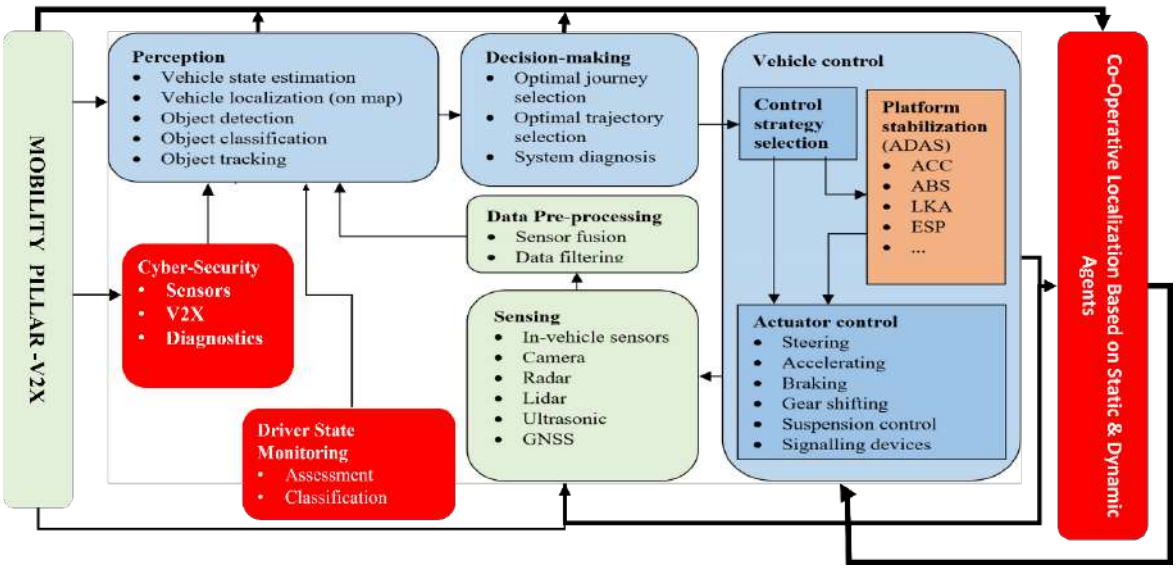


Figure 20: Interaction surfaces

5.1 Cooperative awareness

Cooperative awareness is realized in CPSoSAWARE in two complementary solutions, the Co – Operative Localisation based on dynamic agents (simulator based) and the Co – Operative Localization based on static agents. The integration of the components that cooperate to deliver these two solutions are described in sections 5.1.1 and 5.1.2 that follows.

5.1.1 Co – Operative localisation with dynamic agents (Simulation Based)

The simulation-based demonstration scenario for cooperative awareness is based on Carla. Carla is an open-source autonomous vehicle platform developed by the Urban Computing Foundation and is designed to be used as a testbed for developing and evaluating autonomous vehicle technologies. Carla provides a realistic simulation environment for testing and evaluating autonomous vehicle algorithms, including perception, localization, planning, and control. The Carla platform includes several tools and libraries for working with autonomous vehicles, including a server component that runs the simulation and provides an interface for connecting to the simulation from external clients and a Python client library for connecting to the simulation server and interacting with the simulation. The co – operative localisation with dynamic agents involves the interaction of various sub – modules as depicted in Figure 21.

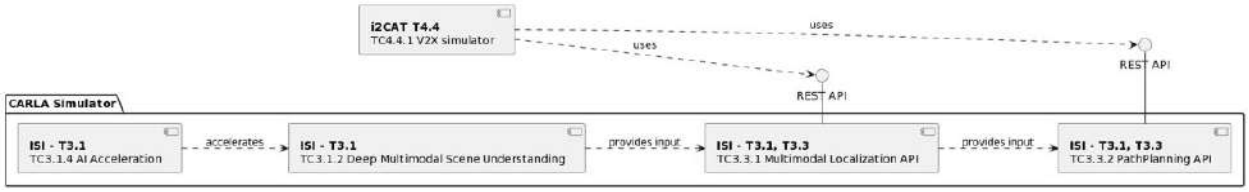


Figure 21: Submodules of the co - operative location with dynamic agents module

5.1.1.1 AI acceleration and deep multimodal scene understanding

As part of integrating Cooperative Awareness module in the CARLA simulator, the first key sub-module is about Accelerated Deep Multimodal Scene Understanding[5]. This sub-module is mounted to each simulated vehicle and is responsible for identifying nearby active road users, i.e., cars, pedestrians, etc., and extract useful scene analysis indicators such as relative distances and angles, using image and LIDAR data.

Before pilot execution: For the tasks of this submodule, we have initially employed two state-of-the-art deep learning models, i.e., SqueezeDet and PointPillar for 2D image and 3D point cloud processing, respectively, evaluated on standard benchmark of KITTI dataset and CARLA simulator in order to be "close to reality" operational. SqueezeDet is a fully convolutional detection network consisting of a feature-extraction part that extracts high dimensional feature maps for the input image, and ConvDet, a

[5] https://gitlab.com/isi_athena_rc/cpsosaware/multimodal-scene-understanding

convolutional layer to locate objects and predict their class. For the derivation of the final detection, the output is filtered based on a confidence index also extracted by the ConvDet layer. Feature-extraction (convolutional) part of SqueezeDet is based on SqueezeNet, which is a fully convolutional neural network that employs a special architecture that drastically reduces its size while remaining within the state-of-the-art performance territory. Its building block is the "fire" module that consists of a "squeeze" 1×1 convolutional layer to reduce the number of input channels, followed by 1×1 and 3×3 "expand" convolutional layers that are connected in parallel to the "squeezed" output. SqueezeNet consists of 8 such modules connected in series. PointPillar network, is designed for 3D object detection using LiDAR point clouds. The architecture of PointPillars consists of three main stages. More specifically, the first stage transforms the point cloud into a pseudo-image by grouping the points of the cloud into vertical columns, called pillars, that are positioned based on a partition of the $x - y$ plane. The second stage consists of a feature extraction backbone network providing high-level feature-rich representations of the input. Finally, object detection takes place in the third stage, which is responsible for producing 3D bounding boxes and confidence scores for the classes of interest.

After pilot execution: After the pilot execution, it was highlighted the need to combine the visual modalities in order to increase overall scene analysis ability, since standalone camera or LIDAR may fail to produce meaningful results. As such, a late fusion strategy takes place combining 2D driven detections and 3D driven detections. Initially, 3D bounding boxes are projected upon the 2D plane and converted to 2D bounding boxes. To fuse 2D and 3D measurements a non-maximal suppression driven approach takes place redefining the bounding boxes on the 2D space. Afterwards, to define vehicle range measurements 2D projects are matched to 3D points of the point cloud. Subsequently, each 3D point of the point cloud [xi, yi, zi] is projected upon the 3D image. Apart from the need of increasing scene analysis and understanding accuracy, it as also important to reduce the execution time of inferense so as to extract in real time usefull indicators about nearby vehicles. Therefore, in order to achieve the goal of acceleration weight-sharing model compression and acceleration (MCA) techniques are applied to the involved models. Concerning SqueezeDet, the focus is on its feature-extraction part, namely, SqueezeNet that, as already described, consists of 8 "fire" modules connected in series. SqueezeNet is responsible for roughly 83% of the total 5.3×10^9 MAC operations and 76% of the approximately 16 MB storage space required by SqueezeDet. Since it constitutes an already efficient network, we only targeted SqueezeNet's "expand" layers in our experiments. Acceleration was performed in 8 acceleration stages (one "expand" module per stage), followed by retraining. Concerning PointPillars, its feature-extraction (backbone) stage is responsible for 97.7% of the total MAC operations required. In total, the Pointpillars network encompasses 4.835×10^6 parameters and requires 63.835×10^9 MACs. For a good balance between acceleration and accuracy loss, we only targeted the convolutional layers of the backbone network comprising the second stage of PointPillars. Specifically, the targeted 2D- and 4×4 transposed 2D-convolutional layers, are responsible for approximately 47% and 44.4% of the total required MACs, respectively. Acceleration was performed on 16 acceleration stages with each stage involving the quantization of a particular layer, followed by fine-tuning. Using acceleration ratios of α = 10, 20, 30, and 40 on the targeted layers, lead to a reduction of the total required MACs by 82%, 86%, 88%, and 89%, or equivalently, to total model acceleration of PointPillars by $5.6\times$, $7.6\times$, $8.6\times$, and $9.2\times$, respectively.

Besides scene analysis and understaning indicators regarding nearby objects, visual data have been also utilized in order to estimate vehicle location as well as to determine the landmarks of the map using renowned SLAM algorithms[6].

Before pilot execution: Initially, state-of-the-art open source SLAM algorithms based on camera and LIDAR sensor have been integrated to CARLA simulator and thoroughly evaluated. The evaluation study highlighted the need for additional backend approaches in order to increase pose accuracy during challenging outdoor conditions of driving.

After pilot execution: For the purposes of pilot setup, the developed relocalization backend is build on top of these SLAM algorithms in order to refine and correct the estimated pose. This is achieved by combining and fusing the topologies of poses and landmarks generated by SLAM approach, through Graph Laplacian Processing technique and Kalman Filter.

For more details about AI Acceleration and Deep Multimodal Scene Understanding, see Deliverable D3.1 [22].

5.1.1.2 Multimodal localisation API

This submodule[7] exploits the concept of cooperation for multimodal sensor fusion among a group of interacting vehicles.

Before pilot execution: Relative measurements among neighboring vehicles, as well as their noisy positions were created synthetically, adding measurement noise to the corresponding ground thruth quantities of CARLA simulator. Ego vehicles process all these informations and additionally exploits data association along with Graph Laplacian Processsing technique in order to formulate a location estimation scheme and to match relative measurements with vehicles' ids.

After pilot execution: Instead of synthetic measuremens, the output of AI Acceleration and Deep Multimodal Scene Understanding modules have been used in order to extract realistic relative measurements and noisy positions of vehicles. More specifically, Multimodal Localization API of ego vehicle is feeded now by the AI Acceleration and Deep Multimodal Scene Understanding modules of itself and its neighbors, in order to estimate the positions of vehicles. The output of this processing is the estimated positions of ego vehicle and its neighbors, much more accurate than the noisy positions of vehicles generated by GPS or other state-of-the-art solutions..

[6] https://gitlab.com/isi_athena_rc/cpsosaware/odometers

[7] https://gitlab.com/isi_athena_rc/cpsosaware/cooperative-localization-and-tracking

See more details about Multimodal Localization API in Deliverable D3.3 [23].

5.1.1.3 Path planning API

After pilot execution setup: Multimodal Localization API highlighted the potential for exploting the estimated positions of a cluster of vehicles in order to design more efficient path planning strategies and to enhance safety, performance and effectiveness of autonomous and interconnected driving. Threfore, this submodule[8] aims to provide a robust way to adjust the acceleration of each platoon vehicle and avoid collisions. This is achieved by transforming the control problem into an iterative, finite-horizon optimization with local constraints. More specifically, we focus on cooperative control in order to design appropriate distributed algorithms such that the group of vehicles can reach consensus on the shared information in the presence of limited and unreliable information exchange and dynamically changing interaction topologies. The control objective of this sub-module is to make the network of vehicles maintain a rigid formation geometry by following a desired trajectory. To achieve this task, we have developed an Alternating Direction Method of Multipliers (ADMM) based scheme realizing distributed model-predictive controllers (MPCs).

See more details about Path Planning API in Deliverable D3.3 [23].

5.1.1.4 V2X simulator

Before pilot execution: Our framework combines a network and traffic simulator into CARLA. More specifically, Artery V2X Simulation framework, which is built on top of OMNET++ framework, was our choice for simulating network communications and more specifically V2X communications. For the control and the coordination of the simulating entities we have chosen SUMO. CARLA simulator supports a set of four combinations of simulation step (fixed and variable) and client-server synchronicity (synchronous and asynchronous), among which, the simulation stability and results repeatability is achieved through the choice of a fixed time step and the synchronous client-server interaction mode. This allows a single external client to define the pace of simulation progress without any concerns regarding the processing speed mismatches. Since we have multiple sub-systems with their own stepping logic, it is evident that a single place must exist, that will act as a clock gate and synchronization point. This role can be realized by the Traffic Control Interface (TraCI) of the SUMO simulator that already supports interactions with both CARLA and OMNET++, in different contexts - in particular, with the Artery V2X Simulation framework. Furthermore, CARLA interacts with ROS through the CARLA-ROS Bridge. Since in synchronous mode, only one client can tick the CARLA server, the Bridge must be also launched in passive mode, for the timing of the ROS subsystem to follow the single system clock source, too. Finally, in order to export to the ROS subsystem important, application-level information, such as the ETSI ITS CAM from the Artery/OMNET++ network simulation, the ros-etsi-its-messages encapsulation library can be used. The whole chain step is

[8] https://gitlab.com/isi_athena_rc/cpsosaware/cooperative_path_planning

controlled by the slowest element, which is the network simulator and can be started or stopped through the ONMET++ user interface.

After pilot execution: V2X simulator was employed in order to simulate realistic conditions of V2V communication among the interconnected vehicles which utilize the Multimodal Localization API. In that way, realistic deployment of Multimodal Localization API has been performed since the impact of network delay on cooperative positions' estimation has been also considered.

5.1.2 Co – Operative localisation with static agents

The main purpose of the co-operative localization is to perform registration between the Prior Map of a test area, obtained through screening the area throughout the training phase and the the current map acquired during scanning the area. The Sparse MAp RegisTration (SMART) module is ideally to define a bijective transformation between two versions of the map, coming from consecutive observations. This would mean that all the landmarks detected in one map would correspond to just one landmark in the second map. Using landmarks instead of points in a dense point cloud is the reason we use the word sparse in the naming of the module.

The problem to be solved is using the detected landmarks in a way in which the outliers are ignored, and registration is performed with a minimal spatial error inflicted, so that the vehicle localization is in turn more accurate. To achieve this, we have to get enough corresponding landmarks from the Landmark Extraction module in subsequent iterations, so that registration is feasible.

5.1.2.1 Software Structure

SmaRt functionalities are inside Least Mean Squares with Measurement Exchange (LME). When they will be used in a standalone module, the interfaces will change accordingly. The pose output produced by LME is better suited as an output here, while not showing the full functionality. For calling LME functions, please refer to the LME documentation. The part relevant to SmaRt, is the calls related to ICP, which is also a class and instantiated as a member variable of RoadClass, once per camera. Once the landmarks are added to the list of a RoadSide object, ICP can be called using first the function to add all points to the point cloud. This function allows for augmentation of Parking Slot landmarks, by adding more "virtual" points on the line defining them (from entry point to back point):

1. void addAllPts2ICP(i32 numOfVirtPts);

We can also have the trained point cloud to compare to, we have to somehow load it from memory. At the moment, for the Algo C implementation this is done by dumping the detected landmarks at the phase of training to several text _les, then loading them in the beginning of the testing phase. For the Proof of Concept stage, saving is done once, when the car is stopped at the drop-o_ zone and loading is done in the beginning of the test drive. The functions to do this, are also in LME:

2. void saveDropO_(); void loadDropO_();

Then, 2d ICP based registration can take place using:

3. void calcPose(const RoadSide& trainedRS, bool smooth = false);

This function performs ICP registration and pose estimation at once, allowing the user to smooth the results so that the median value of past estimations is used, avoiding jitter in the output. The core function of PMD:

4. bool run (Interface::Map const& interfaces, KeyboardFlags const& keyboardFlag s) override;

Updates and gets the output to the interface of LME, that carries the :

5. void updateInterface (TimeStamp time);
6. pmd::interfaces::PMD getInterfaceObject(void);

5.1.2.2 Module Description

This work follows a single camera approach, which allows for redundancy in cases where one of the cameras is defective, or soiled. It also allows late fusion of the results to create an optimized global map. The main process is based on an Iterative Closest Point implementation running on the trained vs. the testing point clouds. These two-point clouds are enhanced versions of the most prominent point on the detected landmarks, which now are only PMD related (closest vertex for lines, entry point for slots). The most important landmarks are the slots in our case, so they can be augmented by additional points on their defining midline.

The general block diagram of SmaRt and its interaction with LME in its intended form is depicted in Figure 22.

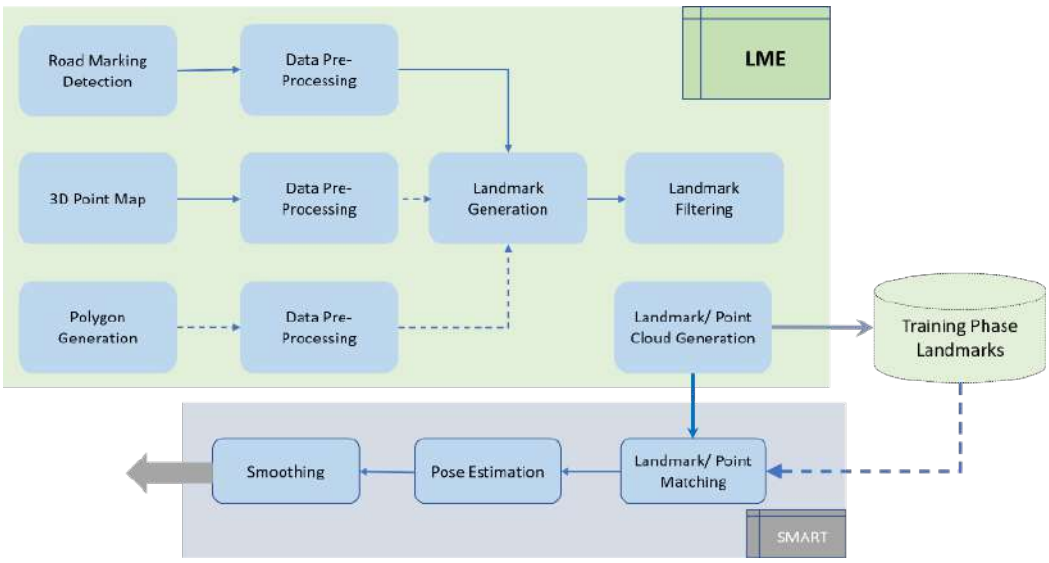


Figure 22: The block diagram of LME and SmaRt

5.1.2.3 Re – Localization module

To be technically able to apply motion planning in the valet parking applications, the basic idea of planning in traditional robotics engineering needs to be adapted to the automotive industry requirements. Due to the huge size of required maps, long driving lengths, security and safety concerns along with dynamics limitations of the automotive section, the motion planning algorithms which are being hired in the automotive industry should satisfy these concerns. To satisfy the requirements of Home Zone Parking, motion planning in autonomous vehicles has been divided in three main sections: global, local planning and collision avoidance (Figure 23).

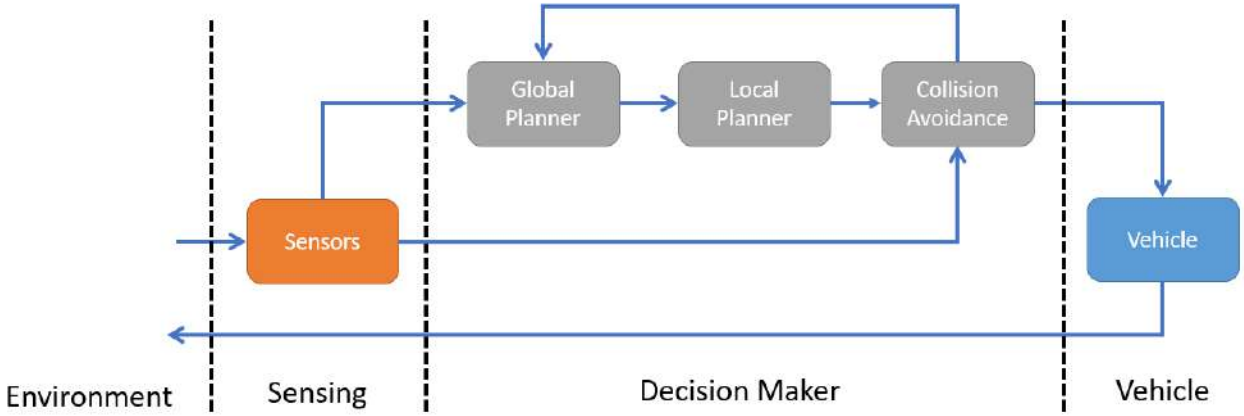


Figure 23: The framework of an autonomous system.

5.1.2.4 Global planning

CPSoSAWARE Co-Operative Awareness solution employs Global path planner to provides the optimal path regarding the provided KPIs of the planning. E.g. using a Hybrid A* used to provide waypoints to the target area, as illustrated in Figure 24.

Figure 24: Planning a global path for a test vehicle in Panasonic Automotive building to go to the Parking area using Hybrid A*

The global path is planned offline before the robot start to move. This global path aids the robot to traverse within the real environment because the feasible optimal path has been constructed within the environment. However, in order to solve the Robot Path Planning (RPP) problem when the robot is faced with obstacles. Local path is constructed online while the vehicle avoids the obstacles in a real time environment. The performances of the algorithm in terms of computational efficiency was observed and evaluated based on the distance, time and number of iteration the algorithm takes to find an optimal path.

5.1.2.5 Local planning

By knowing the global waypoints which are required to be followed, an autonomous agent finds its local optimum path to follow due to the mechanical and physical constraints of the vehicle as illustrated in Figure 25.

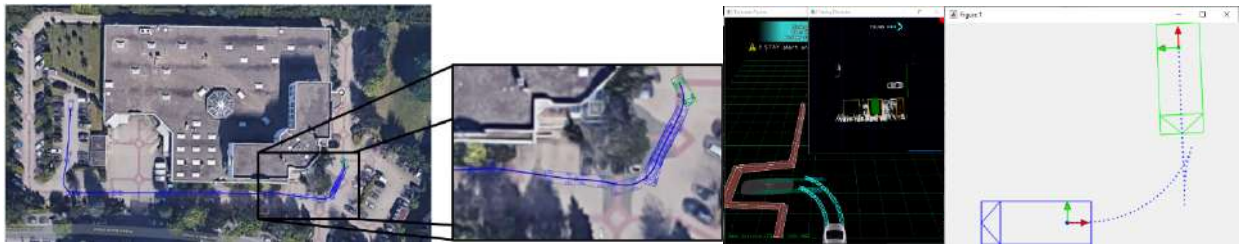


Figure 25: Planning local path section to follow a global Path for a test vehicle in Panasonic building (Left) and a Clothoid based parking maneuver from the parking area to the final target position (Right)

As mentioned above local path planning algorithm is employed to avoid collisions. Collision avoidance algorithms are responsible to avoid any possible collision to the moving or stationary objects, here the algorithms which keep the ego vehicle away from the objects will be used as shown in Figure 26.

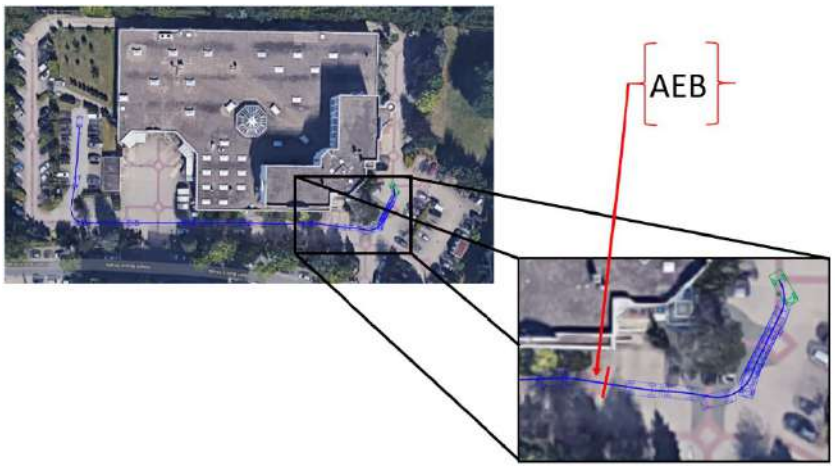


Figure 26: Avoiding a collision while following a path for a test vehicle

Finally, for the co-operative localization solution to be fulfilled, the Re-localization Module is needed. In the case of co-operative localization scenario, tested in CPSoSAWARE, the size of used map is much smaller than the real "last mile" or valet parking solutions:

5.1.2.6 Re – localisation module

The module implements a state machine based on the following states:

1. **MANEUVER_STATE_ERROR (Planning state: OFF):** This state represents the condition where the system has detected some unusual conditions and reported an error in the state machine.
2. **MANEUVER_STATE_MANEUAL (Planning state: OFF):** Through this condition, the car is driven manually or the vehicle is in the training-mode and the relocalization module state is set to "training".
3. **MANEUVER_STATE_INITIAL (Planning state: ACTIVE):** The vehicle is in the trial parking and the re-localization module state is set to "parking". In the case we need to just drive a very short distance forward to recalculate the location of the vehicle: Type of planner: a simple short (1 to 1.5 m) Straight line, no lateral positioning (no steering control, is set to be zero). No especial algorithm is used. Just a simple geometrical calculation Planning speed: 2 kph with longitudinal control (due to the very high uncertainty)

4. **MANEUVER_STATE_EXPLORING (Planning state: ACTIVE):** In this mode, the re-localization module has already got trained and the current state of that is “parking”. The initial re - calculated intermediate and final target positions are provided to the planner as depicted in Figure 27:

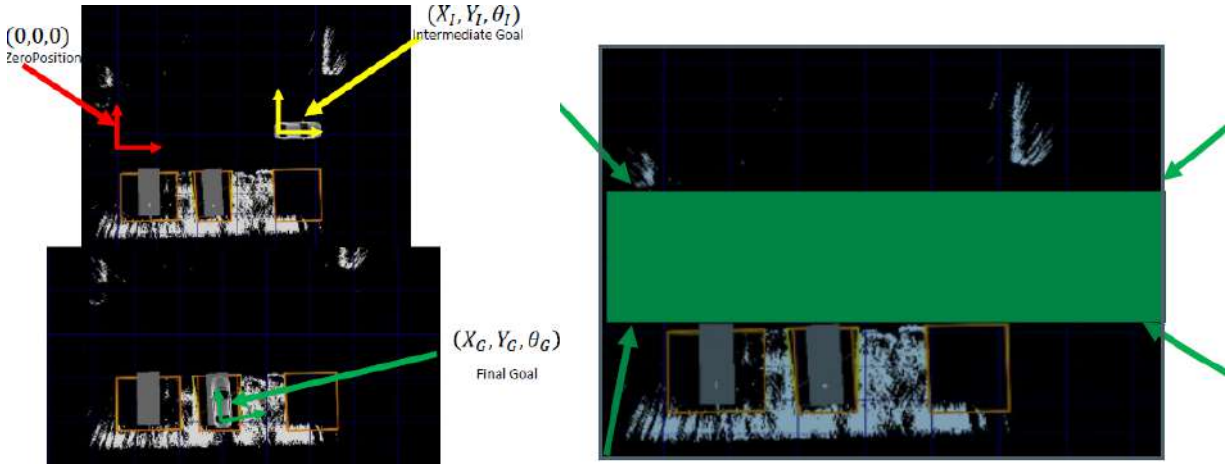


Figure 27: Illustration of the initial position, intermediate and goal positions as derived by the Path Planning algorithms

The path planning method is based on Clothoids with lateral and longitudinal controller Planning speed: up to 10 kph (normally around 3kph in this case) as illustrated in Figure 28.

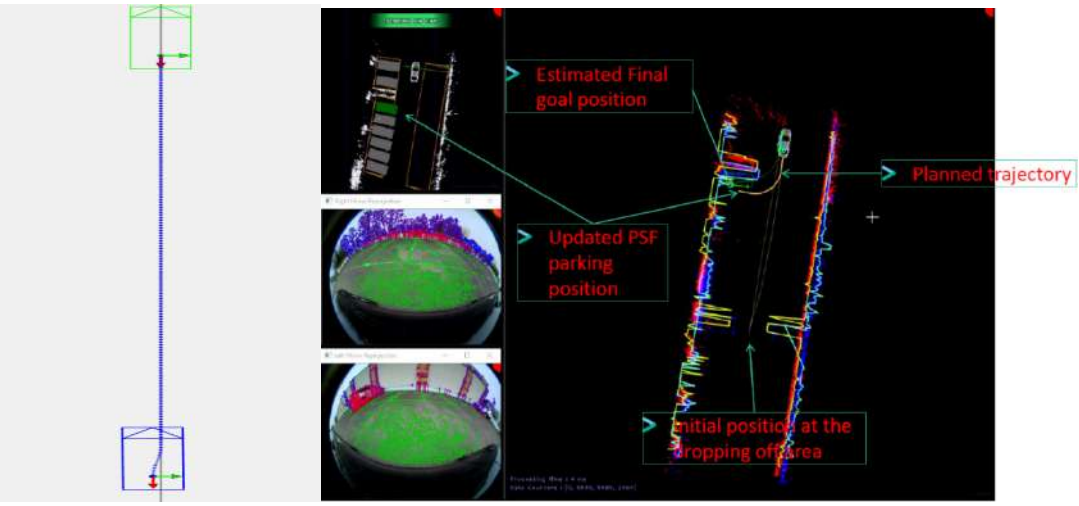


Figure 28: Path Planning part of the cooperative awareness and output of the map registration and final positions

5. **MANEUVER_STATE_AUTO_DRIVE (Planning state: OFF):** This state encodes the situation where whenever a planned path is exported to the vehicle and is confirmed with the controller, the planner state will go the auto drive state.
6. **MANEUVER_STATE_RE_EXPLORING (Planning state: ACTIVE):** In this state the planner waits until the perception part reports a valid parking position, and the vehicle is stationary. When the vehicle is in auto drive mode and an error is detected in path following, or even if a new target position is provided which is way too off from the previous target position it will force the planner to go re-planning state.
7. **MANEUVER_STATE_EXPLORING (Planning state: ACTIVE):** The main work in this state is the error calculation while the path is followed. It is done as follow:
   - The planned path is converted to the world coordinates for the corresponding initial planned position.
   - A virtual path in global coordinates is calculated for the front axle of the vehicle for the planned path.
   - If the vehicle is moving forward the position of the middle of front axle of the ego vehicle is calculated based on the provided world coordinate of the system (odom)
   - From the converted list of the planned path the closet point (in X direction) to the current odomposition is found and the following error is path following is calculated:

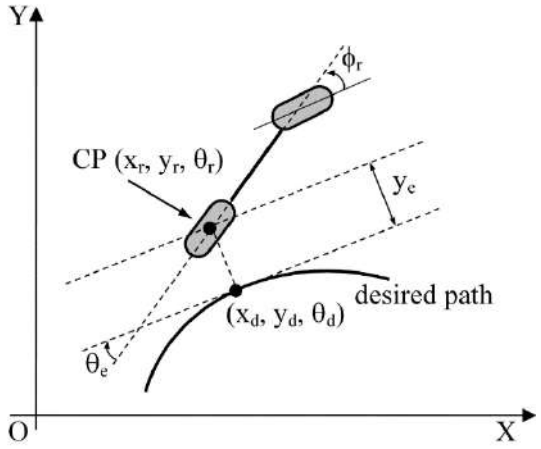


Figure 29: Path Planning steps involved in state 6.

$$\begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix} = \begin{bmatrix} \cos\theta_d & \sin\theta_d & 0 \\ -\sin\theta_d & \cos\theta_d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x_d \\ y_r - y_d \\ \theta_r - \theta_d \end{bmatrix} \rightarrow \begin{cases} \dot{x_e} = & -v_d + v_r \cos\theta_e + y_e \frac{v_d}{l} \tan\varphi_d \\ \dot{y_e} = & v_r \sin\theta_e - x_e \frac{v_d}{l} \tan\varphi_d \\ \dot{\theta_e} = & \frac{v_r}{l} \tan\varphi_r - \frac{v_d}{l} \tan\varphi_d \end{cases} \quad \textbf{Equation 1}$$

If the planner decides that the vehicle is off from the planned path either can re plan using state 4 or request an AEB.

8. **MANEUVER_STATE_PARKING:** While the vehicle following the path in exploration mode, the target position of PSF and Re-localization modules are received and when the planner confirm

that they are identical location. The normal parking manoeuvres (Clothoid-based) will be planned accordingly.

5.1.2.7 Odom fusion

These deliverable intents to describe the fusion of different odometry cues such as Visual Odometry, Vehicle Odometry, and GPS, into a unified estimate, which robustified the estimation of vehicle location under a diversity of weathering, illumination conditions and kinematics.

The basic task of this module is to choose and fuse input odometries from the sources offered by our CPSoSAWARE in order to create an output trajectory which is as correct and robust as possible.

Since GPS is the only available input that allows for an absolute positional measurement, it seems natural that GPS information should be included in the odometry fusion. All other odometries follow the dead reckoning principle, meaning that they have no means to recover from any inaccuracies in a past frame. However, the frame-to-frame accuracy of GPS is much lower than of the other means of collecting odometry information. Experiments indicate that this coarseness makes a direct integration of GPS information in real-time odometry very difficult if the excellent detail of the odometries should be retained.

The first approach of using GPS information in odometry fusion thus is not a real-time one but one that alters that odometry of past frames as well. While this has serious drawbacks for application use, it shows that an integration of GPS is possible while retaining smooth trajectories.

During testing and development, the odomfusion module certainly depends on all modules which create an odometry as their output. Once the work in this module has become mature enough, it may be possible to cancel some of these dependencies if their odometries are not needed as input.

Apart from the common algorithm and math library, odometry fusion depends on the modules that provide its input, potentially including:

- Vehicle Odometry
- Visual Odometry
- GPS

OpenCV is used only during debugging to store images etc. It is not part of the implementation of any of the algorithms. No other third-party libraries are being used in odometry fusion

The first estimator to include GPS information uses nonlinear optimization over a window of past frames. In each frame, it computes the affine transformation that minimizes the distance to the GPS trajectory if applied to the trajectory points in a windowed fashion with its weight declining over time.

In order to allow for quantitative evaluation, several dedicated sets of test data have been selected/ recorded which can be categorized as:

a. Same position in start and end. This is the simplest way to analyse for drift and inaccurately measured motions globally.
b. Repetition of trajectory. This includes two or more recordings and may allow to find inaccuracies where they occur rather than just globally.
c. Measured driving distance: This allows for evaluating the estimated global scale

During the performance optimization of a computer vision system, developers frequently run into platform-level inefficiencies and bottlenecks that cannot be addressed by traditional methods. CPSoSAWARE addresses such system-level issues by means of a graph-based computation model. This approach differs from the traditional acceleration of one-off functions and exposes optimization possibilities that might not be available or obvious with traditional computer vision libraries such as OpenCV. Remote Processing is simply the practice of computing results on a non-host core such as a GPU, a DSP, or other specialized core such as an accelerator. We will refer to equation-2 to discuss its impact on optimization strategies. This equation expresses the latency of remote processing for a single function call:

$$\boldsymbol{L_r(1) = C_{lf} + IPC_{send} + C_{ri} + exec_r(data, params) + C_{rf} + IPC_{recv} + C_{li}} \quad \textbf{Equation 2}$$

where

- L_r Latency of remote processing.
- C_{lf} Host Core Cache Flush of impacted data
- IPC_{send} Total latency from Host-to-Remote-Core activation time. This accounts for line transmission, and possibly also system-thread switching and OS kernel/driver overheads.
- C_{ri} Remote Core Cache invalidate for affected data.
- $exec_r(data, params)$ Remote Execution time varies on data size and other parameters.
- C_{rf} Remote Core Cache flush for affected data.
- IPC_{recv} Total latency from Remote-Core-to-Host activation time (with overheads as in IPC_{send}).

5.1.2.7.1 Runtime Performance on multiple Platforms

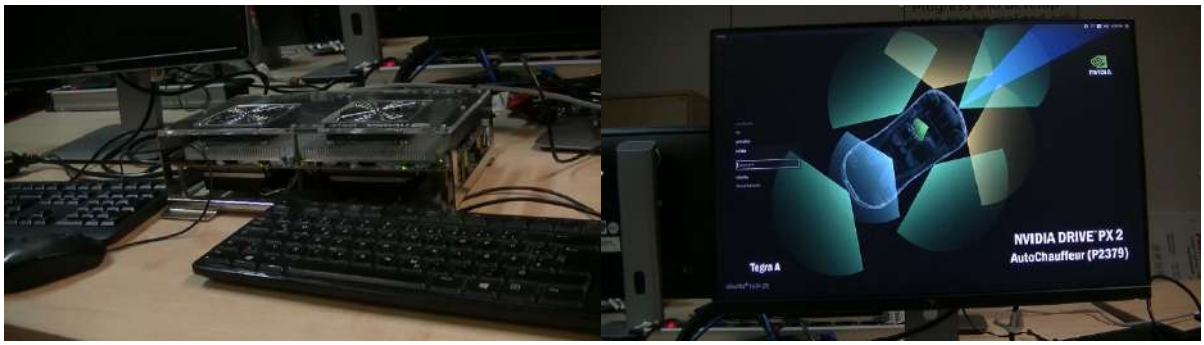
The odom-Fusion Module contributed to CPSoSAWARE project was demonstrated on Dell Latitude E6540 Nvidia DrivePX2, Texas Instruments TDA2X and Raspberry Pi.

The runtime of the different real-time odometry fusion algorithms is displayed in the following table. they have been tested on a standard 2015 Laptop workstation (Dell Latitude E6540), Drive-PX2, TDA2X and Rasberry pi. Note that the implementations have been optimized for fast performance. Also note that it may not be mandatory to run odometry fusion on every frame. The timing in Table 1-Table 4 is given in milliseconds.

Table 1: Odom Fusion Runtime Profile on (Dell Latitude E6540)

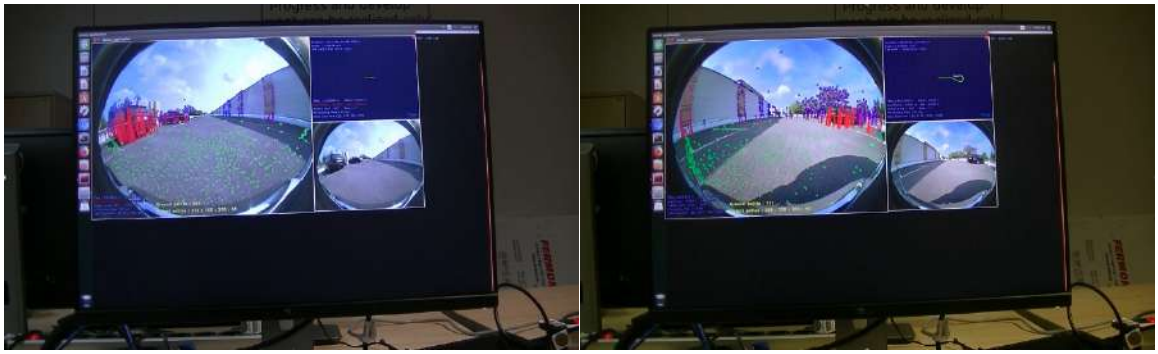
Fusion mode	Average Runtime	Standard Deviation
Ekf (original)	0.008	0.002
Linear KF (veh + vis)	0.011	0.003
Linear KF (veh + vis + GPS)	0.017	0.003

The output of odomFusion on Nividia Drive-PX2 as well as the runtime profiling are presented on **Figure 30** and **Table 2** respectively. As described in the section above, timing is given in milliseconds.



(a) (b)

(c) (d)



(e) (f)

(g) (h)

Figure 30: Odom-Fusion output running on DrivePX2. (a),(b): Depict the hardware setup and the development environment. (c)-(h) Visual output of the odomFusion Running on Nvidia Drive PX2

Table 2: Odom Fusion Runtime Profile on (on Nvidia Drive PX2)

Fusion mode	Average Runtime	Standard Deviation
Ekf (original)	0.004	0.002
Linear KF (veh + vis)	0.009	0.0023
Linear KF (veh + vis + GPS)	0.013	0.0021

The Odom-Fusion Porting on TDA2X corresponds to a multi-core development scheme, where the distribution of computational load on the processors is as follows:

- ARM15 (80%)
- 2 DSP (30%)

- 1EVE (15%)
- 3 EVE not used
- 15fps

In association to the above subsection, the runtime profiling is illustrated in Table 3:

Table 3: Odom Fusion Runtime Profile on (on TDA2x)

Fusion mode	Average Runtime	Standard Deviation
Ekf (original)	0.009	0.004
Linear KF (veh + vis)	0.019	0.005
Linear KF (veh + vis + GPS)	0.023	0.0054

The Odom-Fusion Porting on Raspberry Pi involves development scheme, where the computational load is on the CA72 ARM core, reaches 62%. The timing profile for the odom Fusion on raspberry pi is depicted on Table 4.

Table 4: Odom Fusion Runtime Profile on (on Raspberry Pi)

Fusion mode	Average Runtime	Standard Deviation
Ekf (original)	0.007	0.002
Linear KF (veh + vis)	0.015	0.0025
Linear KF (veh + vis + GPS)	0.019	0.0033

5.2 V2X communications Simulation Environment

5.2.1 Interface connection from V2X communications simulation environment to storage service

The aim of this integration was to serve a specific scenario with vehicular communications and record all the messages that are being sent and received along the simulation. A scenario with the SUMO simulator containing different number of vehicles was created, and this simulation was connected to OMNeT++ which will translate the vehicles into nodes and send CAM messages from each vehicle to all its neighbours.

Therefore, the objective is to collect all the CAM messages transmitted in a simulation and store them in an external storage service. That would be highly beneficial since some simulations can be really large

(with lots of vehicles or for an extended period of time) and the amount of data generated needs to be sorted and stored efficiently.

5.2.1.1 Dataset creation

First, the information from all the CAM messages of the simulation was needed. So, for each vehicle of the simulation, the time when it sends or receives any message is detected, and at that instant, its information will be dumped into a CSV file. Therefore, at the end of the simulation, each vehicle will have two files, one with all the messages sent by that vehicle, and one with all messages that it has received.

The name of the resulting files contains the OMNeT++ ID of the vehicle and the type of message stored (sent or received). An example of a "sent" and "received" file could be *"684174695_sent.csv"* and *"684174695_received.csv"* respectively.Also, these files store some of the most relevant parameters of the CAM messages, which can vary depending on the type of message:

- Sent file. The name corresponds to the transmitter vehicle's ID:

Table 5 Parameters stored in a "sent" file

Sent Time	Internal CAM reference time in milliseconds [ms]
Simulation Time	Simulation time when the message is sent in seconds [s]
Origin Latitude	Transmitter vehicle position (geographic coord.): Latitude
Origin Longitude	Transmitter vehicle position (geographic coord.): Latitude
X	Transmitter vehicle position (cartesian coord.): X-axis, horizontal
Y	Transmitter vehicle position (cartesian coord.): Y-axis, vertical

- Received file. The name corresponds to the receiver vehicle's ID:

Table 6 : Parameters stored in a "recieved" file.

Station ID	ID of the transmitter vehicles
Sent Time	Internal CAM reference time of the sent message in milliseconds [ms]
Received Time	Internal CAM reference time of received message in milliseconds [ms]

Received Simulation Time	Simulation time when the message is received in seconds [s]
Origin Latitude	Transmitter vehicle position (geographic coord.): Latitude
Origin Longitude	Transmitter vehicle position (geographic coord.): Latitude
Destiny Latitude	Receiver vehicle position (geographic coord.): Latitude
Destiny Longitude	Receiver vehicle position (geographic coord.): Latitude
Destiny X	Receiver vehicle position (cartesian coord.): X-axis, horizontal
Destiny Y	Receiver vehicle position (cartesian coord.): Y-axis, vertical

5.2.1.2 Send datasets to the database

Since all the files are generated, a way for sending them all to the database was needed. Thus, the implementation of a ROS interface that will read all the CSV files and send its information to the storage service was needed.

ROS was chosen because it is a standard method to interact with other simulators (such as OMNeT++ and CARLA), and it allows to send data to external applications like this database. ROS follows a structure of nodes that can publish information into a topic, or subscribe to them, and is orchestrated with a ROS Master.

For this service two clients were implemented, a *publisher* and a *reader*:

- **Publisher:** It is responsible for publishing the CAM messages to the ROS master specifying different topics for each type of message (Sent and Received). It is able to publish data already generated and stored in a CSV file or to publish the data when it is generated in the simulation at the instance of sending and receiving a message. To publish the data stored in a CSV file, it reads the file and splits the information into lines which publishes afterwards. In the case of publishing the data when the simulation is running, it waits for the triggers of sending or receiving a message to publish it.
- **Reader:** This client has another ROS node that will read all the messages sent to the ROS master by the previous client and then it will publish them into the storage service.

The main reason for using this system is that the publisher and the reader clients can be executed from different computers. The publisher corresponds to the machine that has run the simulation and has all

the generated CSV files. The reader client needs to be executed from a machine that must have access to IBM's database.

Finally, in order to publish the messages into the storage service, the reader node needs to follow the following steps:

1. Access the database web from IBM[9]
2. Introduce the credentials to authenticate the account username: aa, password: aa
3. The information is stored as a JSON file, with the following namespace = “cars_data”

- And last, the messages will be stored in different directions based on the indicated topic: Sent Topic[10], Received Topic[11]

5.2.2 Interface connection from V2X communications simulation environment to ISI location improvement simulator

The collaboration between i2CAT and ISI was aimed to feed the ISI location improvement algorithms with more realistic data gathered from the V2X messages. The objective of this integration is to create a simulation on CARLA, which will use the Local Dynamic Map (LDM) data to improve the vehicle location. So, the data from the LDM will be generated on OMNeT++, and then used in CARLA for the location improvement.

In order to do so, we have set a common scenario, specifically the Town05 map provided by CARLA simulator. We have used and modified this scenario in SUMO (creating a customized simulation with 25 vehicles) and fed the OMNeT++ simulation with it, this way, being able to simulate the transmission of messages between vehicles.

Afterward, once the OMNeT++ simulation has finished, we can collect all the parameters extracted from the sent and received CAM messages, and combine them into a global CSV file with the LDM’s of all the vehicles.

---

[9] https://CPSoSAWAREeu.draco.res.ibm.com/CPSoSAWARE -SAT/integration/1.0.0/auuthenticate

[10] https://CPSoSAWAREeu.draco.res.ibm.com/CPSoSAWARE-SAT/integration/1.0.0/namespaces/cars_data/classes/sended_updated/schemas/aa/data/all

[11] https://CPSoSAWAREeu.draco.res.ibm.com/CPSoSAWARE-SAT/integration/1.0.0/namespaces/cars_data/classes/recived/schemas/aa/data/all

5.2.2.1 Dataset creation

The generation of the dataset is performed during the simulation. Every time a vehicle receives a message from any neighboring vehicle it stores the information in a new line of the database. But there are some parameters that need to be obtained each time a message is sent (not received), so these ones are collected during the simulation, and once it's finished will be added to the previous file, generating a complete dataset with all the LDM's parameters. The database is recorded in CSV format, and each entry has the following parameters:

Table 7: Parameters stored in the Local Dynamic Map (LDM)

LDM of the vehicle	SUMO Identifier of the vehicle receiving the message.
LDM Simulation Time	Time in which the LDM of the vehicle has been updated. Corresponding to the last received CAM. Units: seconds.
Vehicle Identifier	SUMO Identifier of the vehicle sending the message.
Generated time	Time of the simulation in which the message has been generated by the sender vehicle.
Received time	Time of the simulation in which the message has been received.
Position X	SUMO cartesian coordinate X of the sender vehicle.
Position Y	SUMO cartesian coordinate Y of the sender vehicle.
Speed X	Speed on the X axis of the sender vehicle in m/s.
Speed Y	Speed on the Y axis of the sender vehicle in m/s.
Acceleration X	Acceleration on the X axis of the sender vehicle in m/s^2.
Acceleration Y	Acceleration on the Y axis of the sender vehicle in m/s^2.

This database contains the LDM's of all the vehicles in the simulation. To obtain the LDM for a specific vehicle, the file just needs to be sorted by the parameter "LDM of the vehicle" with the name of the desired vehicle.

5.3 Real Vehicle Environment

For the real vehicle demonstrator of the automotive Pilot, multiple modules (Figure 31) have been developed and integrated corresponding to all 3 phases of the automotive Pilot namely:

- Sense
- Understand
- Act

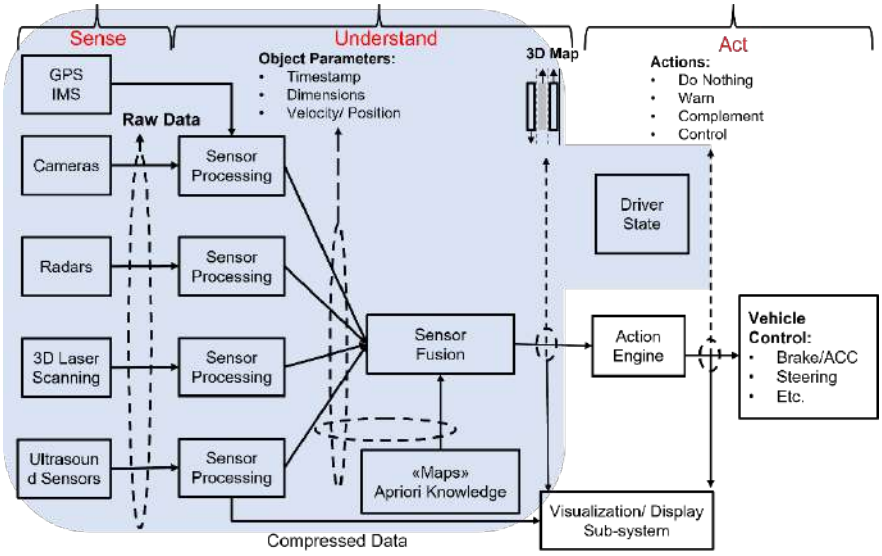


Figure 31: Modules Integrated on CPSoSAWARE platforms.

The integration framework engaged on the real vehicle is a ROS-like framework. The fundamental entities of the framework are: (a) nodes, (b) messages, (c) topics, and (d) services. Nodes are processes that perform computation. The framework is designed to be modular at a fine-grained scale. The use of the term “node” arises from visualizations of ROS-based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph, with processes as graph nodes and the peer-to-peer links as arcs. Nodes communicate with each other by passing messages. A node sends a message by publishing it to a given topic, which is simply a string such as “odometry” or “map.” A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

CPSoSAWARE involves delivering results on Car-PC, Drive Px2, Raspberry Pi and Texas Instruments TDA2X as illustrated on Figure 32.

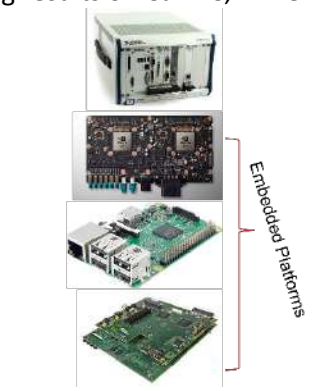


Figure 32: Integration Platforms

5.4 Cybersecurity in connected vehicles

CPSoSAWARE tackles the cybersecurity challenges in connected vehicles from the sensor and communication layers as well. The involved CPSoSAWARE components and their interactions are depicted in Figure 33 and discussed in more details in Sections 5.4.1, 5.4.2 and 5.4.3 that follows.

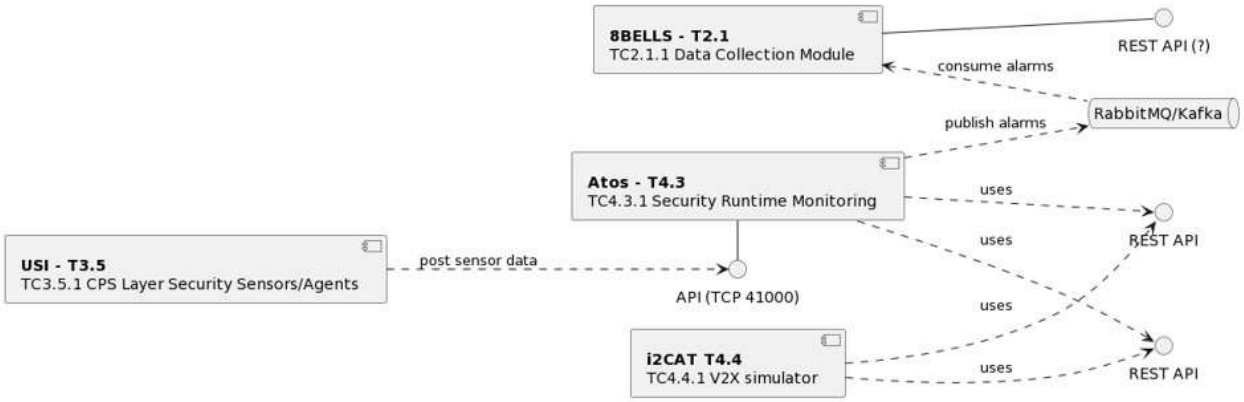


Figure 33: Cybersecurity components and interactions

5.4.1 Sensor Layer

The automotive sector is going through a significant development trend regarding autonomous driving. Sensors, communication systems, actuators, etc., are some of the components becoming more and more common in vehicles due to this. This has led to more complexity, which has increased the number of ways cyber-attacks can occur. This leads to attackers having access to the vehicles from outside. Several security-related projects have been conducted as a result of these concerns. Bold initiatives are being taken by technology giants, automobile manufacturers, and governments all over the world to build safer and more affordable AVs and bring them to market quickly. Cooperation is essential for tackling the issue of cybersecurity. Table 8 summarizes the scenarios usecases explored during the CyberSecurity Demonstrators of CPSoSAWARE.

Table 8: Attack scenarios for the real vehicle demo as presented in deliverable D6.2 [24]

Use Cases- CyberSecurity Use Case	
1	Attack on the Camera Sensor Layer: This scenario would involve a cyber-attack based on activating some malicious software which got installed during the software update process. Throughout this use-case the camera sensor could be attacked in a number of different ways, which could vary between adding noise lying on specific bands of the frequency spectrum/ introducing morphological deformations/ on the whole or parts of the image.
2	Attack on the Camera Sensor Layer by de-synchronizing the data: Throughout this scenario, the cyber-attack will be geared towards disturbing the association between the captured frames and the timestamp assigned to them. This will cause the failure of the perception engine, as all the architectural modules performing stochastic filtering on the scene observations will be affected by error. This use case should study the potential and the limitations of the cyber-attack detection and mitigation engine in assessing and recovering the failures.

3	Attack on the Camera Sensor by a remote agent: In addition to the scenario, the cyber-attack detection and mitigation engine will be used to detect and mitigate the camera signal distortion in the case that a malicious remote agent interferes with the test vehicle by knowing the IP of the processing unit and sharing some erroneous data. More specifically, this use case will assume that the remote agent sends via V2X communication: time zone/ daylight related data in order some sensor parameters (e.g.: gain/exposure time) to be tuned accordingly.

The demo scenarios executed as part of automotive use case demonstrations involve a malicious user holding a remote control through which it attacks to the sensors data, disturbing both the quality of the sensor signal and the timestamps of the data recorded. As illustrated in Figure 34, the scene is perceived while a malicious user attacks to the vehicle sensors to degrade the scene understanding output. After the pre-processing and perception phase, the cyber attacks are detected and classified.

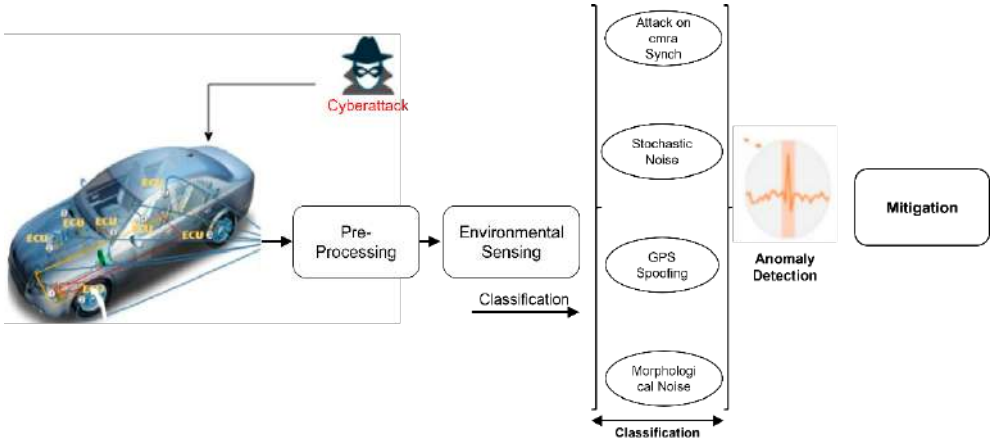


Figure 34: Phases of Cyber-attack detection, classification and mitigation for the automotive pilot

The types and the scenarios of the attacks were defined in D2.2 deliverable contributed by CPSOSAWARE's consortium at the beginning of the project and is also presented in Figure 34 above.

The scenarios presented in Table 8 were used to investigate the potential and limitations of the cyber-attack detection and mitigation engine across a wide range of perception functions related to:

- Moving Object Detection
- Self-Localization
- Occupancy Grid Mapping/ Object Boundaries Definition
- Fully autonomous parking.

The number and the extend of perception engine components tested, exceeded the initial planning as this was specified in Wp2, WP3 and WP4. More specifically, while in the aforementioned work packages it was planned to have the cyber-attack detection and mitigation engine tested only for use cases (1) and (4), we finally tested it across perception components related to object detection, navigation and automated parking. The modules involved in the attack scenarios are enclosed in the orange segment of Figure 35.

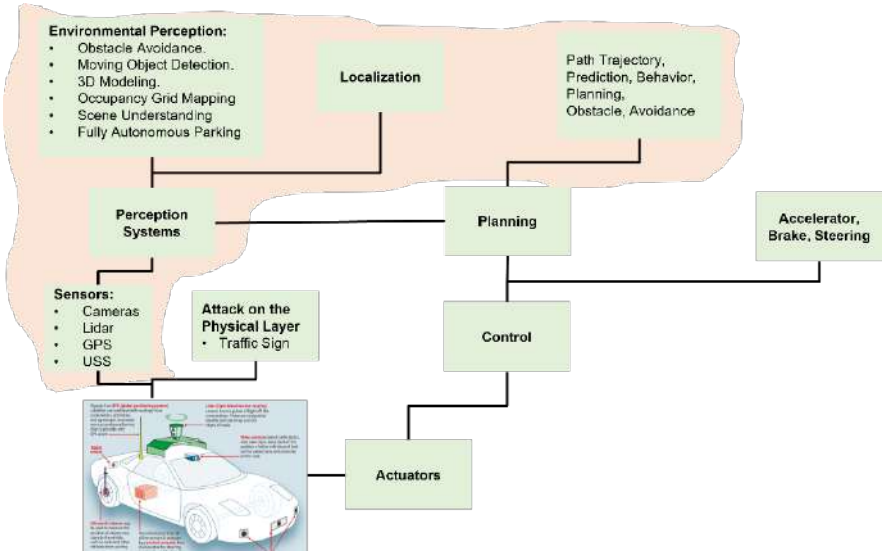


Figure 35: Perception Components involved in the cyber attacks of the automotive pilot

5.4.2 Communication Layer

As far as the communication layer is concerned, it basically offers two external interfaces used by the service stakeholders to publish and to be notified about certain events. These interfaces expose methods to interact with a set of MQTT brokers, one located on the OBU (On-Board Unit) and another one located all the way up at the Edge Node, providing enablers for communication between the CPS Layer and the Edge Layer.

The communication between the OBU and the RSU (Roadside Unit) is done using the ETSI ITS-G5 V2X communications protocol architecture using the GeoNetworking protocol, the Basic Transfer Protocol (BTP) and the IEEE 802.11p as access layer.

The IEEE 802.11p enables communications Vehicle-to-Infrastructure (V2I) and Vehicle-to-Vehicle (V2V). The testbed uses this second characteristic to send messages directly between OBUs with a broadcast approach. In order to signal if a message has to use a V2I or V2V link, the transmitter OBU uses a specific selector in the message.

All these communications are enabled by the so called V2XCOM module, which handles the vehicular communications between the involved entities. The V2XCOM module in deployed in the OBU, in the RSU and in the Edge Node.

The architecture of the module is depicted in Figure 36.

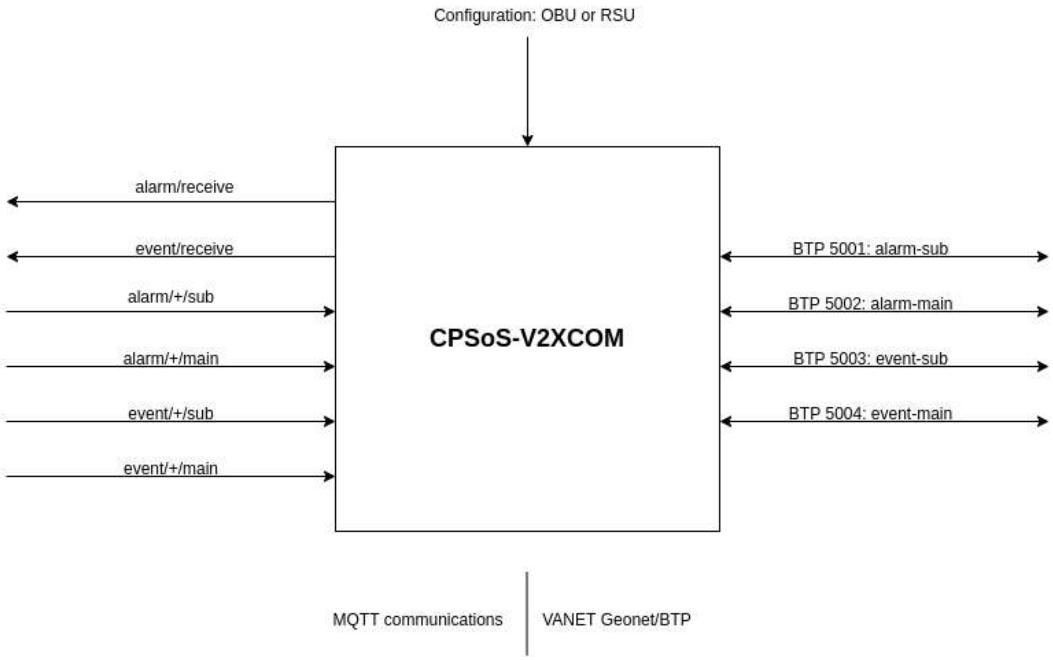


Figure 36: Architecture of the V2XCOM module

The module connects with the SRMM modules through MQTT queues.

The module publishes to the following queues:

- alarm/receive: The received alarms are published into that queue.
- event/receive: The received events are published into that queue.

The queues to subscribe depending on the needs are the following:

- alarm/+/main: The payload sent through MQTT is sent with Geonetworking Single Hop Broadcasting (SHB) and BTP 5001 port. The alarm is only received by RSU.
- alarm/+/sub: The payload sent through MQTT is sent with Geonetworking SHB and BTP 5002 port. The alarm is only received by the OBUs.
- event/+/main: The payload sent through MQTT is sent with Geonetworking SHB and BTP 5003 port. The event is only received by RSU.

- event/+/sub: The payload sent through MQTT is sent with Geonetworking SHB and BTP 5004 port. The event is only received by the OBUs.

The methods used to interact with the queues are:

Subscription to a topic:

```
mosquitto_sub -h <Ip of the broker> -p 8883 -t "<type>/receive"
```

Publishing to a topic:

```
mosquitto_pub -h <Ip of the broker> -p 8883 -t <type>/<source>/<destination>

-m "<Raw message>"
```

Where:

- type: event | alarm.
- source: SRMM name.
- destination: main | sub: with this selector the OBU can send the messages directly to the RSU (main) and from there to the Edge Node, or to broadcast (sub) to nearby OBUs.

5.4.3 Security runtime and monitoring management

As described in the previous deliverable D5.2 [25], **Security Runtime Monitoring and Management** (SRMM) system is the main component of the Cybersecurity Layer, developed by the Task 4.3 - *CPSo-SAWARE Security Runtime Monitoring and Management*. It monitors the entire system, receiving security events through the Sensor Layer, to identify possible threats and divergent behaviours arising from attacks against the system. Due to the dynamic architecture and the possible loss of communication between the different components of the system, the SRMM is a distributed component with several instances in different parts of the systems. For the use case of Connected and Autonomous Vehicles, worked out in the context of WP6, we have deployed three types of SRMMs:

- **Light SRMMs**: Deployed inside vehicles, they are the lowest in the hierarchy. They perform the local intelligence despite a limited computing capacity, allowing system to maintain service even if vehicles lose connection to the edge. Their alarms are sent to Area SRMMs.
- **Area SRMMs**: These are in a physical area, such as a road or a street. They receive security events from Area sensors and alarms from Light SRMMs. They perform the area intelligence, raising area alarms that are communicated to Global SRMM and all Light SRMMs within the area.

- **Global SRMM**: it is placed in the cloud, which has an overview of the entire system. It receives alarms from area SRMMs and global events to perform general intelligence, by broadcasting alarms to all area SRMMs.

All SRMM types have the same functionalities: receive events and alarms, raise and send alarms and execute mitigation actions. The only difference is the number of detection rules that each can manage and the target of these rules.
Communication between the different SRMMs use the MQTT protocol to send raw messages. This service is already presented previously in Section 655.4.2. To use it, the SRMMs serialize their internal structures and send to other SRMMs as raw messages.

As input method, the SRMM receives events from the *Sensor Layer* which are then correlated to generate the alarms. The system has three methods for obtaining these messages:

- *API*: SRMMs open a TCP connection, by default in port 41000, to accept messages in the format which was presented in deliverable D4.8 [26], Section 3.3.2 and is reflected here for convenience (userdata fields are optional):

```
"a": {“type”: <string>,                    "userdata5": <string>,

"date": <string>,                          "userdata6": <string>,

"device": <string>,                        "userdata7": <string>,

"interface": <string>,                     "userdata8": <string>,

"plugin_id": <integer>,                    "userdata9": <string>,

"plugin_sid": <integer>,                   "log": <string>,

"src_ip": <string>,                        "fdate": <string>,
```

Figure 37 Security Event Format

- *Agent*: A subsystem of the SRMM to monitor raw logs from rsyslog service, described in detail in deliverable D3.5 [27], in Section 5. This component parses logs from the different sensors and send security events to the above API. A plugin is defined for each sensor for which the Agent identifies their events. This plugin is a file with the log path, a regular expression for each raw log to be identified, and the relationship between the subexpressions and the fields of the security events.

- *RabbitMQ*: A message broker service to receive security events from external sensors. These messages must be in JSON format with the same structure that the API messages.

When a sequence of security events matches one of the rules that the SRMM system has configured, it raises an alarm which is the output method of the system. To display the raised alarms and their related information, the system has two methods:

- *Dashboard*: It is a web interface with a set of views that display real-time incident information, such as number of events or alarms, Fig. 9; the dashboard also allows consulting the list of alarms, Figure 38; and the details of each alarm,Figure 40.

Figure 38: Dashboard

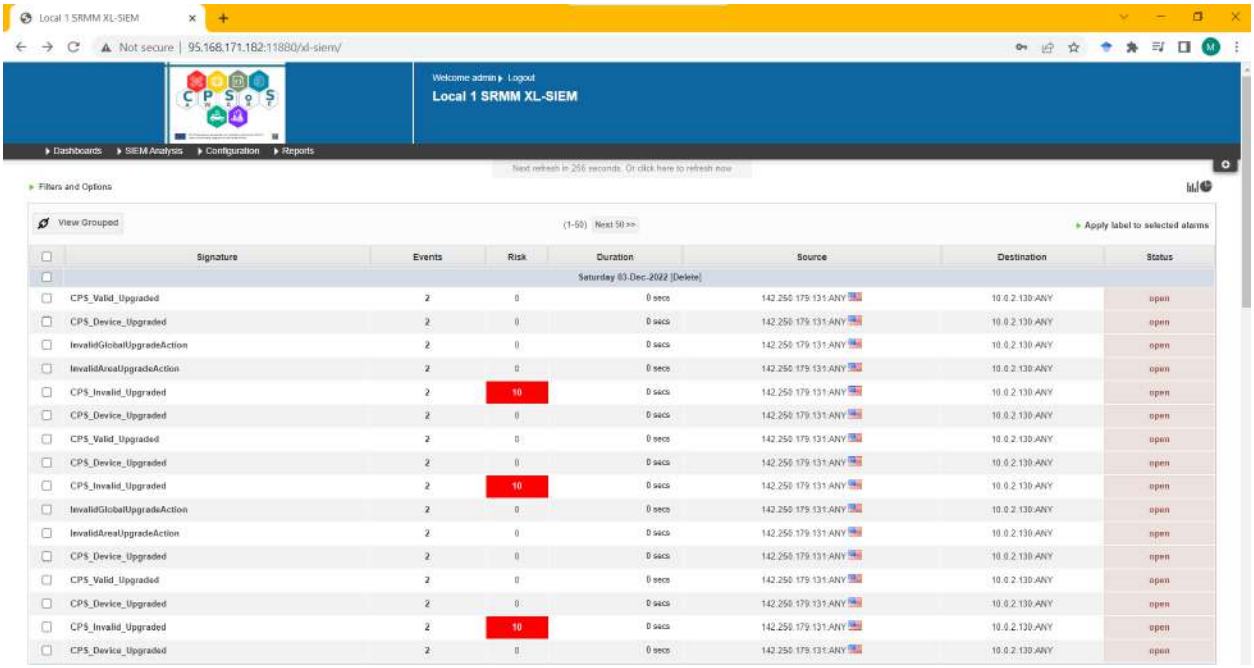


Figure 39: List of alarms

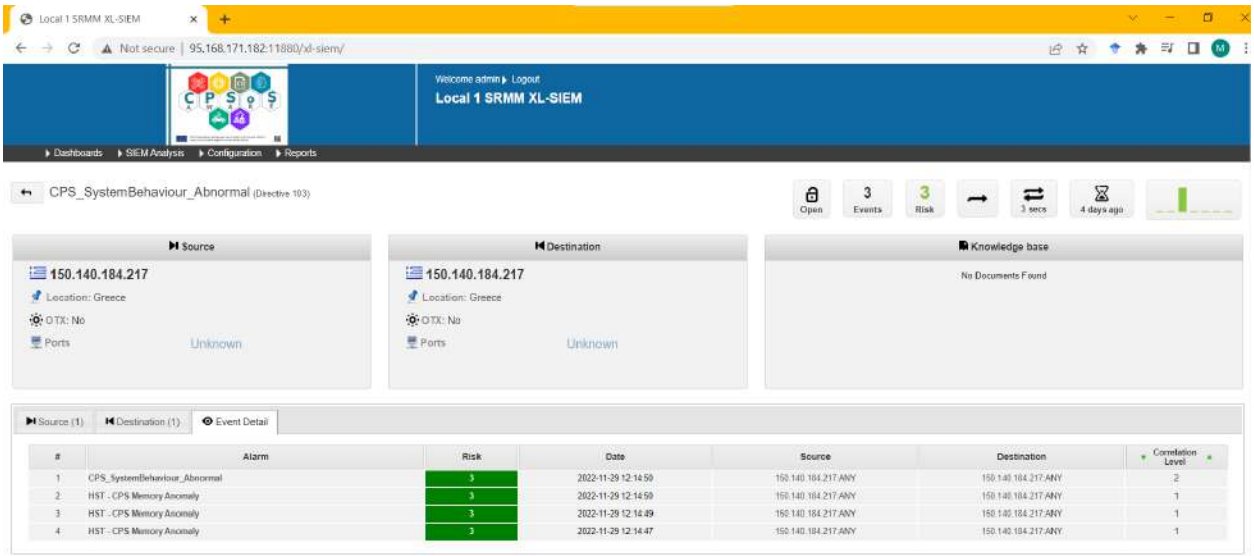


Figure 40: Detail of an alarm

- *RabbitMQ*: Where a third part service can subscribe to a queue to receive the alarm produced by SRMM. The alarm uses a JSON structure, Figure 41 (already presented in D4.8 [26], section 3.3.4).

```
{"Alarm": {
        "DST_IP_HOSTNAME": <string>,            "BACKLOG_ID": <string>,
        "RELATED_EVENTS": <string>,             "RELATED_EVENTS_INFO": {List of <Event>},
        "DST_IP": <string>,                     "PROTOCOL": <integer>,
        "PLUGIN_NAME": <string>,                "RISK": <integer>,
        "SRC_IP": <string>,                     "SRC_PORT": <integer>,
        "PRIORITY": <integer>,                  "SENSOR": <string>,
        "RELIABILITY": <integer>,               "SRC_IP_HOSTNAME": <string>,
```

Figure 41: Alarm JSON format

In addition, the SRMM can associate actions to alarms as a mitigation method. There are three types of actions: Send an *email*, Figure 42, which is a notification that could initiate a manual mitigation method because it is a user who has to receive such an email and initiate the mitigation of the attack. Open a *ticket* in another program, which may or may not initiate mitigation actions. Or execute a program or script, a *command*, Figure 43, that initiates mitigation actions automatically, without human interaction. In addition, several actions can be associated to the same alarm.

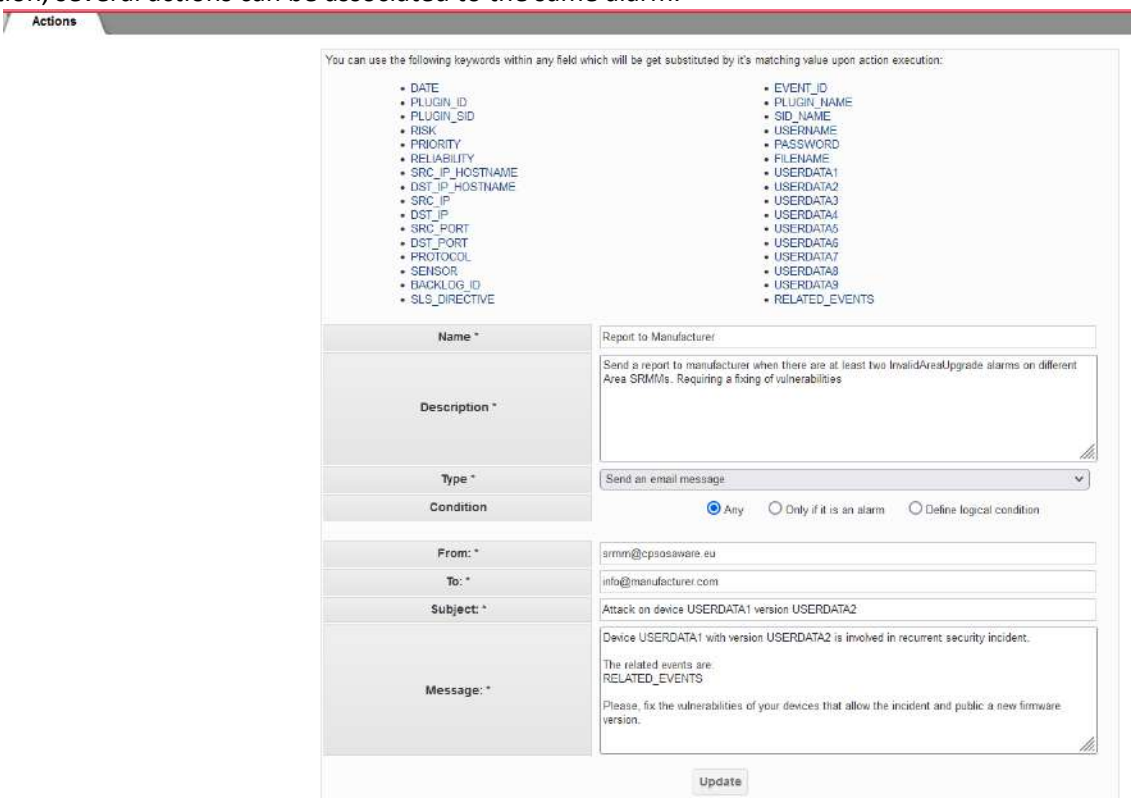


Figure 42: Email notification

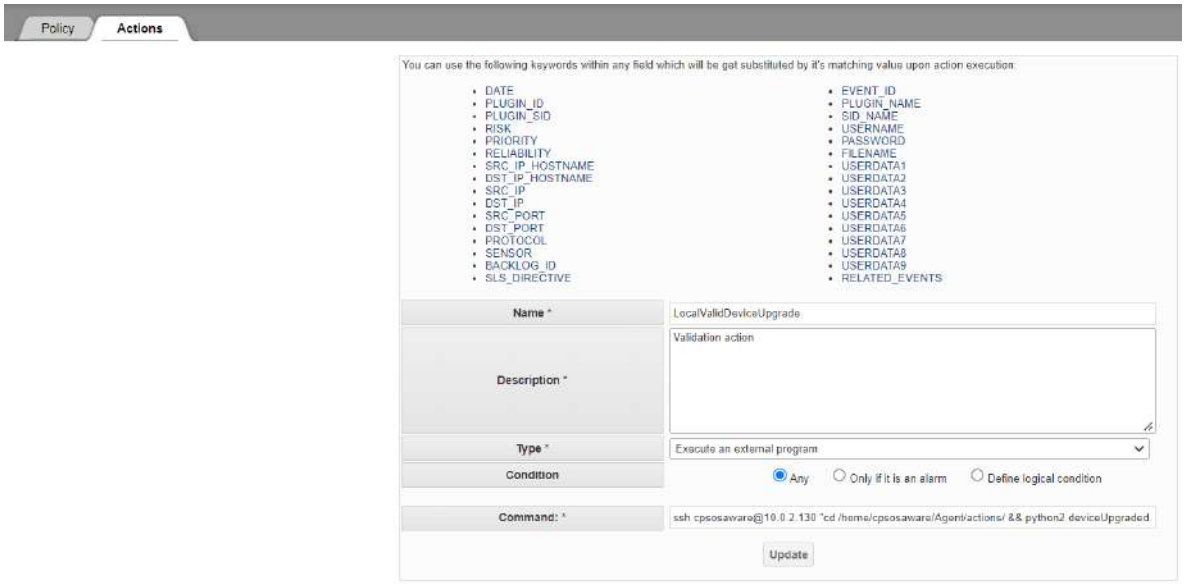


Figure 43: Action associated to an alarm

Finally, there is a configuration panel, integrated with the alarm interface, which allows creating and editing correlation rules and cross-correlation rules, registering new sensor devices, or configuring mitigation actions Figure 44.

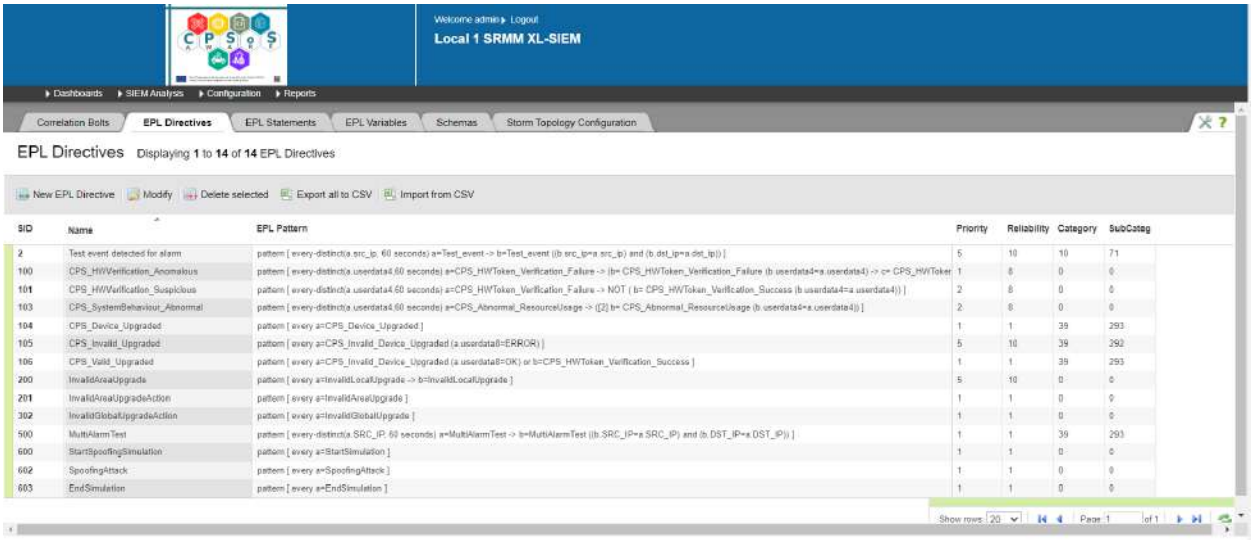


Figure 44: Rule configuration interface

5.5 Human in the loop system

Human in the loop use case concentrates on the cooperation between the vehicle systems and the human driver. In some situations, like more demanding road conditions, Autonomous Driving Systems (ADS), when incapable of handling the situation in an automatic way, need the human driver to take over control of the vehicle. However, to conduct the transition of control properly and safely, the driver must be fit to continue the task of driving. For that purpose, the Driver State Monitoring System (DSM) is a necessarily in-vehicle component that should constantly monitor the driver's availability. Another reason for DSM implementation is the detection of safety-critical states that can impact driver's capabilities, resulting in accidents. Misdirected attention and fatigue/drowsiness are among the most significant factors related to road events. As such, they were the main focus of the Human in the Loop use case and driver state monitoring in the CPSoSAWARE project.

5.5.1 Integration Environment

The DSM solution used in the project was developed by Catalink as an Android application that allows to monitor and assess the driver's state, like distraction and drowsiness, in real time. In order to do so, the application uses the front camera of the smartphone and allows for monitoring the general status of the driver and estimating driver's fatigue level throughout the drive.

The estimations are based on utilising Google's ML Kit12, a standalone library offering the possibility of on-device machine learning processing. ML Kit allows for the integration of machine learning capabilities into an application by exposing the so-called vision APIs. In the case of the DSM application, ML Kit was used to achieve a fast, efficient, and real-time face recognition and facial landmarks extraction.

The application detects the following features related to driver drowsiness and inattention and generates an alert when they are detected (as can be seen in Figure 45):

- Eye closures,
- Yawning,
- Distraction.

[12] https://developers.google.com/ml-kit/

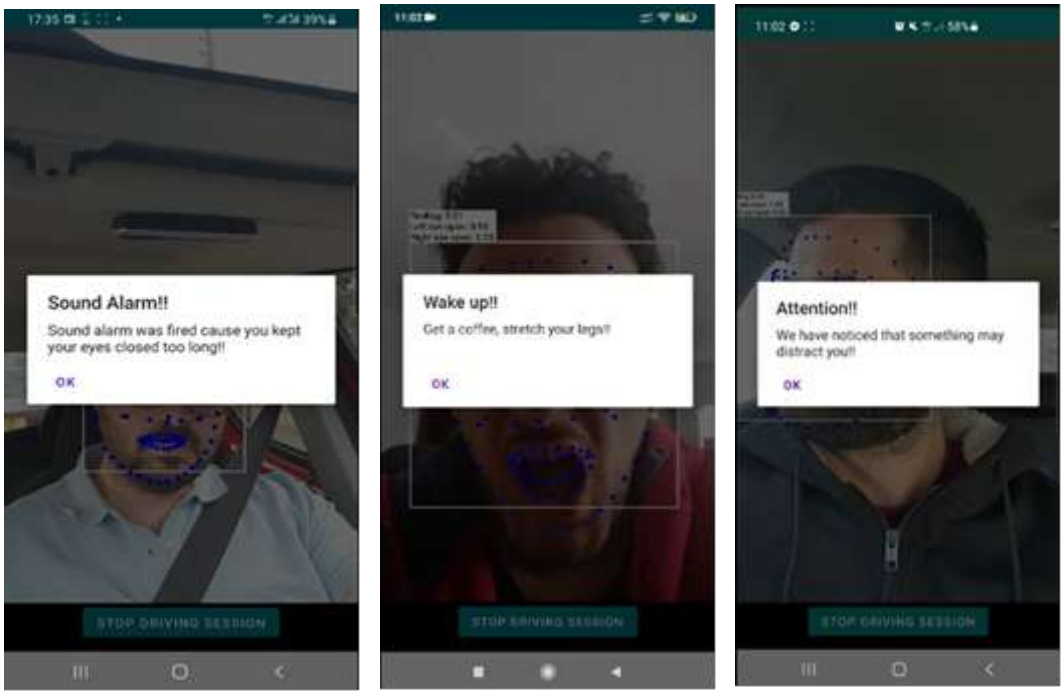


Figure 45: Result of the "Eye closure" scenario. (b) Result of the "Yawning" scenario. (c) Result of "Distraction" scenario. Source: Catalink

The application saves data in JSON files containing information about Unique Session ID, Session Timestamp, Unique Frame ID, Frame Timestamp, the frame number, the number of the detected faces in the frame, if the driver is yawning or has his/her eyes closed in this frame, if he/she is looking left/right or has his/her hands off the wheel, and finally if the alert was fired. The content of the JSON file allows for deep analysis of the application's functionality and accuracy of the state detection, as was done in the tests conducted in the project.

A detailed description of the application and its functionality can be found in Deliverable D6.2 - Small scale evaluation trials [28].

5.5.2 Runtime evaluation

In the scope of the project the overall performance of the DMS application was tested, both in terms of driver state validation and UX perspective. For that purpose, a study on 8 participants was conducted, resulting in collecting a dataset with video sequences and JSON files, utilizing two different smartphones and an interior dash camera.

The study included naturalistic driving, and static scenarios based on Euro NCAP testing recommendations for distraction testing13. The application's functionality was also tested under laboratory conditions for the detection and recognition of certain events and actions in predefined use cases in order to assess the UX-related aspects of performance.

Testing equipment included:

- Tested DMS solution,
- Heart rate monitoring (with a wearable device integrated with the DSM application),
- Contextual cameras (as an additional source of visual information),
- Test vehicles (each participant had been driving their own, or familiar to them, vehicle).

Every session involved the use of two phones, each having the DSM application installed, and with its frontal camera directed towards the driver. One of the smartphones was placed in the instrument cluster, and the other one in the participant's chosen position, which, according to personal preferences, was either a windshield under the rear mirror (Position A), or infotainment (Position B). The positions are presented in Figure 46.

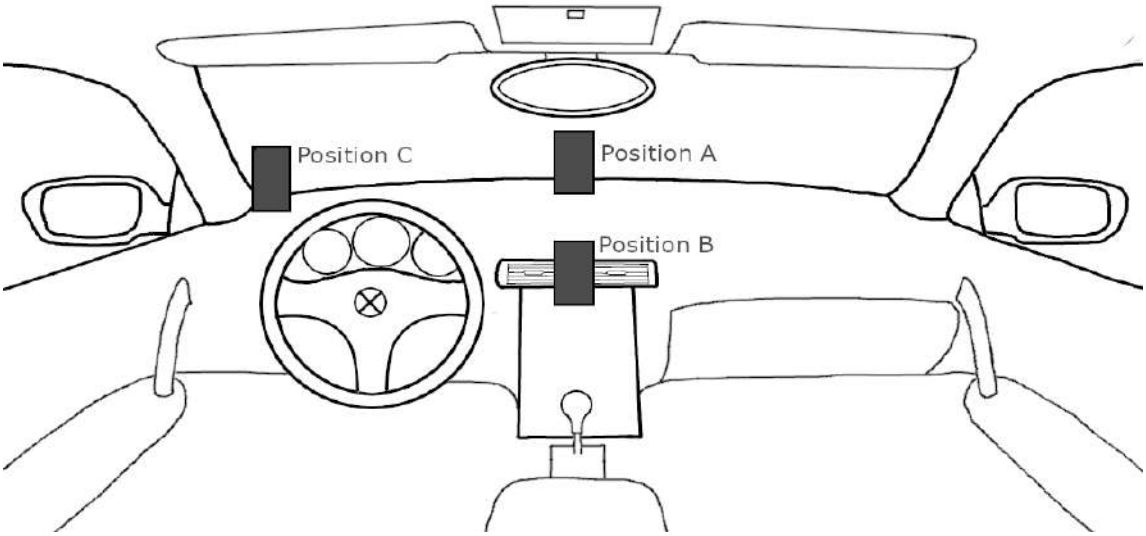


Figure 46: A schematic representation of a dashboard in an average car with phones' positioning during test drives. Source: Robotec.ai.

[13] European New Car Assessment Programme (Euro NCAP). (2022). Assessment Protocol — Safety Assist Safe Driving. Implementation 2023. Version 10.0.1.

Every participant performed a series of static testing sequences, as described in the abovementioned Euro NCAP protocol. The sequences included gazes toward different objects inside the car cabin and in the outer environment and were split into three groups: long-duration gazes, frequent and short-duration gazes, and gazes toward the phone. For every participant, the static scenarios were followed by a naturalistic driving part, which involved urban and expressway roads.

The application also underwent testing in a laboratory environment, which provided controlled conditions for testing the application's performance in specific use cases. It was meant to check how the application worked outside the context of driving and without the noise inherent to it (like drastic and rapid light changes or shocks caused by driving on uneven surfaces). Detailed descriptions of the testing methodology and a list of sequences done are provided in D6.5 [2].

A simple analysis was performed on the data to summarize the application performance. For a proper analysis, a synchronization of output JSON files, DSM application's recordings, and interior contextual camera recordings was performed. Selected variables were then processed and plotted together in order to visualize and further qualitatively discuss the obtained outcomes. The results can be found in D6.5 [2].

Following the performed analysis, the application, being now under development, can be considered as working properly in a controlled environment. The real road tests pointed out the directions in which the DSM application can be further improved to fit users' needs and convenience (e.g., the use of IR camera, if available, working in the background). However, even without these improvements, it still can be characterized as largely correctly recognizing some visual signs of drowsiness, such as closing eyes and yawning, and signs of distraction, such as turning the head away from the driving direction. As such, the tested DMS application is a valid component of the CPSoSAWARE project. After further upgrades, it also gives a great opportunity to promote the idea of driver monitoring among casual users and help to understand the mechanisms behind it.

5.5.3 Multi HW Implementation Platforms

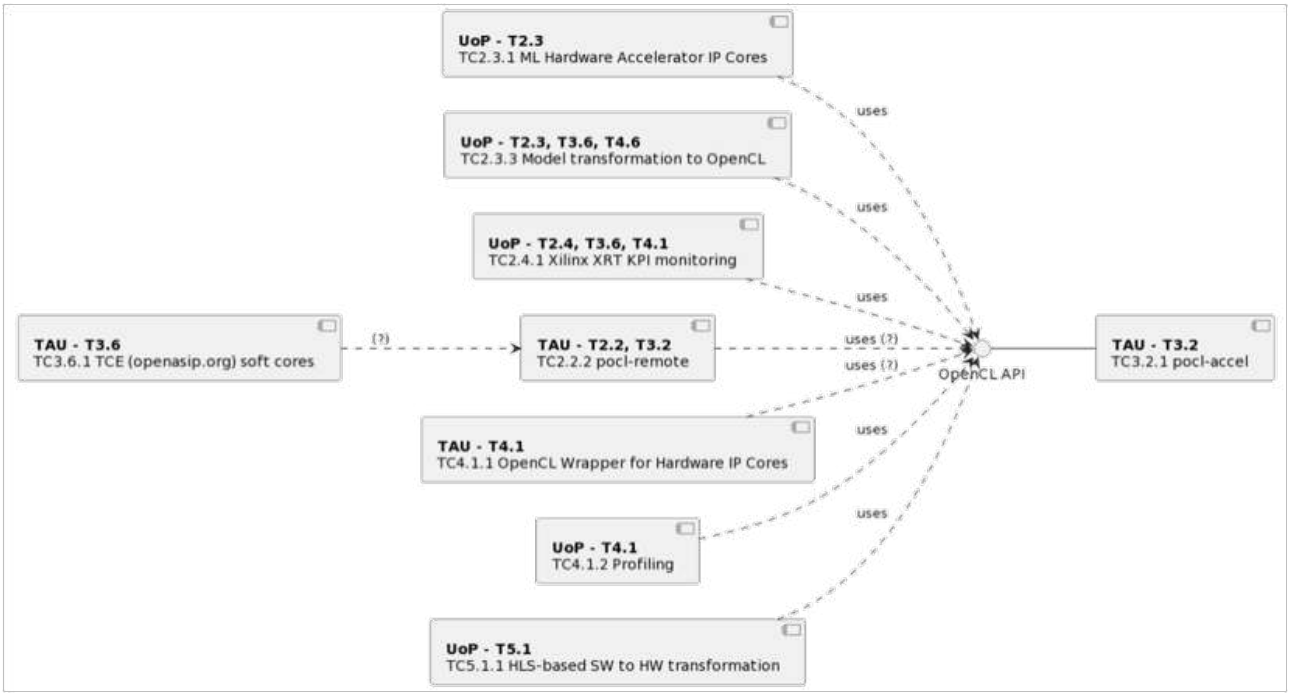


Figure 47: Multi HW implementation components

In the context of the automotive pillar, UoP has developed two use cases to demonstrate the progress on multi hardware implementations. Aconvolutional neural network (CNN) use case is based on PoCL with AlmaIF interface while the Driver State Monitoring (DSM) is based on a non-PoCL environment. The implemented components as presented in Figure 47 and their integrations are described in the following section.

5.5.3.1 CNN Use Case

For the purposes of "TC2.3.1 ML Hardware Accelerator IP Cores", the SqJ accelerator is used for the acceleration CNNs performing the image recognition task. SqJ is a convolutional and maxpool layer accelerator implemented as a single IP core described in Vitis HLS. It is implemented as a single computation engine by accelerating CNN layers in a time-shared manner. It can be called by software programs running on the operating system of the host side of a system-on-chip, which includes a processing system (e.g. an ARM CPU) and a programmable logic system (e.g. an FPGA). In this way, a software program, running on the processing system of a system-on-chip, can accelerate CNN inference by offloading the CNN convolution and max-pool operations to the programmable logic where the SqueezeJet accelerator is running.

In TC2.3.1, a software program which performs image recognition will be presented. Specifically, labelled images from the ImageNet validation dataset will be provided as input to the program and the program

will output a prediction regarding the input image. The CNN used for the purposes of this TC is the SqueezeNet v1.1 CNN.

In order to take advantage of the arbitrary precision characteristic of the FPGA, the SqueezeNet v1.1 CNN model is quantized from floating-point to 8-bit signed integers using the Ristretto tool without hurting the image recognition accuracy. In this way, the CNN model is compressed, and the CNN layer parameters can fit into the FPGA on-chip memory.

The SqueezeNet v1.1 inference will be executed on both the ARM processor and the FPGA of the Zynq system-on-chip[14] FPGA to showcase both the prediction accuracy and the acceleration achieved by using the SqJ accelerator compared to the ARM-only execution.

Input: Image, CNN weights.

Output: Classification prediction.

In the context of "TC2.3.3 Model transformation to OpenCL", high-level synthesis kernels are developed and integrated to the PoCL FPGA accelerator. These kernels perform simple operations such as vector addition and multiplication, as well as more complex ones, such as CNN convolution and fully connected layers' operations.

For the purposes of this TC2.3.3, a sign recognition CNN model has been trained and quantized and can be executed using PoCL and the PoCL FPGA accelerator. A software OpenCL program can be used to present the programmability offered by the PoCL accelerator, by executing the different CNN layers by selecting the required kernel for each CNN layer execution.

In the sign recognition CNN example, an OpenCL application uses the PoCL FPGA accelerator to execute the convolutional and the fully connected CNN layers by selecting each time the appropriate kernel. In this way, the sign recognition CNN model is transformed to an FPGA accelerated OpenCL application.

Input: sign recognition CNN model.

Output: OpenCL application.

For "TC5.1.1 HLS-based SW to HW transformation" High-Level Synthesis implementation of the AlmaIF interface allows the development of PoCL applications which offload computation kernels to the programmable logic of an FPGA system-on-chip.

[14] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

CNN computation kernels are transformed to High-Level Synthesis descriptions and are used as PoCL FPGA accelerated kernels.

For the purposes of this TC5.1.1, the SqJ accelerator is used as a PoCL kernel which accelerates CNN applications, such as the image recognition task of the ImageNet images using the SqueezeNet v1.1 CNN model.

Additionally, for this TC, the AlmaIF interface is complemented by streaming interfaces which are required for the acceleration of CNN applications. A PoCL FPGA accelerated application will be using the PoCL FPGA accelerator to accelerate the SqueezeNet v1.1 CNN layers.

Input: CNN computation kernels.

Output: High-Level Synthesis descriptions compatible with PoCL FPGA acceleration and AlmaIF interface.

5.5.3.2 DSM Use Case

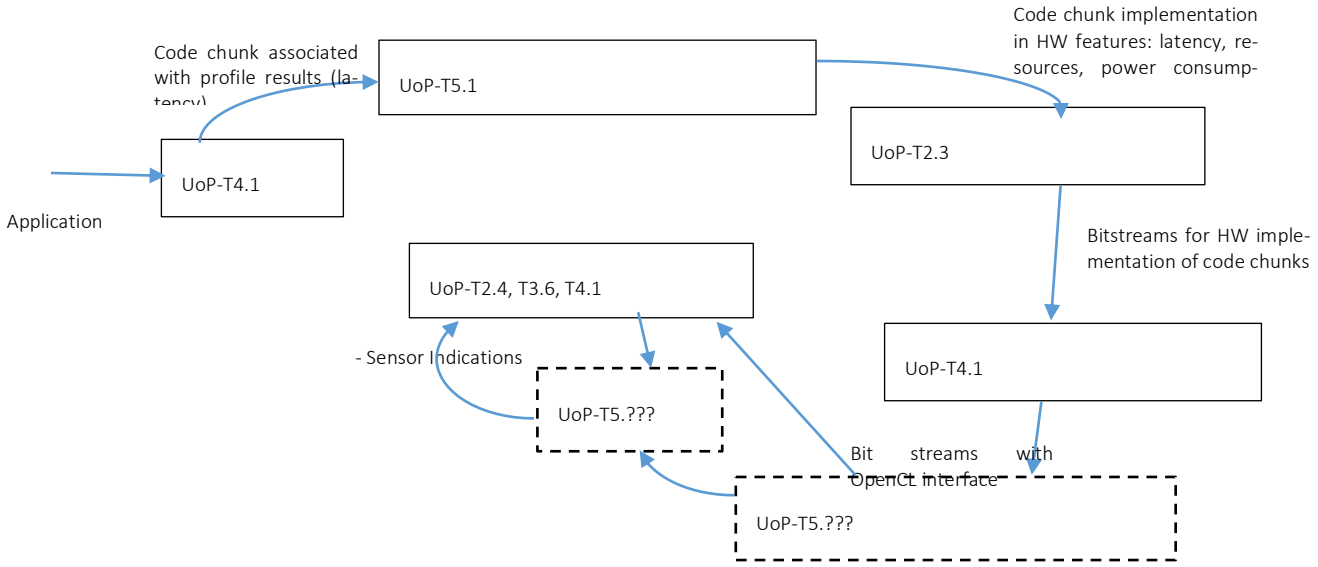


Figure 48: DSM components

The profiling of the source code of the DSM use case ("TC4.1.2 Profiling"*)* was performed at various levels (application level, HLS estimations, XRT real time profiling). The source code routines or code chunks in general served as input to the TC4.1.2 component and the measured timing information was the output for each one of the profiled routines. More specifically, in top software level, the start and end time of the following code segments was recorded to extract their average latency:

a) Frame read

b) OpenCV library call to detect the driver's face bounding box

c) Bounding box coherency rules

d) Landmark position prediction (Kernel_predict() routine)

   - Argument passing
   - Similarity Transform
   - Read pixel intensities
   - ERT cascade stage loop (ERT regression trees loop (Predict_trees() routine) / Single regression tree processing)

e) Return of the landmark position correction factors

f) Update landmark position with correction factors

g) Final landmark position coherency rules

h) Update driver drowsiness parameters (EAR/MAR, PERCLOS and counted yawnings, sleepy eye blinks)

i) Draw output frame with updated landmarks and information about drowsiness parameters

The top-level application repeats the following steps: it retrieves a frame from the opened input stream (a). Then, OpenCV library is called to find the bounding box coordinates where the face exists (b). In the final version of the DSM module developed by UoP, coherency rules on the bounding box are applied as a next step (c) to reject false OpenCV bounding box estimations (e.g., bounding box of invalid size or in invalid position). Then, the landmark prediction function takes place in (d). In the original DEST source code, the ERT model parameter values were accessed when needed. In the hardware-friendly version that the UoP ported to Ubuntu environment all the functionality of (d) was included in the Kernel_predict() routine that needs all the ERT model parameters to be passed as arguments. This function also incorporates the following subfunctions: Similarity Transform warps the landmark shape to match the specific face in the bounding box and then, the intensities of specific frame reference pixels are read before the cascade stages of the ERT model are executed to correct the mean landmark positions that re read from the ERT model. In each cascade stage, many regression trees are visited and in each regression tree, the intensity of a pair of reference pixels is compared to decide the next node in the tree that will be followed and the next pair of pixels that will be compared. When a leaf is reached in the regression tree the corresponding landmark position correction factor vector is selected.

In the original open-source version of the Deformable Shape Tracking (DEST) package video tracking application, that was initially ported to Ubuntu OS in order to perform the top-level profiling, the functionality of code segments (c), (g) did not exist while in (h) only landmarks were drawn. Moreover, the functionality of (d) and its subroutines was entirely based on Eigen math library calls. The profiling of the original DEST video tracking application highlighted that 80% of the frame processing time was spent on (d). The absolute latency of (d) as measured on an Intel i5 platform was 116ms. This fact forced UoP to replace time consuming Eigen library calls with fast C code that could also be ported to reconfigurable hardware.

This Eigen library call replacement in (d) accelerated the frame processing latency by more than 240 times since it was reduced below 0.5usec on the same platform. Although UoP initially ported all the functionality of the Kernel_predict() routine (d) in hardware, resource estimations and a new profiling of the subroutines of (d) showed again that 80% of the Kernel_predict() latency was spent on Predict_trees() which is called within Kernel_predict() once for each ERT cascade stage. Therefore, the porting of Predict_trees() in hardware was selected leaving in software the code of the routines that performed the Similarity Transform and the pixel intensities reading since these routines are complicated but do not have iterative operations and thus, they consume a large number of resources if implemented in hardware.

The latency, resources and power consumption of the hardware implementations of the Kernel_predict() and Predict_trees() hardware kernels depends on various parameters of the ERT model (number of cascade stages, regressor trees and tree depth, reference pixels, etc). UoP has trained a number of different ERT models that differ in these parameters and are appropriate for different environmental conditions. Each one of these models has a different hardware kernel counterpart to implement its time consuming operations such as Kernel_predict() and Predict_trees(). The profiling of these alternative hardware kernel implementations has been performed in Vitis High Level Synthesis (HLS) tool, to accurately estimate the minimum and maximum latency in clock cycles, the resources allocated for these implementations and their power consumption.

Input: DSM C code chunks.

Output: Utilisation, measured latency of the code chunks, required memory and power consumption, resources (if implemented in hardware).

The output of the TC4.1.2 Profiling component is the list of the code chunks that were used as input to this module accompanied with a latency characterization. Specifically, the name of a subroutine can be accompanied with one or more latency metrics. As described in TC4.1.2 these latency metrics can include absolute time as was the 116ms required by the initial landmark prediction routine of the DEST video tracking application or a percentage (80%) of the time required e.g., for processing a single frame. Based on these latency metrics the routines that are candidates for hardware implementation such as the Kernel_predict() or the Predict_trees() are selected. The system starts with an all-software solution and all the code is developed e.g., in C/C++. The hardware candidates, however, should be described in a C/C++ style that is friendly to hardware synthesis. The component that transforms a C/C++ description in

hardware is the "TC5.1.1 HLS-based SW to HW transformation". In the DSM module developed by UoP, Xilinx Vitis HLS and Vivado HLS tools were employed to perform the hardware synthesis. Since various alternatives for the hardware implementation of a process had to be investigated the output of TC5.1.1 is the list of (code chunk, latency) pairs extended with other features that can be estimated by HLS such as the resources allocated by the implementation of each chunk and its power consumption.

Input: List of (code chunk, latency) pairs

Output: list of (code chunk, latency, resources, power consumption) tuples

The HLS tool of TC5.1.1 except from estimating resources, power and latency of the routines that are candidates for hardware acceleration, also synthesizes the actual hardware IP cores. These cores are the implementation of the most computationally intensive parts of the ERT model processing. As described in TC4.1.2, several alternatives of the ERT Machine Learning model have been trained in the DSM use case. These models are accelerated by hardware IP cores that have different number of resources, power consumption, latency and achieve a specific accuracy under various environmental conditions (driver gender, features, light exposure of the camera images, etc). In other words, a different tradeoff is achieved between speed and cost in each one of the alternative ERT models and their corresponding hardware IP cores. Each hardware IP core is represented by an FPGA configuration bitstream in Xilinx Vitis, and this is the output of the "TC2.3.1 ML HW Acceleration IP cores" component

Input: list of (code chunk, latency, resources, power consumption) tuples

Output: Bitstreams for HW implementation of code chunks

The bitstreams generated by the TC2.3.1 component must be invoked through OpenCL commands. These commands initially detect the hardware platform i.e., the FPGA ZCU102 board of the DSM module and then initialize a command queue. The kernel arguments are then loaded on this queue and the kernel is initiated. The command queue can be blocked by OpenCL waiting for the kernel results to be available and transferred back to the top-level software that called the hardware kernel. The OpenCL wrapper ("TC4.1.1 OpenCL Wrapper for HW Cores") should be aware of the routine name of the kernel that is implemented in hardware as well as its input/output arguments and types.

Concerning the specific DSM application, the Kernel_predict() routine required 21 arguments with 14 of them being large buffers since all the ERT model values should be available to the kernel. These buffers are prepared in an initialization routine (predict_prepare()). The Predict_trees() kernel requires only 4 medium sized buffers since it is called in each cascade stage with only the arguments that concern this specific stage. In an accelerated version of the Predic_trees() kernel, 3 of the 4 buffer arguments are split in pairs of buffers with half size, in order to increase parallelism in the processing and reduce the transfer latency.

The full list of the kernel arguments used in the DSM module is the following:

```
void predict_kernel(int kr_maxTreeSizes,
                    int kr_numCascades,                  // Input. Number of cascade stages
                    int kr_meanShape_cols,               // Input. Number of landmarks
                    int kr_irows,                        // Input. Rows of the input frame
                    int kr_icols,                        // Input. Cols of the input frame
                    unsigned char *kr_img,               // Input. Start pointer of the input frame
                    int * kr_treeSizes,                  // Input. Size of the regressor trees
                    float * kr_meanResiduals,            // Input+Ouput. Updated mean residuals (accumulated correction factors)
                    int * kr_Global_node_split1,         // Input. Next node of the regressor tree (left direction)
                    int* kr_Global_node_split2,          // Input. Next node of the regressor tree (right direction)
                    float* kr_Global_node_thres,         // Input. Thresholds in regressor tree nodes to select left or right direction)
                    float*  kr_Global_node_mean,         // Input. Correction factors for the landmark coordinates
                    float* kr_learningRates,             // Input. Forgetting factor (can be constant)
                    float * kr_meanShapes,               // Input+Output. Updated landmark coordinates
                    float *kr_shapeRelativePixelCoordinatesX,     // Input. Reference pixels coordinate X
                    float *kr_shapeRelativePixelCoordinatesY,     // Input. Reference pixels coordinate Y
                    int* kr_closestShapeLandmarks,       // Input. The closest landmark to each reference pixel
                    int* kr_numCoordsV,                  // Input, Number of reference pixels
                    float* kr_shapeToImage, // Input. Warped mean shape
                    float* kr_estimate,                  // Input+Output. Intermediate shape coordinates
                    float* kr_fnal)                              // Output. Final shape coordinates


Predict_trees() with single port arguments:
          void predict_trees(
                    float*          kr_sr_tg,    // Input+Ouput. Updated mean residuals (accumulated correction factors)
                    int             kr_tg_len,   // Input. Number of trees
                    int             kr_casc,             // Input. Cascade stage
```

```
                    u_split_t*    kr_split1,    // Input. Next node of the regressor tree (left direction)

                    u_split_t*    kr_split2,    // Input. Next node of the regressor tree (right direction)

                    u_intensities_t*          kr_intensities,          // Input. Reference pixel intensities

                    u_thres_t*   kr_node_thres,          // Input. Thresholds in regressor tree nodes to select left or right direction)

                    float*                    kr_node_mean) // Input. Correction factors for the landmark coordinates



Predict_trees() with double port per argument:

          void predict_trees(

                    float*                              kr_sr_tg,

                    int                                 kr_tg_len,

                    int*                                kr_casc,

                    u_split_half_t*                     kr_split_h1_1,

                    u_split_half_t*                     kr_split_h2_1,

                    u_split_half_t*                     kr_split_h1_2,

                    u_split_half_t*                     kr_split_h2_2,

                    u_intensities_t*                    kr_intensities,

                    u_thres_half_t*                     kr_node_thres_h1,

                    u_thres_half_t*                     kr_node_thres_h2,

                    float*                              kr_node_mean)
```

The bitstreams of the kernels with their OpenCL interface along with the corresponding ERT models populate a library that can be accessed locally or remotely from the DSM application to switch between ERT models and their supporting hardware kernels according to the environmental conditions. For example different (ERT model, hardware IP core) pairs may be appropriate for male or female driver, for daytime or nighttime conditions, etc.

Input: Bitstreams for HW implementations of Kernel_predict() or Predict_trees() kernels. The input arguments of these kernels are listed above

Output: Bitstreams for HW implementations of Kernel_predict() or Predict_trees() kernels with OpenCL interface. The output arguments of these kernels are listed above.

Xilinx Real Time (XRT) is a library that can be used to monitor several parameters of the hardware IP components that have been loaded on the Programmable Logic (PL) of the FPGA, in real time ("TC2.4.1 Xilinx XRT KPI monitoring"). In the DSM module XRT has been employed to dynamically change the hardware functions implemented by the PL module. More specifically, alternative bitstreams can be loaded dynamically to switch between the different IP hardware cores that support the different ERT models. For instance, if the environmental conditions indicate that a different ERT model than the one already loaded, can achieve a better tradeoff between accuracy, speed and power, XRT can load it dynamically reconfiguring the PL module while the software will load the corresponding ERT model. One option is to have the multiple IP cores that support the different ERT models built in a single bitsream and call the names of different functions incorporated in this bitstream, according to the IP core that has to be executed. A more flexible option is to have one IP core in each bitstream and load the appropriate bitstream (from a local source or a remote library) according to the IP core that has to be used.

Input: Sensor Indications (e.g., about night or day), HW kernel real time information, IP cores and their features (latency, power, accuracy)

Output: Bitstreams of the IP cores downloaded to the target platform.

6 Human-Robot Interaction in Manufacturing Environment

The manufacturing use-case comprises a series of features integrated to define and support several informative messages with the aim to support an Augmented Reality training on the job programmed on a HoloLens device. The augmented reality (AR) application is designed to support training and operations in case variants in the assembly process are present ensuring thus a proper operation continuum because of the optimized, safe, and resilient network of information flowing in the work cell system.

In the reference use-case the information provided to the operator during the training phases are:

- Functional operation for the assembly tasks: support to operations, localization of the work-scene, images of relevant parts, actions and so on
- Robot coordination information: information to the operator related to the safe moments of interaction with the robot according to Human Robot Collaboration Standards
- Anthropometric parameters: height of the operator used by the robot to place the gripper in the ergonomically optimized height depending on the specific operator; the information is also provided to the operator in the HoloLens representation
- Ergonomics alert: an alert derived from a real-time RULA index - Rapid Upper Limb Assessment - evaluation from the camera 3D pose landmarks
- Operator's State Monitoring parameters
    - Dizziness: calculated from the facial expressions from the operator
    - Heartbeat: acquired by a smartwatch and used to provide an indication of stress, fatigue or other physiological parameters
    - Body Thermal field maximum evaluation: indicator of fatigue, stress, tiredness, risk of faint, fever…

It is important to underline that in the CPSoSAWARE project the system was developed upon potentially important features, connectivity and functionality were tested, but all the physiological and ergonomics indicators used, though meaningful and representing the intended use were simply taken from literature without dedicated developments as a demonstrator of the aimed functionality. For example, the values calculated from the RULA index in dynamic way have no direct and sure meaning (RULA index is calculated as an integral of a complete operation and not as a dynamically updated parameter). For this reason, the results represented have no direct relevance and were not investigated experimentally to evaluate the effective and more correct thresholds or trends of use. The functionality and potential connectivity were, on the other end evaluated).

Figure 49: Hololens representation of main information provided to the operator in case of training

Figure 49 provides a snapshot of an assembly image with most of the above listed information detailed. More details on the AR application and its information will be provided in D6.5 [2].

The different CPSs have been connected according to the current high-level architecture Figure 50.

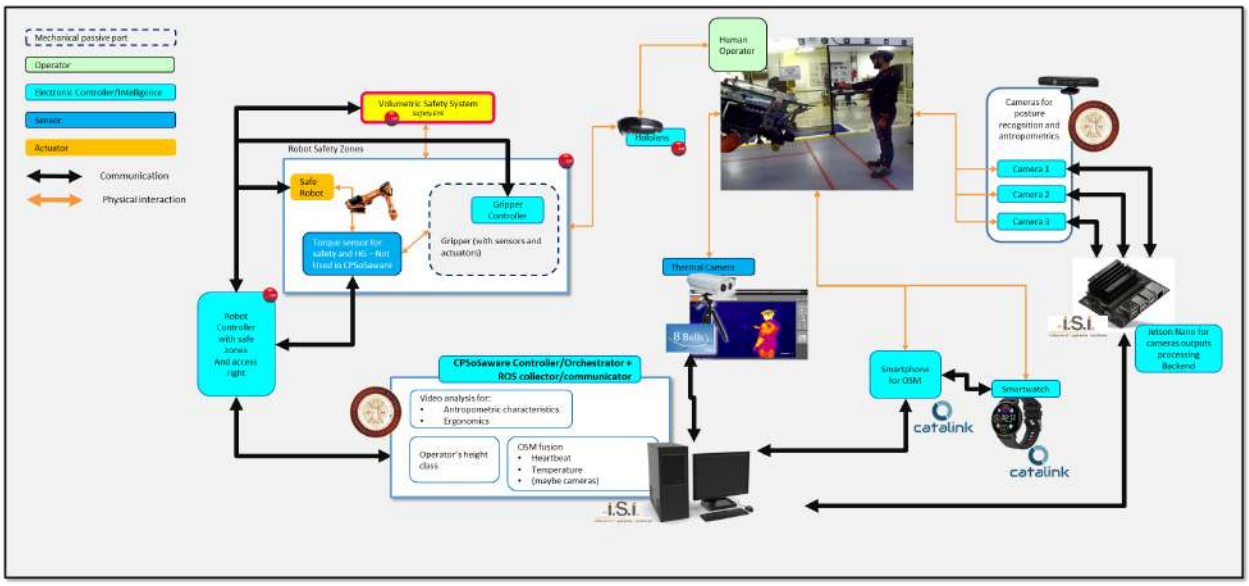


Figure 50: Systems connected in the Manufacturing use-case

The various systems are connected and communicate through a ROS system which is coordinating messages through a proper set of Topics and subscriptions

In details the involved components are described in the following sections.

6.1 ROS communication

For the seamless integration of the components and the algorithms described above, we chose the Robotic Operating System (ROS) as our middleware. ROS is an open-source framework that contains a set of tools and libraries for developing distributed applications. Programs run on isolated nodes that can communicate using a publish-subscribe model. We have developed one ROS node for every camera used in the experiment . Each node captures a frame of the camera with a frequency of 10Hz, extracts and processes the landmarks, and finally publishes them to a relevant ROS topic. The decentralized approach of ROS enables us to connect the cameras to different physical machines, like the Jetson TX2 embedded device. A different node subscribes to the topics published by the cameras and extracts an optimal landmark set which will be used subsequently for the calculation of the posture of the operator.

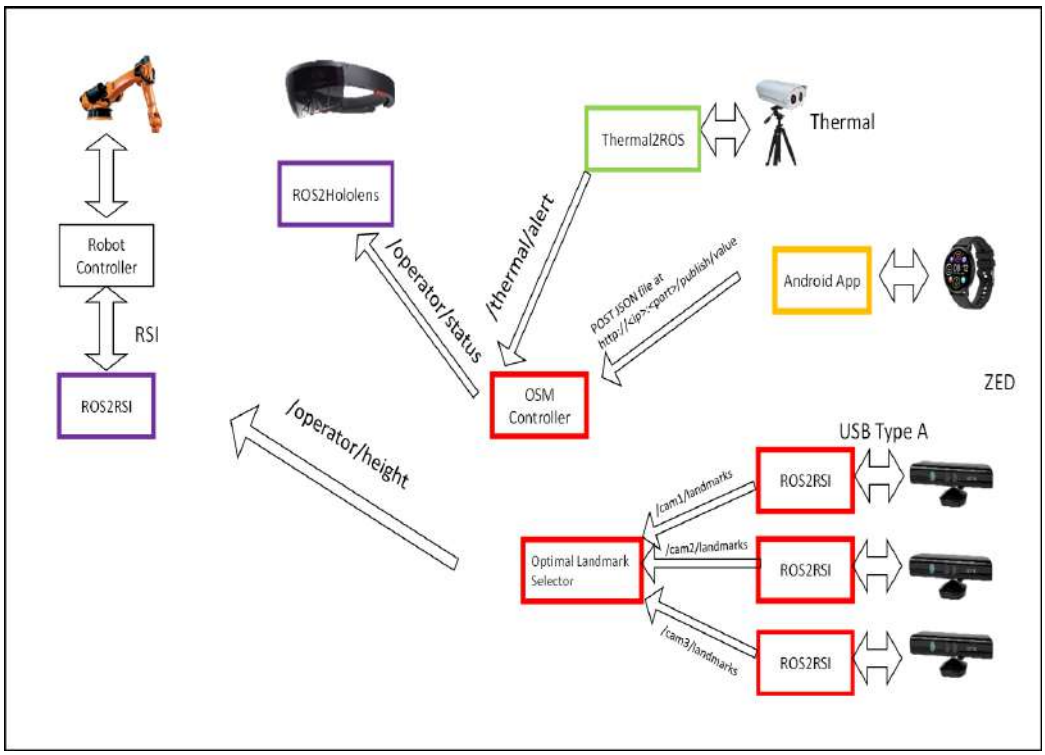


Figure 51: A view that focuses on the ROS interactions

The following table (Table 9) summarized all the information regarding the exchanged messages, their type, as well as the involving entities.

Table 9 ROS exchanged messages in manufacturing use case

Publisher	Subscriber	TYpe of interface	Url/ROS Topic	Data Type	Description
Android app	OSMCon-troller	REST	http://<ip>:<port>/publish/value	POST a JSON file	Example: { "operator_state" : "alerted/drowsy", "heart_rate_value" : 75.0, "timestamp": "2022-03-21T12:46:13.101Z" }

Thermal2ROS	OSMCon-troller	ROS	/thermal/alert	Int16	An Alert which indicates the operator's status. Possi-ble values: 1-> Hypother-mia, 2-> Normal, 3-> Signs of fatigue, 4->Hyperther-mia
zed_ros	Optimal Landmark Selector	ROS	/cam#/Landmarks	Point-Cloud2	Landmarks detected by the zed camera
Optimal Landmark Se-lector	ROS2RSI	ROS	/operator	Int16	The class of the height of the operator as calculated by the fusion of the three sets of landmarks
ROS2RSI	Robot Con-troller	RSI			Set the height of the opera-tor
OSM Controller	ROS2Ho-lolens	ROS	/osm/status	String	The status of the operator as resulted by the fusion of the inputs by the thermal camera and the smart-watch.

OSM Controller	ROS2Ho-lolens	ROS	/osm	String	The status of the operator as resulted by the fusion of the inputs by the thermal camera and the smart-watch. Example: {
					
					"operator_state" : "alerted or drowsy",
					
					"heart_rate_value" : 75.0,
					
					"thermal_alert": 1
					
					}

OSM Controller	ROS2Ho-lolens	ROS	/ergonomics	String	An alertr raised if the pos-ture of the operator is not good.

6.2 XR Training: Tools and system architecture

At the design phase of a new project that includes operators and human-robot interaction, one of the most important aspects to define are the Human Machine Interfaces (HMI). The HMI define the interaction between operators and robots in a workplace; aim of designer is define the best choice for specific use case. In addition to the more traditional interfaces with monitor and physical dashboard, in smart factory perspective, we can introduce innovative interfaces taken advantage from IoT solution and interconnection devices.

HMI is the main contact point between human and machines, so it must be useful and with low cognitive load for users; an easy to use and easy to learn HMI could support operator on its assigned tasks with minimum effort. Definition of the HMI is critical both to achieve advanced functionality and to access cognitive overload. Moreover, operator's work environment is "noisy", with many distracting elements.

How is possible to see in literature, there are mainly three types of interfaces:

- Physical
- Natural
- Graphical

Physical interfaces include an interaction with elements such as buttons and handles; this type of interface is characterized by possibility to interact with few physical devices with few information and low level of complexity. Generally, this interaction elements have a visual indicator that suggest operator the available solutions. Natural interface is a category of HMI refers to the modes of interaction typical of human communication; this category includes Voice (interfaces that uses natural language to interact with machines), Gestures (interfaces that uses a library of few gestures, each one associated to a command) and Haptic (interfaces that uses a library of signals associated with a few simple communications in feedbacks receiving).

Graphical interfaces are the more used interfaces in industrial environment. Possibility to receive and to send a lot of information with minimum effort represents a very strength point. This category includes a lot of solutions, that differs by device type, such as monitors, touch devices (tablets, smartphone, etc.) or by technology (Visual, Augmented/Mixed reality).

To pursue our objective in project, to find a solution for training and supporting operators in a guided assembly process scenario, we identified a solution with a low cognitive load to allow operators to work on tasks without moving away from operative workspace and without use devices that make difficult the assembly process.

On this scenario, we identified holographic interface solution to support operator on its tasks, using an extended reality (XR) application with a Head Mounted see-through Display (HMD), that allows operator to work with hands free, visualizing all the information required with virtual augmented reality objects and an interactable Graphical User Interface (GUI) into a see-through display. Extended Reality represents part of Reality Spectrum, as described below:

- Augmented Reality (AR): overlay digital 3D contents onto the real world
- Mixed Reality (MR): it is possible to have interaction between real world and overlayed information (AR)
- Virtual Reality (VR): user is immersed into an all-digital world

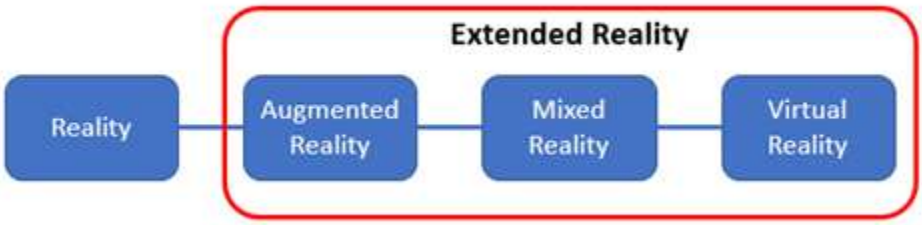


Figure 52: Reality spectrum

As follows, we present the tools and devices identified and used in planning and development phases of the XR holographic training sub-system into automotive UC. In particular, the Microsoft Hololens 2 HMD, the game engine and editor software Unity and the ROS libraries' set for communication.

6.2.1 Hololens 2

The Microsoft Hololens is a wearable head mounted see-through display device that allow to visualize the physical environment augmented with 3D digital objects. The aim is to integrate AR/MR technologies into the workspace to improve the human operators' conditions, growing the productivity and create new opportunities in interaction with innovative HMI.

The use of smartglasses like Hololens allow to receive and elaborate heterogeneous data and visualize the results and the GUI through the see-through display, working with hands free without controllers and joypads. Furthermore, using a see-through display and eXtended Reality technologies, release the sense of sickness into operators, very usual with HMD not see-through and Immersive Reality technologies.

The Microsoft HoloLens 2 is a wireless device that could executes customized apps based on business needs, helping workers in training, learning, communication, and collaborative processes. As we said, the use of Mixed reality technology allows us to interact with the physical world though digital objects, allowing for extensive interaction between humans, computers, and the environment. This new approach is based on advanced systems in computer vision and see-through display technology, graphics and CPU

processing power, input systems through gestures or voice. The application of mixed reality has led to the inclusion of environmental input, spatial sound, location and positioning in both real and virtual spaces.

Figure 53: Microsoft Hololens 2

Microsoft HoloLens 2 is equipped with:

- Display:
  - Optics: See-through holographic lenses
  - Resolution: 2K 3:2 (2048x1080px per eye)
  - Density: >2.5K radiants
  - FOV: 52 deg.
- Sensors:
  - Head tracking: 4 light cameras
  - Eye tracking: 2 Infrared cameras
  - Depth: 1MP depth sensor: operating with two modes:
    - AHAT (Articuled Hand Traking): depth taking through high framerate (45fps)
    - Long Throw: depth taking with low framerate (1-5 fps)
  - Inertial Measurement Unit (IMU):
    - Accelerometer: used for calculating linear acceleration along X, Y or Z axes
    - Gyroscope: used for rotations
    - Magnetometer: used for the absolute orientation
  - Camera: 8MP, 1080p@30fps video
- Computing:
  - CPU: Qualcomm Snapdragon 850
  - RAM: 4GB LPDDR4x
  - Storage: 64GB (flash memory)
- Connectivity:
  - WiFi: 802.11ac
  - Bluetooth: v5.0
  - USB: type-C (data and power supply)

Figure 54: Microsoft Hololens 2: components

The system brings a series of improvements compared to the first-generation device, including a customized DNN virtual content that allows a complete virtualization of all the information accessible through the device. The presence of a custom holographic processing unit of second generation (HPU 2.0), enables low-power real-time computer vision (CV) that runs all CV algorithms on the device (head tracking, hand tracking, gaze tracking, spatial mapping, etc.) and hosts the DNN core. The CPU of the SoC remains fully available for applications.

Hololens 2 allows also a fully articulated hand tracking and gaze tracking mechanism, improving older functionalities. Operator could use both hands (and custom application could take advantage from this, making new mechanisms of interaction) and interact with a very complete set of new gestures (press, grab, direct manipulation, touch interaction and scrolling)

These features allow us to use the device in different application fields, like assembly guided assistance, configuration of robotic flow operations, inspection of operator's workstations to visualize data info and status. In fact, on an assembly guided application, operators could see through the HMD all the information required to complete the process, such as a digital animated ghost of the operation, a step-by-step GUI with info and images, and a full operative sheet.

As reported into literature, using an HMD could involve a bigger cognitive load for operators, mainly due to a limited Field Of View (FOV) of device (compared with human natural FOV) and compared with use of paper instruction. But if we take into consideration a different application field, in which the use of device is limited in time, just for training, or just for visualize info of status system without compromise the assembly process, or another, a short and very specific guided activity that only a specialized operator could carry on without instructions, a bigger cognitive load with a smaller error rate could be a useful solution.

For the aim of this current project, a device like Microsoft Hololens is the right choice in term of flexibility (covering Reality spectrum, from Augmented to Virtual) and balancing between cognitive load and usability.

Hololens 2 will hosts our UWP (Universal Windows Platform) holographic application to support operators in training of a new assembly workstation or to guide operators with a step-by-step GUI for a dynamic assembly process that requires a specialized operator or detailed instruction.

6.2.2 Unity editor

Unity is a development platform that allows to create application for different OS and device, in which is required the development of virtual 3D scene and the use of a real-time game engine. The platform allows us to develop UWP applications able to run on device with ARM64 CPU and using scripting backend API IL2CPP. Core of the platform is the game engine (for physics and interaction digital-digital and digital-real), but the editor is the interface that users use to write codes and design scenarios.

The Editor is the tool used for design and develop the 3D scenes and represents the integrated environment used for building the application. Its interface could be customized; in our case, we preferred an easy-to-use interface in which the IDE window is divided into 4 macro-areas, ad showed below. In particular:

- The ***Hierarchy*** tab on the left, is used to show the hierarchical representation of all the components (GameObject) in the 3D scene also in relation between each other, in a parent-child tree schema.
- The ***View*** tab on the middle, is the 3D scene in developing; we had a Scene Point of View (POV), in which we can insert and manipulate the components in a 3D environment (X, Y, Z space), and the Game tab, in which we can see the preview of application result
- The ***Inspector*** tab on the right, is used to visualize and modify every single property and parameter of a GameObject, adding new components or removing old; from inspector tab is possible to set a GameObject as active/not active in hierarchy.
- The lower space of interface is occupied by ***Project*** and ***Console*** tabs. The project tab is the visualization of the root folder of the project in a tree-hierarchy form, in which we can interact with Assets, libraries, scripts, templates, materials and prefabs (GameObject saved for in a specific configuration); the console tab is very useful in development phase to follow the debug.

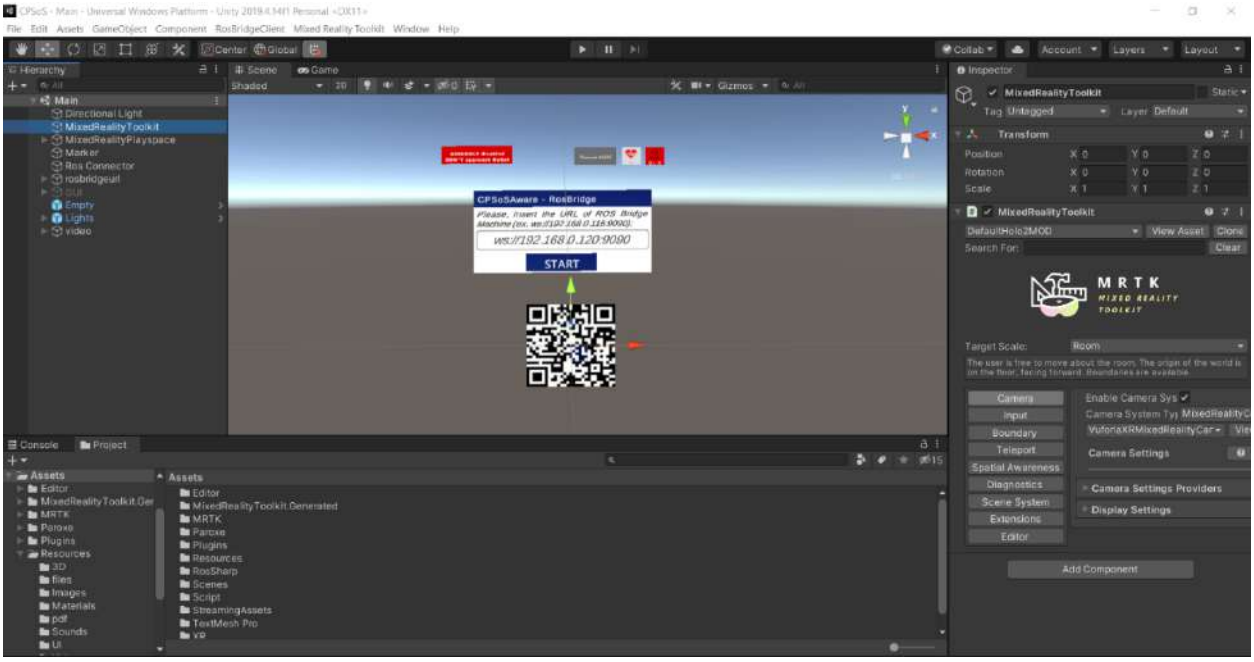


Figure 55: Unity editor

In the upper section, the menu bar contains the access way to all the features of game engine, in particular the configuration of Player to build application for a specific Operative System (OS) or specific device (in our case, the Hololens 2). As we see before, the game engine is based on the concept of the GameObject. It represents the container for components; each component implements a functionality into the virtual scenario. GameObjects can be created from scratch (adding an empty object in the hierarchy using the "Create Empty" function) or by dragging a customizable prefab object ('*<object_name>.prefab'*) into the hierarchy tab. All objects have the *Transform* component active by default, which contains not only the information about the position in XYZ world, the rotation in Quaternion, and the scale of the object within the 3D scene; the Transform is the only component that cannot be removed from the GameObject and the only one in the case of an empty object. The component also contains information about the "parenting" of the object, which cannot be modified through the inspector tab but only through the hierarchy tree (drag&drop) and via script; furthermore, the component is inherited by the 'child' GameObjects (*Child*); the Component represents the transformation (position/rotation/scale) of the *Child* respect to 'parent' element (*Parent*).

We don’t examine over in depth the IDE on this document, because of the aim of project, but we carry on description of our solution, and investigate over on ROS communication, and how, through the IDE/App is possible to communicate with the ROS Network

6.2.3 ROS connector

To allow holographic application to visualize in real-time an XR guide for training in assembly process and results about the status of workstation, we have to design a UWP app for Hololens with a communication solution to allow information exchange with all the actors in the use case network.

Hololens 2, *C#* language and a UWP app based on *.Net*, offers a lot of solutions in term of communications. In the image below, we show a brief recap of possible choices.

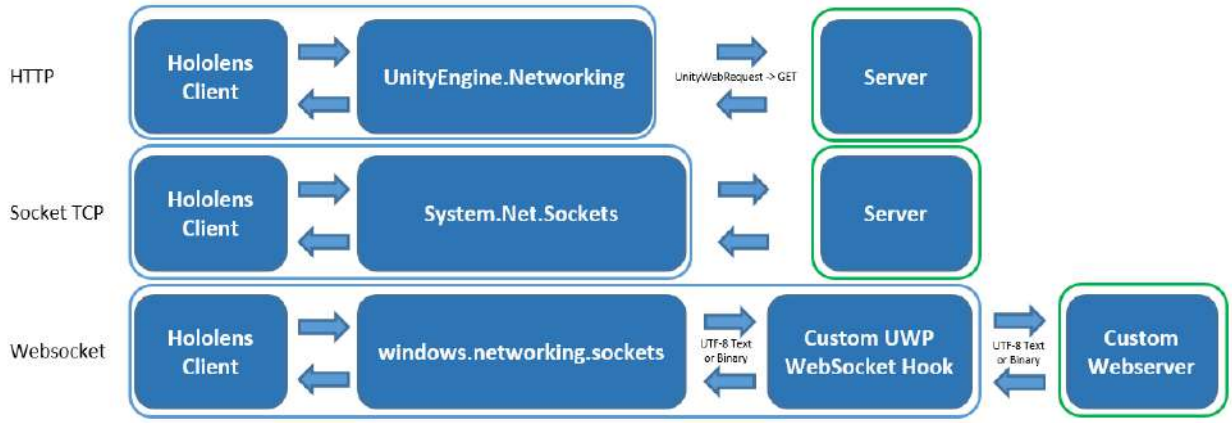


Figure 56: Hololens communication solutions

However, with the aim to allow a communication integrated into a network with heterogeneous device and systems, according to Use Case's project partnership, we choose a solution that allow us to integrate our application in a ROS network.

ROS (Robot Operating System) is an open source framework used to develop software for robotics; it provides a suite of libraries and tools designed for heterogeneous machines by creating an hardware abstraction. The mainly feature of cross-collaboration is at the base of the choice to use this framework in the project to allow an easy exchange in communication between different system and different programming languages.

In order to reach the target of communication between our *C#* application (*UWP* app installed into the hololens) and the rest of the *ROS* Network of the project, we used the *Apache 2.0* licensed software *ROS-SHARP*, to implement, into the virtual scenario in development on Unity platform, a tool for sending and receiving messages from *ROS*.

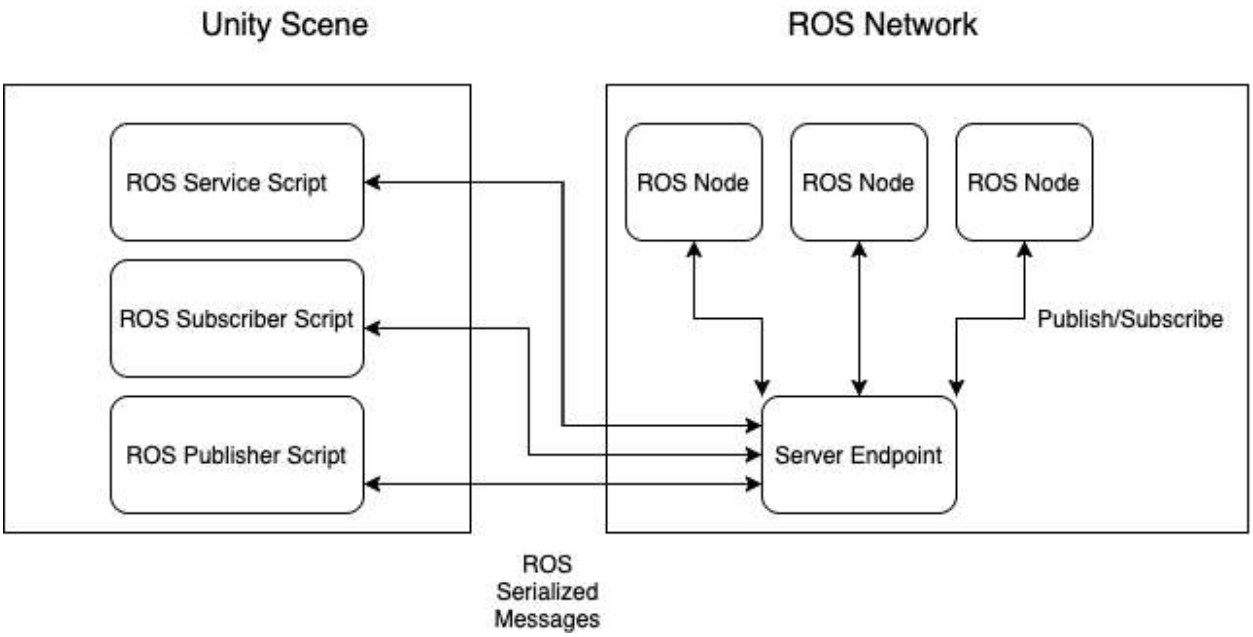


Figure 57: ROS – Unity Communication

ROS Sharp is a set of libraries in *C#* for communicating with *.NET* application; on the base of tool there is a *TCP* endpoint running as a *ROS* node which facilitates message passing between *Unity* and *ROS*. The plugin contains messages and all the scripts necessary to publish, subscribe and call service; user could customize all scripts or messages to reach the target. The package, as open source code, is released as source code or unity asset package.

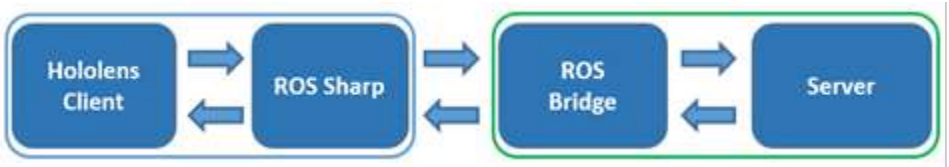


Figure 58: Hololens communication – ROS solution

As showed in the image, our solution consists of a *ROS Sharp* instance integrated into the *UWP* application, customized with scripts for subscribing to the *ROS Topics* implemented on the system. To communicate with rest of the system, our solution requires a server integrated in a ROS node machine able to run an instance of a *websocket server* based on *Rosbridge*.

The *Libraries* section of *ROS Sharp* contains *.Net* solutions for *RosBridgeClient*, *Urdf* and *MessageGeneration*; in particular:

- *RosBridgeClient*, the *.NET API* to *ROS* through *rosbridge_suite*

- *MessageGeneration*, the *.NET* library for generating *C#* source code for *ROS* message, services and action
- *UrdfImporter*, the *URDF* parser for *.NET* application

Rosbridge_suite is a *BSD* licensed library that contains *rosbridge*; it provides a *JSON API* to *ROS* functionality to allow messages exchange. We designed our application to works with *websocket* protocol. It is a hidden layer into the application, but allow to transport a *JSON* message in the format *Operation-Topic-Type*:

```
{ "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

Figure 59: JSON message for subscribing (example)

The message contains all the information required in a ROS communication:

- *op*: type of operation, represents what action is required to do, subscribing to a specific topic or publishing on a specific topic
- *topic*: the topic name
- *type*: the message type

The solution allows us the possibility of customizing the message type on communication also; for the scope of use case, and according to work group, all the communication messages exchange in the *ROS* network follow the standard type. *RosSharp*, via *RosBridgeClient*, contains the folder *Std*, in which we can found all the standard message type in *.cs* format (*Bool.cs, String.cs*, etc etc…). It is mainly important to increase the system flexibility.

The packages *rosbridge_library* and *rosapi* allow all the services required to communicate in a *ROS* network, including settings params and topics; the core of the suite is the *rosbridge_server* library, that provides the *JSON-ROS* conversion and the *WebSocket* connection.

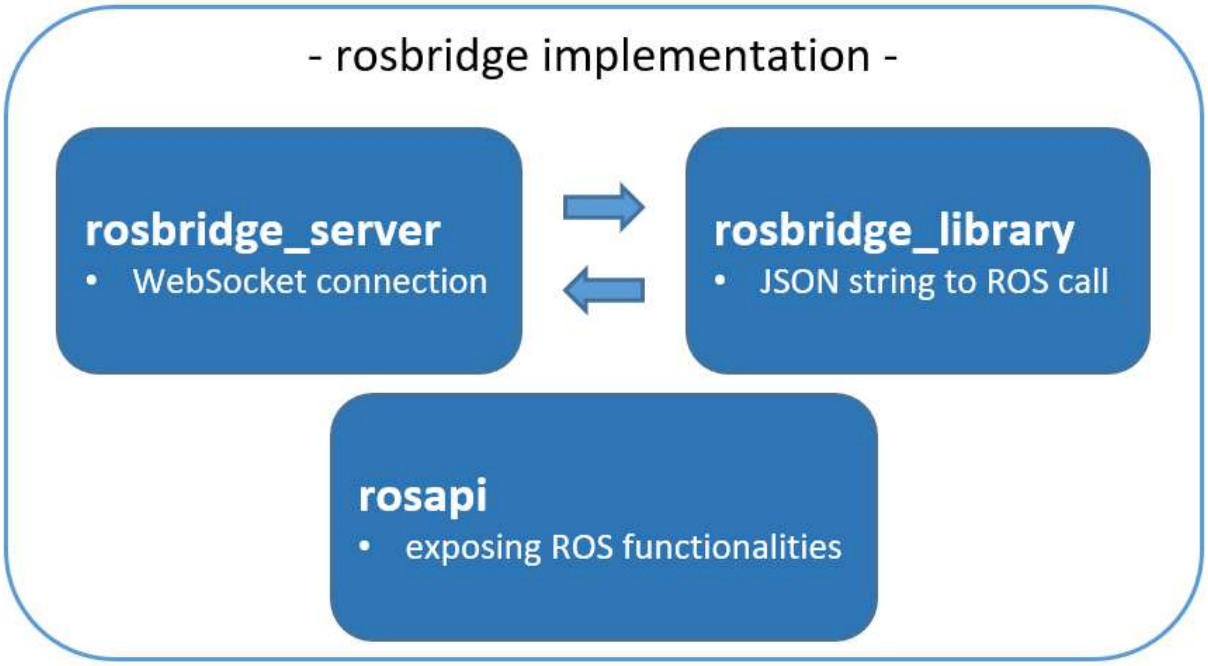


Figure 60: Rosbridge implementation

6.2.4 Application and test

To implement the features described before in our *UWP* application, we started with integration and configuration of the *ROS Sharp* libraries package into the integrated development environment selected. To do this, we used the Unity package version of the tool. Then, system is ready with a simple configuration, introducing in the main scene an *Empty* GameObject, renamed in "*ROS Connector*". This virtual object will contain all the *Components* required for *ROS* communication; in a first time, is required to add the script *RosConnector.cs*, located in "*Assets/RosSharp/Scripts/RosBridgeClient/RosCommuncation/*". This script is responsible of the *WebSocket* connection to the *Rosbridge server*.

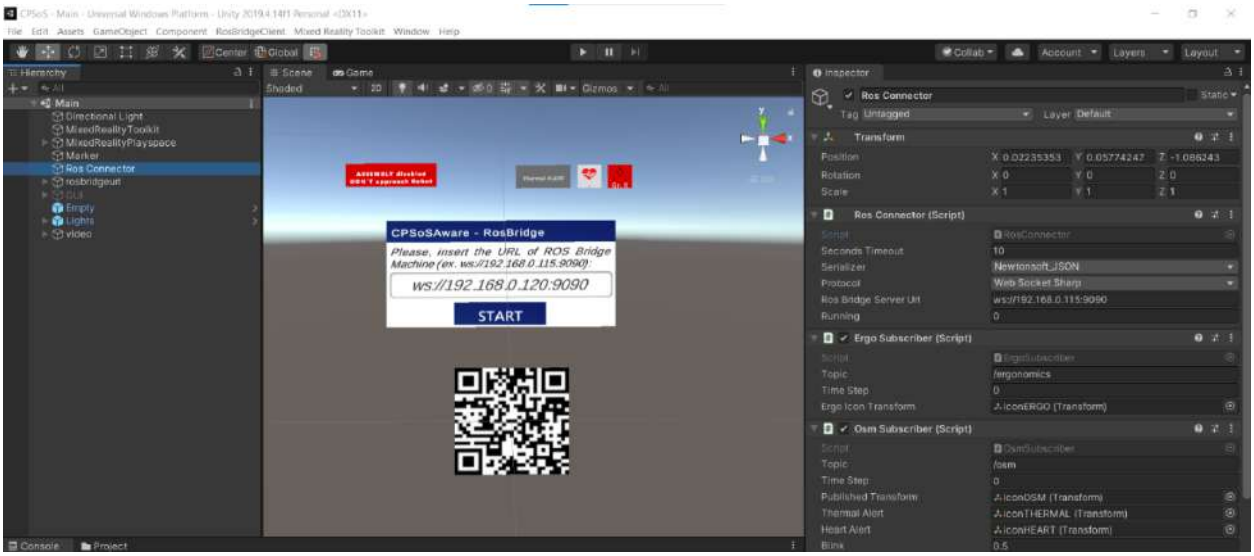


Figure 61: Unity Editor – Ros Connector details

As visible in the inspector tab, *RosConnector* class allow us to set parameters to *rosbridge*; in particular:

- *Timeout*: time in seconds after that connection is closed
- *Serializer*: class or library used to serialize *JSON*; we referred to *Newtonsoft's JSON.NET* framework.
- *Protocol*: a *RosSharp.RosBridgeClient.Protocols* protocol used by *ConnectAndWait* method to connect to *RosBridge* server. The choice is between "*Web Socket Sharp*" when using Unity Editor or "*Web Socket UWP*" when using the Hololens.
- *url*: a string used for *ROS* bridge server *url*. In the example "*ws://192.168.0.115:9090*"

The core of *RosConnector*, as we said, is the possibility to connect to a *ROS* network to publishing or subscribing at/to a *rostopic*. First of all, according to work group, we identify the topics and the types of messages content to communicate over the network.

From	Information	Type	To
System controller	height group indication	Scalar(e.g. 1003)	hololens
Robot controller	Start of movement cycle	Scalar	hololens
Robot controller	Manipulation position confirmation	Scalar	hololens
Ergonomics System - controller	Wrong position detected	Scalar	hololens
OSM system management	Warning Operator State detected	String	hololens

We defined four main topics, "*osm*", "*ergonomics*", "*operator*" and "*robotics*"; in particular:

- *osm*: for Operator State Monitored Status, Heart Rate Value and Thermal Alert Value.
- *ergonomics*: for OK/NotOK ergonomics status of operator movements
- *operator*: for Operator's Height Group ID
- *robotics*: for Robot Operative State

The *osm* topics is used to communicate three different information data from different sources: these data were merged in a single formatted string (*MessageTypes.Std.String*) on a *JSON* packet and wrote on the network (with pre-defined frequency). Below an example of message received on topic */osm*:

data: {operator_state:alerted, heart_rate_value:75.0, thermal_alert:2data: {operator_state:alerted, heart_rate_value:75.0, thermal_alert:2

}

The *ergonomics* topic is used to communicate a *true/false* status of ergonomics index analyzed on operator movements; to guarantee a more flexible tool (with possibility to add other values or codes), it was designed as an *Int16* (*MessageTypes.Std.Int16*) numeric data. For example, on topic */ergonomics* we could receive message,

data:'1001'data:'1001'

The *operator* topic is used to communicate height of the operator and to adapt position of robot (and consequently the height of operator's work plan). Analogously to *ergonomics*, how result of ergonomics

analysis (we'll examine in deep later), an *Int16* numeric value was shared through this topic to communicate operator's height and group ID.

Finally, the *robotics* topic is used on communication between robot and *UWP* holographic application to inform operator on assembly phases. Similarly, to *ergonomics*, also for *robotics* we defined an *Int16* numeric value for messages to represent an *OK/NotOK* status.

Then, to pursue our objective, we created and added to the *RosConnector* GameObject, all the subscriber required for communication in *CPSoSAWARE* system, one for each topic we would subscribe. So, we created and customized four subscribers (three for *MessagesTypes.Std.Int16* and one for *MessageTypes.Std.String*) to allow connection between our holographic app and the rest of *ROS* network.

Details about topics, parameters' values and behavior of Holographic Graphic User Interface will be described later.

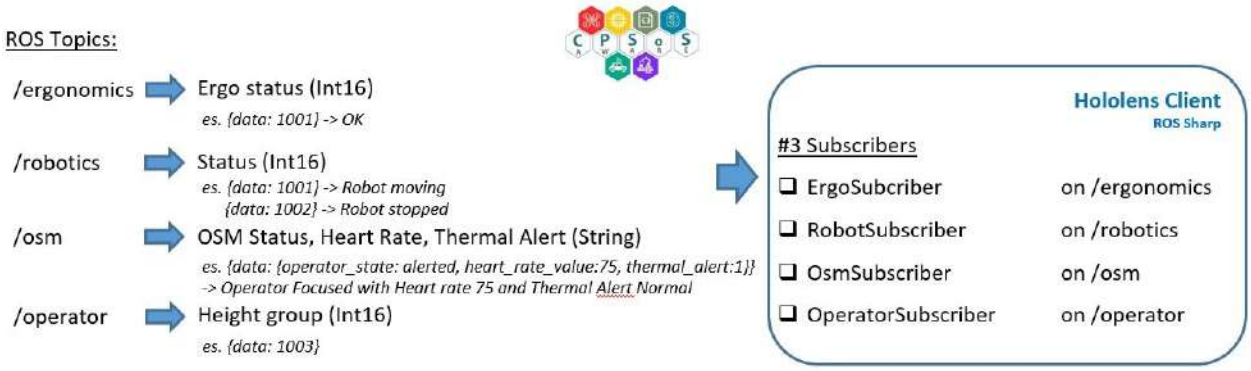


Figure 62: ROS Topic and Subscribers

To verify our described solution, in the first phase we tested on a local network the Holographic *UWP* App in development on a realistic scenario of communication: we used a local network with two machines, a *Windows10* machine running the Unity Editor (IDE) and an *Ubuntu20* machine running *ROS* and the *ROSBridge* server. The image below shows the schematic behavior.

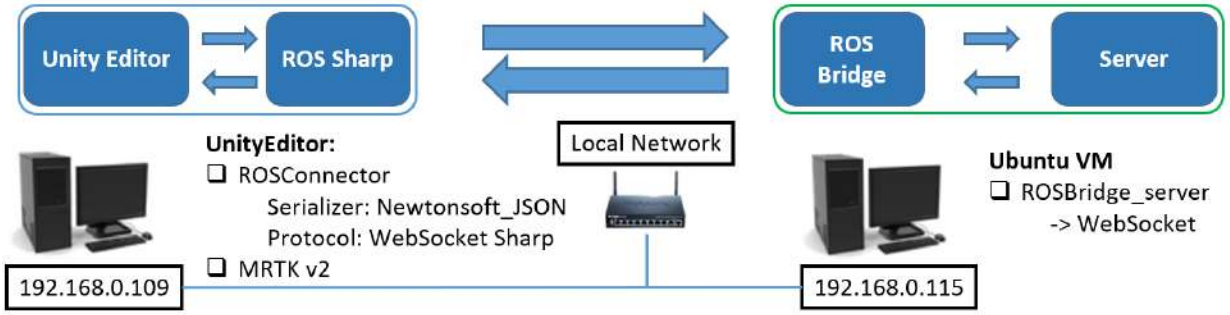


Figure 63: Testing scenario for Unity-ROS communication

Starting the holographic application through the Editor's Scene Tab, the connection between *RosConnector* and *rosbridge websocket server* was established and the app initiates subscriptions to the established topics. Through a second virtual machine terminal, it was possible to verify the receipt of messages published on a specific topic in the IDE Console.

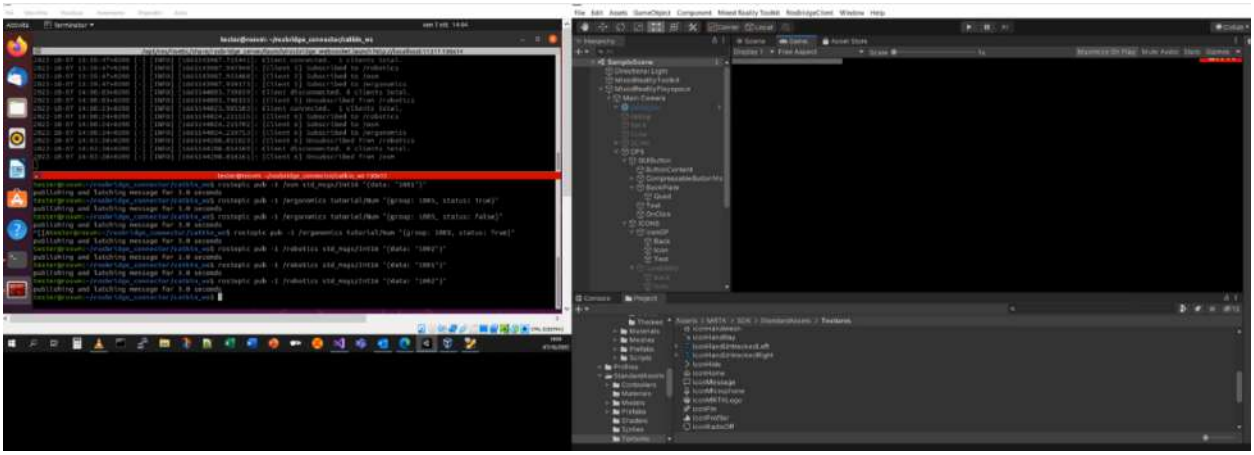


Figure 64: Configuration output: Ubuntu with ROS machine (left) and Unity Editor on Windows10 (right)

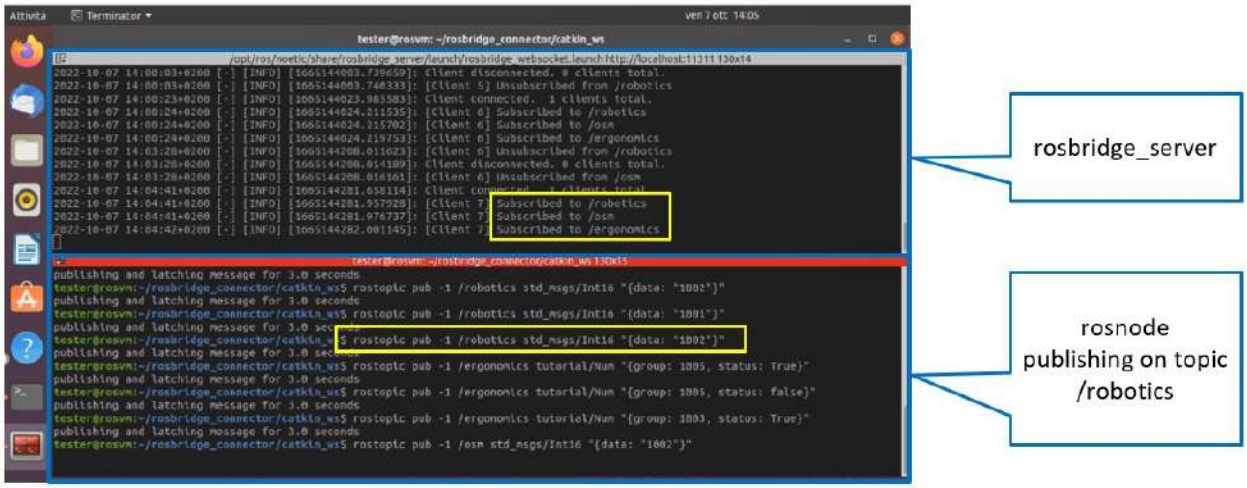


Figure 65: Details of rosbridge server output with subscribed client at topics (up) and example of publication (buttom)

6.3 Anthropometric Recognition

In the proposed architecture, as presented in, 3 stereo cameras are utilised in order to cover the most possible visible area of the operator's working space, so as to increase the possibility to have a good capture of the operator pose, with the most suitable direction (i.e., direct capture in front of the operator), at least from one of the cameras, while the operator is moving in different directions and positions. Each RGB camera of the stereo set is used to monitor the human's actions. A pose estimation algorithm is running to extract the posture 2D landmarks that are used then for the 3D landmark estimation based on a Direct Linear Transform (DTL) triangulation approach. Each stereo camera is used to extract the 3D pose landmarks of the operator, in real-time, for height estimation and to calculate the current ergonomic state based on the RULA score.

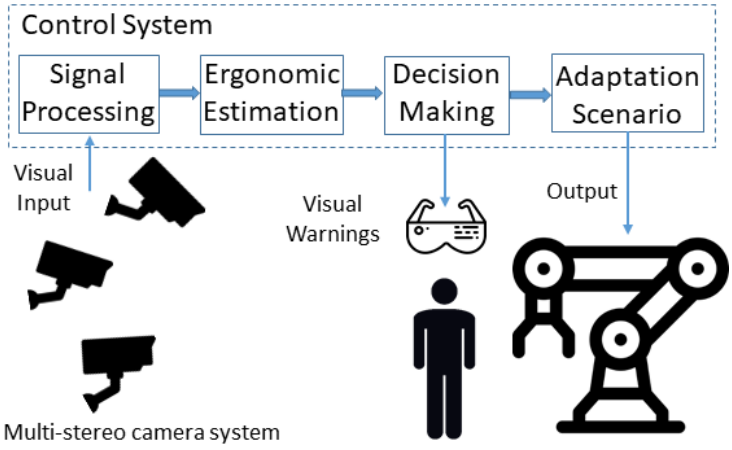


Figure 66: Architecture of the multi-stereo camera system.

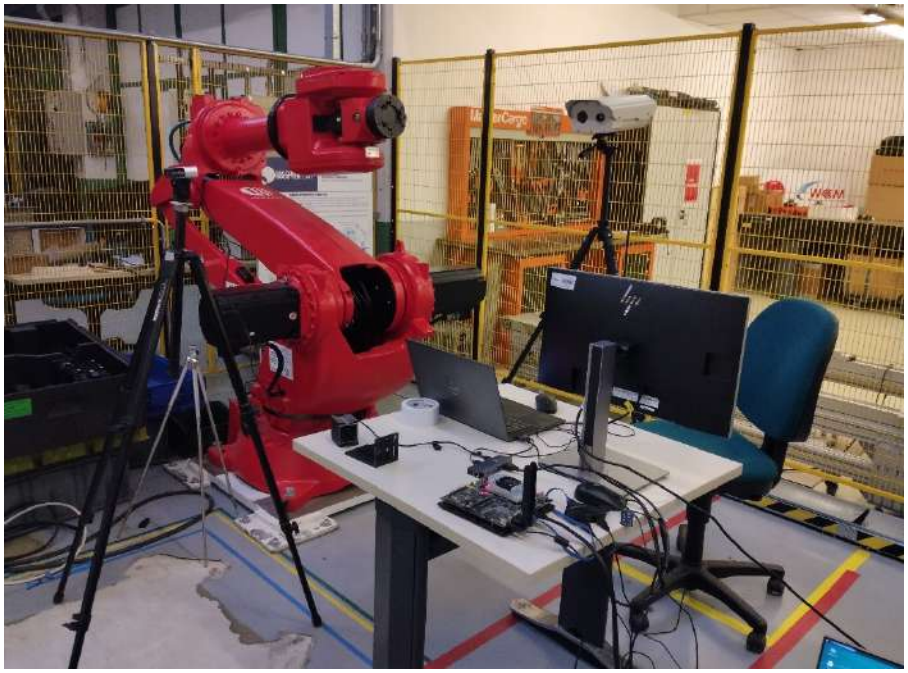


Figure 67: Part of the used equipment.

Figure 67 present an example of the equipment that are used during the implementation of the scenario in the manufacturing environment. Two stereo cameras are directly connected to the computer server which runs all the necessary algorithms (2D landmark estimation, 3D landmark calculation, anthropometrics and ergonomics estimation) and the third one is to a Jetson device. Jetson runs only the 2D landmark estimation algorithm and then sends all the appropriate captured information (i.e., 2D landmarks per

frame) via ROS messages to the server (wireless connection). All of them have (i.e., server and external devices) to be connected to the same local network. More details about the architecture, the algorithms that are used and the steps that are followed will be presented in the D6.5 deliverable.

Table **10** presents the form of the ROS message that is sent based on the operator's height class, while Table **11** presents the ROS message regarding the ergonomics state of the operator.

Table 10: ROS message based on the operator's height.

Operator's height (in mm)	**ROS message**
height < 1544	1001
1544 <= height < 1641	1002
1641 <= height < 1727	1003
1727 <= height < 1829	1004
height > 1829	1005

Table 11: ROS message based on the ergonomics state of the operator.

Ergonomics state	**ROS message**
Safe sequence of poses	1001
Harmful sequence of poses	1002

The server sends the ROS messages (i.e., class of operator's height to the robot and ergonomics status to the Hololens) every 6 seconds (60 frames with a rate of 10 frames per second (fps)) to the ROS topics \operator and \ergonomics, correspondingly.

During the pilot in the CRF premises, there were not any changes to the initial architecture plan regarding the multi-camera system. The integration of the architecture in the real environment was successfully implemented taking also into account the observations that have been identified from the simulated environment. However, there were some considerations about the research findings that makes the real-case scenario more challenging. More specifically, in the simulated environment, we are able to set up the cameras in any place that we want since the simulated cameras in the unity environment do not have physical limitations regarding the wires (they can be placed in any location and at any distance), and there are no connection problems.

- **Locations of the cameras**. It is not feasible for the cameras to be placed in any location that we wanted, and this is mostly for security reasons, since they may block the free access of the operator, they might activate the "safety eye" that affects the movement of the robot, and also they have to be placed in a small area inside the working room (few available locations).

- **The physical connection of the devices to the server**. The multi-stereo cameras system is connected to pcs and jetson devices and it requires wires. Large extensions of the wires may have as a result downgraded the captured resolution quality of the cameras which then affects the quality results of the algorithm.

- **Network issues**. It is necessary for a stable and continuous working connection in a local network for all devices (i.e., jetson, thermal camera, smartphone, KUKA robot) since they are connected to the server wireless.

- **Calibration of the cameras**. The calibration of all cameras as an integrated system into the same world coordinate system could negatively affect the performance of the algorithm by introducing possible errors.

Finally, the integration in the real environment shows the correctness of the initial assumption and ensures that at least 3 stereo cameras are necessary for a successful implementation since at least 1 camera must have a perfect view of the operator's body which is critical, especially in the case of the operator's height estimation.

An example of the integrated system's output is presented in the following Figure **68**.

6.4 Backend for processing outputs of different cameras to increase robustness of the scene analysis

For the reliable height estimation of the operator, we must select the best possible landmarks configuration from the involved cameras. Due to varying camera viewpoints and operator's movement, it is expected that landmarks, detected from different cameras, will not be quite the same, nor their accuracy. To this end, it its required to design a fusion approach which couples together the multicamera system to provide as output the final group of anthropometric landmarks of the operator. For this task, we create an undirected graph, consisting of cameras and landmarks as nodes and cameras along with detected

landmarks as edges. In order to take advantage of this graph topology formulation, we apply the Graph Laplacian Processing technique using as anchor points the average landmark from each camera configuration, for re-estimating landmarks' position in an optimal manner

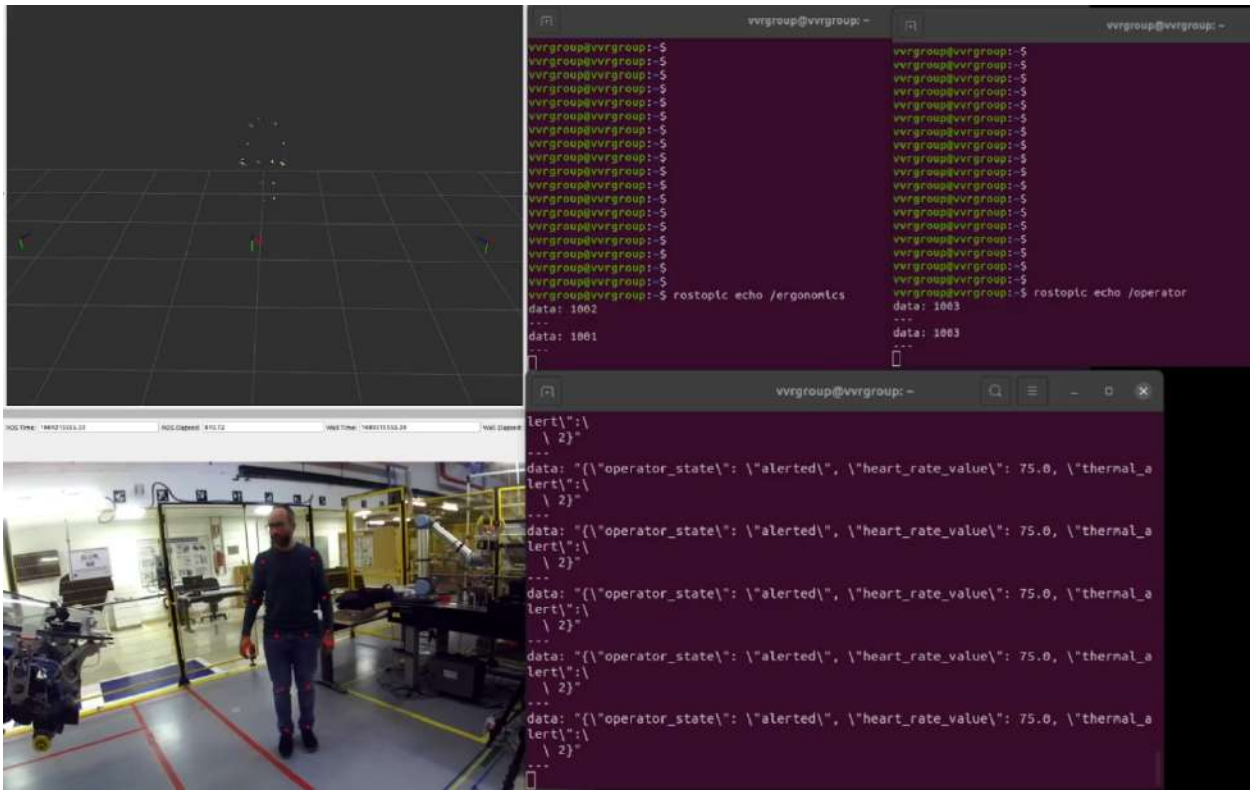


Figure 68: Output of the integrated system.

6.5 Operator State Monitoring

The state monitoring of the operator is conducted through the exploitation of two components, a thermal camera and an Android application. The first is responsible for measuring the body temperature of the operator while the second is responsible for monitoring the general status of the operator based on the yawning activity that he/she presents and his/her heart rate values. More information about each component can be found in the below subsections.

6.5.1 OSM Android Application

The OSM Android Application is responsible for monitoring the general status of the operator while he/she is interacting with the robot. The main functionalities of the application are the yawning

monitoring of the operator, and the retrieval of the heart rate values. For the first, the application exploits ML Kit which is a standalone library developed by Google, which offers the means to conduct machine learning operations on the edge devices. From the above library the OSM application makes use of the Facial Recognition algorithm which locates the face of the user and extracts several facial landmarks. For the yawning estimation we utilized a MAR (Mouth Aspect Ratio) algorithm, using as input the 38 points which form the landmark of the mouth as shown in Figure 69.

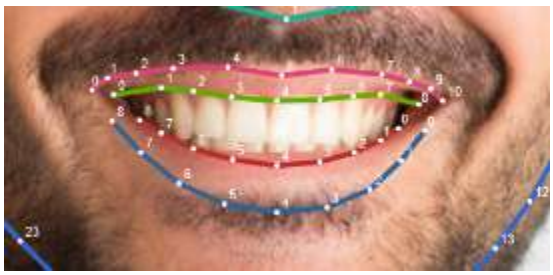


Figure 69: ML kit face detection – Point of Mouth.

From those points we utilized points 3 ,4 and 5 both from the bottom and the upper lip alongside with points 0 and 10 which are the edges of the mouth. Having the coordinates of those points we calculated the MAR using the following equation:

$$\mathbf{MAR} = \frac{\|p_2 - p_8\| + \|p_3 - p_7\| + \|p_4 - p_6\|}{2\,\|p_1 - p_5\|}$$

Where the points refer to Figure 70.

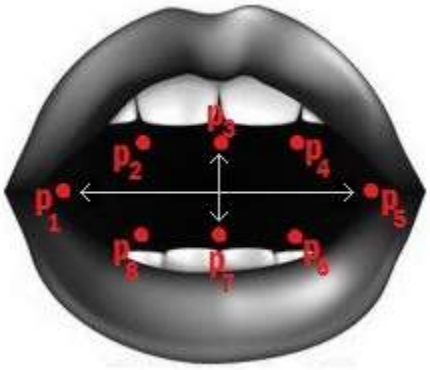


Figure 70: Mouth point utilized for calculating MAR

For the second functionality, the heart rate retrieval, a smartwatch was also introduced, and a dedicated application has been developed. The smartwatch was a Samsung Galaxy Watch 4 running the Google WearOs and the watch application was responsible for retrieving the heart rate of the operator and communicating the value to the handheld (mobile) application via a dedicated channel. The workflow of the whole operation is shown in Figure 71.

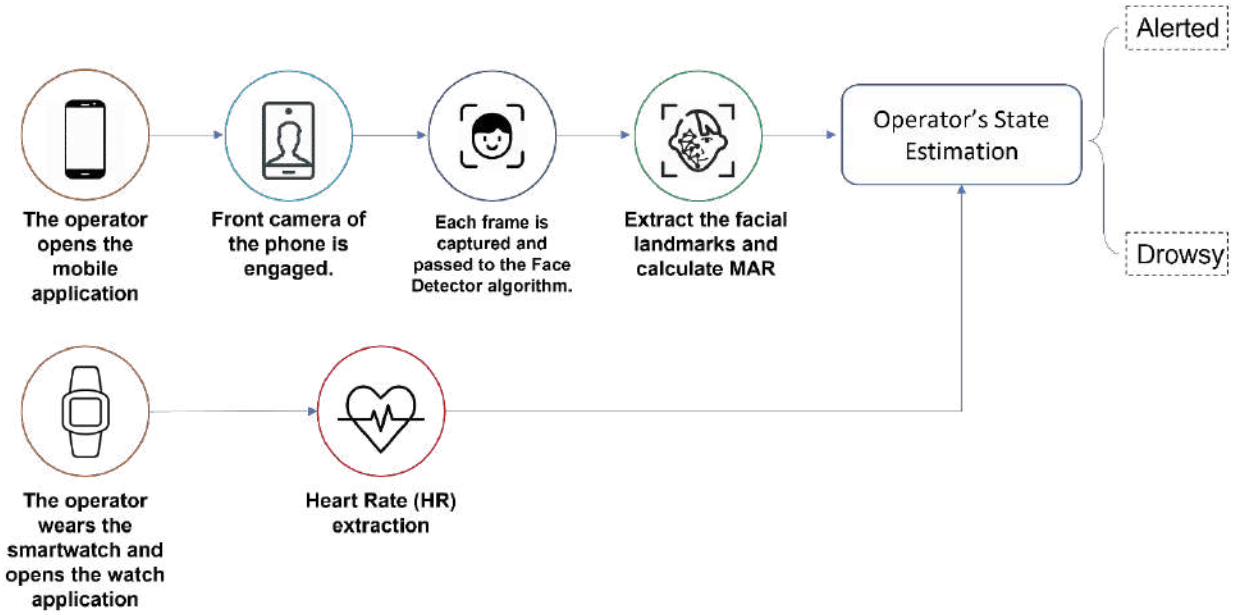


Figure 71: Workflow of the OSM Android Application

As depicted the results of the MAR calculation and the Heart Rate value both contribute to the final estimation of the state of the operator. The analysis results into one of two defined classes namely *alerted* and *drowsy*.

For the communication of the results, as a first step we investigated the possibility of directly sending the necessary ROS messages to the robot. This operation, though, could not be established for the Android smartphone and thus an API was developed and hosted in a central computer provided by ISI. The smartphone, the smartwatch and the computer were connected in a local network and every 5 seconds the Android application was sending an HTTP POST request to the dedicated API with the following data (Figure 72).


```
{
    "operator_status": "alerted",
    "heart_rate_value": 80,
    "timestamp":"2022-11-21T12:46:13.101Z"
}
```


Figure 72: OSM Android Application result sent to dedicated API

Upon receival, those data were sent via ROS messages to the robot and the hololens.

7 Conclusions

This deliverable continued the previous and initial version of the integration activities for the CPSoSaware components as reported in D5.2. The activities of T5.2, reported in this report, advanced in parallel with the preparation and execution of the demonstrators as reported in D6.4. The knowledge and experience gained from the demonstrators (both simulation based and real environment) has been used for fine tuning the components and their integrations aiming to improve their performance and robustness of their operation.

8 References

[1] "D1.4 : Second Version of CPSoSaware System Architecture".

[2] "Deliverable D6.5 Final evaluation and assessment of CPSoSaware platform – CPSoSaware Consortium – December 2022.".

[3] "D6.4 Preliminary Evaluation and Assessment of CPSoSaware Platform".

[4] [Online]. Available: https://www.jenkins.io/.

[5] [Online]. Available: https://www.nongnu.org/cvs/.

[6] [Online]. Available: https://subversion.apache.org/.

[7] [Online]. Available: https://git-scm.com/.

[8] [Online]. Available: https://www.mercurial-scm.org/.

[9] [Online]. Available: https://www.jenkins.io/doc/book/pipeline/.

[10] "Deliverable D4.4: Preliminary Version of CPSoS Simulation Tools and Training Data Generation".

[11] "Khronos® OpenCL Working Group, "The OpenCL™ Extension Specification", V3.0.12, Chapter 46, https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Ext.html#cl_khr_command_buffer".

[12] "D3.2 OPENCL PROTOTYPE TO SUPPORT DISTRIBUTED EXECUTION OF KERNELS AND DATA TRANSFERS IN CPSS".

[13] "T. Leppänen, A. Lotvonen, P. Jääskeläinen, 2022, "Cross-vendor programming abstraction for diverse heterogeneous platforms", Frontiers in Computer Science, vol. 4, https://doi.org/10.3389/fcomp.2022.945652".

[14] “Deliverable D2.3:”.

[15] “Deliverable D3.6:”.

[16] “Z. Ruan, T. He, B. Li, P. Zhou and J. Cong, "ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, pp. 9-16, doi: 10.110”.

[17] “D. Tiwari, S. Gupta and S. S. Vazhkudai, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2”.

[18] “ISO/TS 15066:2016 Robots and robotic devices - Collaborative robots”.

[19] A. Canciani and J. Raquet, “Absolute positioning using the Earth’s magnetic anomaly field,” *Navig. J. Inst. Navig.,* vol. 63, p. 111–126, 2016.

[20] C. Yang, J. Strader, Y. Gu, A. Hypes, A. Canciani and K. Brink, “Cooperative UAV Navigation using Inter-Vehicle Ranging and Magnetic Anomaly Measurements,” in *In Proceedings of the Guidance, Navigation, and Control Conference*, AIAA, Kissimmee, FL, USA, 8–12 January 2018.

[21] F. Teixeira, J. Quintas, P. Maurya and A. Pascoal, “Robust particle filter formulations with application to ter-rain-aided navigation.,” *Int. J. Adapt. Control Signal Process,* vol. 31, p. 608–651, 2017.

[22] “D3.1 Algorithms for monitoring the user and analyzing the scene by fusioning multimodal data”.

[23] “Deliverable D3.3:”.

[24] “Deliverable D6.2”.

[25] “Deliverable D5.2:”.

[26] “D4.8 Final Version of CPSoS Runtime Secutiry Monitoring Approaches”.

[27] “D3.5 Modules for enabling Security and Trust”.

[28] “Deliverable D6.2:”.

[29] “D1.2 Requirements and the Use Cases”.

[30] [Online]. Available: https://www.jenkins.io/.

[31] “Wu, Bichen, et al. "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving." Proceedings of the IEEE conference on computer vision and pattern recognition workshops. 2017”.

[32] “Iandola, Forrest N., et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size." arXiv preprint arXiv:1602.07360 (2016)”.

[33] “ He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.”.

[34] “Cheng, Jian, et al. "Quantized CNN: A unified approach to accelerate and compress convolutional networks." IEEE transactions on neural networks and learning systems 29.10 (2017): 4730-4743”.

[35] “Cheng, Jian, et al. "Quantized CNN: A unified approach to accelerate and compress convolutional networks." IEEE transactions on neural networks and learning systems 29.10 (2017): 4730-4743”.

[36] “Lang, Alex H., et al. "Pointpillars: Fast encoders for object detection from point clouds." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019”.

[37] “Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018”.

[38] “Yan, Yan, Yuxing Mao, and Bo Li. "Second: Sparsely embedded convolutional detection." Sensors 18.10 (2018): 3337”.

[39] “Graham, Ben. "Sparse 3D convolutional neural networks." arXiv preprint arXiv:1505.02890 (2015)”.

[40] “Rothe, Rasmus, Matthieu Guillaumin, and Luc Van Gool. "Non-maximum suppression for object detection by passing messages between windows." Asian conference on computer vision. Springer, Cham, 2014”.

[41] [Online]. Available: https://github.com/riebl/ros.

[42] [Online]. Available: https://www.asam.net/standards/detail/openscenario/.

[43] [Online]. Available: https://github.com/carla-simulator/traffic-generation-editor.

[44] “D4.7 Final Version of Design and Implementation of Smart Dynamic Network Structures for Dependable CPSs”.

[45] “Preliminary Version of Design and Implementation of Smart Dynamic Network Structures for Dependable CPSs”.

[46] “https://wiki.debian.org/iwconfig#:~:text=iwconfig%20is%20similar%20to%20ifconfig,frequency%2C%20SSID).”.

[47] “https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/networking_guide/sec-networkmanager_tools”.

[48] [Online]. Available: https://man7.org/linux/man-pages/man8/sysctl.8.html.

[49] B. Wang, L. Yu, Z. Deng and M. Fu, “A particle filter-based matching algorithm with gravity sample vector for underwater gravity aided navigation,” *IEEE/ASME Trans. Mechatron,* vol. 21, p. 1399–1408, 2016.

[50] D. .. -V. N. P. K. a. K. -D. K. S. Sedighi, “Implementing Voronoi-based Guided Hybrid A in Global Path Planning for Autonomous Vehicles,” in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, Auckland, New Zealand, 2019.