SmartEdge

Semantic Low-code Programming Tools for Edge Intelligence

This project is supported by the European Union's Horizon RIA research and innovation programme under grant agreement No. 101092908

Deliverable D3.1

Design of tools for continuous semantic integration

Editor Darko Anicic (SAG)

Contributors Aparna Saisree Thuluva (SAG), Kirill Dorofeev (SAG), Alessio Carenini (CEF), Marco Grassi (CEF), Mario Scrocca (CEF), Jean-Paul Calbimonte (HES-SO), Davide Calvaresi (HES-SO), Banani Anuraj (HES-SO), Mehrdad Bagheri (CONV), Iisakki Kosonen (Aalto), Danh Le-Phuoc (TUB), Anh Le-Tuan (TUB), Duc-Manh Nguyen (TUB), André Paul (Fhg)

Version 1.0

Date 15 December 2023

Distribution PUBLIC

DISCLAIMER

REVISION HISTORY

Revision	Date	Responsible	Comment
0.1	*15.04.2023*	*SAG*	*ToC*
0.2	*01.05.2023*	*SAG*	*Standardized Semantic Interfaces - first draft (T3.1)*
0.3	*15.05.2023*	*CEFRIEL*	*DataOps - first draft (T3.2)*
0.4	*01.06.2023*	*HES-SO*	*Swarm orchestration - first draft (T3.3)*
0.5	*15.06.2023*	*SAG*	*SmartEdge Schema*
0.6	*01.07.2023*	*SAG*	*Recipe Model*
0.7	*15.07.2023*	*SAG*	*First draft*
0.8	*01.10.2023*	*SAG, CEFRIEL, HES-SO*	*Updating captions, titles, and references*
0.9	*30.11.2023*	*SAG, CEFRIEL, HES-SO, W3C*	*Post internal review and quality check*
1.0	*12.12.2023*	*SAG, CEFRIEL, HES-SO*	*Final submission*

LIST OF AUTHORS

Partner	Name Surname	Contributions
SAG	*Aparna Thuluva*	
SAG	*Darko Anicic*	
SAG	*Kirill Dorofeev*	
SAG	*Haoyu Ren*	
CEF	*Alessio Carenini*	*Section 4*
CEF	*Marco Grassi*	*Section 4*
CEF	*Mario Scrocca*	*Section 4*
CONV	*Mehrdad Bagheri*	*Sections 4.2.2.2, 4.3.5, 4.4.5, UC2 5.5*
HES-SO	*Jean-Paul Calbimonte*	*Sections 3.2.2, 5*
HES-SO	*Davide Calvaresi*	*Sections 3.2.2, 5*
HES-SO	*Banani Anuraj*	*Section 5*
TUB	*Danh Le-Phuoc*	*Section 3.3.2*
TUB	*Anh Le-Tuan*	
TUB	*Duc-Manh Nguyen*	*Section 3.3.2*
Fhg	*André Paul*	*Section 3.2.2*

ABBREVIATIONS

Acronym	*Description*
ADAS	*Advanced Driving Assistance System*
AGV	*Automated Guided Vehicle (a.k.a. AMR)*
AI	*Artificial Intelligence*
AMR	*Autonomous Mobile Robots (a.k.a. AGV)*
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
AuR	*Autonomous Robotics*
BLE	*Bluetooth*
BPMN	*Business Process Modelling Notation*
COCO	*Common Objects in Context*
CORBA	*Common Object Request Broker Architecture*
CPU	*Central Processing Unit*
CQLES	*Continuous Query Evaluation over Linked Streams*
CQLES-QL	*CQLES-Query Language*
CSI	*Continuous Semantic Integration*
D-RDMA	*Declarative RDMA*
DB	*Database*
DDS	*Data Distribution Service*
DMA	*Direct Memory Access*
DPU	*Data Processing Unit*
FDA	*Food and Drug Administration*
FPGA	*Field Programmable Gate Arrays*
FPS	*Frames Per Second*
GDPR	*General Data Protection Regulation*
GPU	*Graphical Processing Unit*
GVA	*Group Virtual Assistant*
HCF	*High-level Communication Framework*
HW	*Hardware*
I2I	*Infrastructure to Infrastructure*
ID	*Identity*
IDE	*Integrated Development Environment*
IDM	*Identity Management*
IoT	*Internet of Things*
IT	*Information Technology*
JSON	*JavaScript Object Notation*
JSON-LD	*JSON-Linked Data*
KB	*Knowledge Base*
JVM	*Java Virtual Machine*
KG	*Knowledge Graph*
KPI	*Key Performance Indicator*
LCF	*Low-level Communication Framework*
LiDAR	*Light Detection and Ranging*
LPWAN	*Low-power WAN*

LTCF	*Long-Term Care Facilities*
ML	*Machine Learning*
MQTT	*Message Queuing Telemetry Transport*
MX	*Mendix*
NATS	*Network Address Translation*
NIC	*Network Interface Card*
NIST	*National Institute of Standards and Technology*
NPU	*Neural Processing Unit*
OMG	*Object Management Group*
OBU	*On-board Unit (V2X wireless communication hardware inside a connected vehicle)*
OPC	*Open Platform Communications*
OPC-UA	*OPC Unified Architecture*
OS	*Operating System*
OT	*Operations Technology*
P4	*Programming Protocol-independent Packet Processors*
PACK-ML	Packaging ML
PLC	*Programmable Logic Controller*
PM	*Particulate Matter*
PVA	*Personal Virtual Assistants*
R2RML	*RDB to RDF Mapping Language*
RDF	*Resource Description Language*
RDF-Star	*RDF-Star*
RFAC	*Robotic Flexible Assembly Cells*
RMDA	*Remote Direct Memory Access*
RML	*RDF Mapping Language*
ROS	*Robot Operating System*
V2X RSU	*V2X Roadside Unit (V2X wireless communication hardware, installed near or inside the traffic sensor node/controller node)*
RTSP	*Real Time Streaming Protocol (video)*
RU	*Rack Unit*
RUST	*Refactoring Using Source Transformation*
RVIZ	*ROS Visualization*
SAREF	*Smart Applications Reference*
SCADA	*Supervisory Control And Data Acquisition*
SDK	*Software Development Toolkit*
SHACL	*Shapes Constraint Language*
SLAM	*Simultaneous Localization and Mapping*
SotA	*State of the Art*
SotP	*State of the Practice*
SPARQL	*SPARQL Protocol and RDF Query Language*
SQL	*Structured Query Language*
SUMO	*Simulation of Urban Mobility (Microscopic traffic simulation software)*
SW	*Software*
TBD	*To Be Defined*

TCP	*Transmission Control Protocol*
TM	*Technology Module*
TRL	*Technical Readiness Level*
TTN	*The Thing Network*
UC	*Use case*
UDP	*User Datagram Protocol*
URDF	*Unified Robot Description Format*
V2I	*Vehicle to Infrastructure*
V2V	*Vehicle to Vehicle*
V2X	*Vehicle to Anything*
WAN	*Wide Area Network*
WiFi	*Wireless Fidelity*
WoT	*Web of Things*
WoT TD	*WoT Thing Description*

Executive Summary

Deliverable 3.1 is the first iteration of the design of tools for Continuous Semantic Integration (CSI) in the SmartEdge project. The document will be updated in month 23 in deliverable D3.2 based on the final revision of the design and the first implementation of the tools for CSI.

Continuous Semantic Integration is a prerequisite for edge intelligence. In the SmartEdge project, the edge intelligence will be realized via low-code applications that are based on sematic recipes. This deliverable defines a semantic model for recipes. Low-code applications process data from connected devices, e.g., robots, industrial devices, vehicles, simulated assets etc. Recipes organize devices in so-called swarms. These devices have different capabilities, communicate via different protocols, exchange information in different formats, and may change over time. For all these reasons, CSI in the SmartEdge project provides standardized semantic interfaces, i.e., a unified access to device data. The interface also provides semantic meta-data about connected devices and relays on established standards. The challenge of continuous semantic integration of different information models and data formats in SmartEdge is tackled with the DataOps toolbox. Once the capabilities and data of connected devices are unified, devices can be orchestrated in swarms. Swarms may accomplish certain goals. SmartEdge provides a low-code approach to orchestrate connected devices with the goal of providing swarm intelligent apps.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Deliverable 3.1 provides the first iteration of the design of tools for Continuous Semantic Integration (CSI) in the SmartEdge project. In this deliverable we report the status of the work in Work Package 3 (WP3), which aims to provide CSI via three tasks: (i) the edge semantics with standardized semantic interfaces for IoT devices; (ii) a DataOps toolbox for continuous semantic integration, and (iii) a declarative and low-code approach for creation and orchestration of swarm apps based on recipes. To this goal, we design concepts for these three tasks. The concepts are based on requirements from SmartEdge use cases and the work from D2.1. The design will be revisited in deliverable D3.2 and requirements from D2.2. D3.2 will also provide the first implementation of tools for Continuous Semantic Integration.

1.1 CONCEPT OF CONTINUOUS SEMANTIC INTEGRATION

The concept of Continuous Semantic Integration is not established. Thus, we explain what CSI is and why it is needed.

In general, the Internet of Things (IoT) together with edge intelligence brings several benefits across various industries and everyday life. These technologies enable the seamless flow of data between devices and systems, leading to improved efficiency and productivity. They can lead to cost savings by optimizing operations. IoT devices generate a vast amount of data. This data can be analysed to gain valuable insights and lead to better decision-making systems. But all these promises come with a hypothesis that the data generated with IoT devices can be easily consumed by intelligent applications. This is not always true, and very often it is a challenge. The reason is that IoT devices have different capabilities, communicate via different protocols, exchange information in different formats, and may change over time. For all these reasons, it is not an easy task to integrate data generated by IoT devices and make them consumable for application developers. Figure 1.1 introduces the concept of Continuous Semantic Integration as a building block between IoT devices and added-value apps. CSI in the SmartEdge project provides Standardized Semantic Interfaces and runs on the edge. Its purpose is to provide a unified access to IoT device data. The interface also provides semantic meta-data about connected devices and relays on established standards. For example, capabilities of devices and their data are described in a machine-interpretable way with standardized vocabularies. With this, the SmartEdge project aims to enable applications that consume IoT device data in a uniform manner.

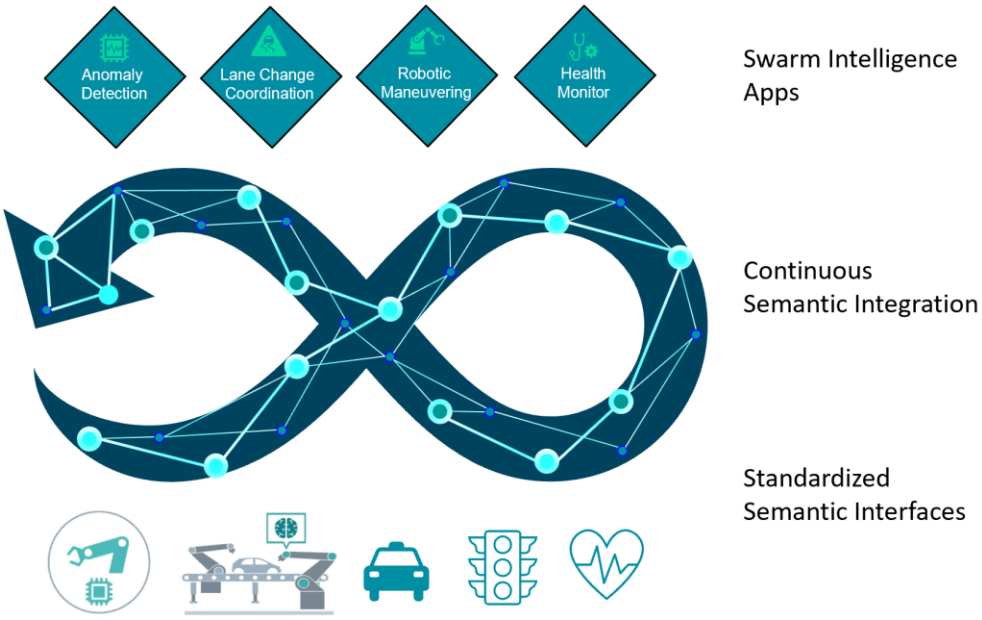

Figure 1.1: Continuous Semantic Integration for SmartEdge

1.2 Structure of the Document

The document has the following sections. Section 2 refers to functional requirements, which are defined in deliverable D2.1 and are relevant for this work; Section 3 provides the concept for Standardized Semantic Interfaces in SmartEdge. This work is primarily the subject of Task 3.1; Section 4 outlines the initial design of the DataOps toolbox in SmartEdge, which is in the scope of Task 3.2; Section 5 reports the current contribution in Task 3.3 on a low-code approach for orchestration of swarm edge applications; and finally, Section 0 closes the document highlighting some of the conclusions found and sketching the next steps in WP3.

2 FUNCTIONAL REQUIREMENTS

This section refers to functional requirements, which are defined in deliverable D2.1 and are in scope of WP3, see Table 2.1, Table , and Table . These requirements will be updated in deliverable D2.2. Thus, we will update them accordingly in the next version of this deliverable, i.e., D3.2.

Table 2.1: Requirements for Hardware and Protocols (Section 3.6.5 in D2.1)

ID	Task	Related Use Case(s)	Priority
HP-001	SmartEdge must integrate dumb devices using protocols such as OPC-UA and DDS.		
	T3.1, T3.2	UC-4	High
HP-005	Capability to read Controller status messages (JSON stream).		
	T3.1, T3.2	UC-2, UC-3	High
HP-006	Capability to read ETSI G5 protocols for relevant parts (V2X).		
	T3.1, T3.2	UC-2	High
HP-007	Capability to read C-ITS protocols for relevant parts (V2X).		
	T3.1, T3.2	UC-2	High
HP-008	Ability to handle public transit open data (tram locations) from outside.		
	T3.1, T3.2	UC-2	Medium
HP-009	Support for Helsinki's open data API for providing data to Helsinki from the swarm sensors.		
	T3.1, T3.2	UC-2	Medium
HP-011	Capability for sending/receiving control messages (e.g., green requests) between operational controllers and vehicles using appropriate format and protocol.		
	T3.1, T3.2	UC-2	High
HP-014	SmartEdge must support integration of heterogeneous devices with digital interfaces and different standard industrial protocols, including mesh, for data collection from "dumb" IoT nodes of the Level 1 Swarm and its reliable forwarding to the Level 2 Swarm node.		
	T3.1, T3.2	UC-5	High
HP-018	The swarm edge components of the SmartEdge toolchain must be deployable onto an underlying software framework, such as Kubernetes or ROS 2. SmartEdge is a series of compatible tools, forming a toolchain, as such it will likely be deployed on top of a suitable software framework.		
	T3.1	UC-3	High

Table 2.2: Requirements for Low Code Programming (Section 3.6.6 in D2.1)

ID	Task	Partner Short Name	Related Use Case(s)	Priority
LC-005	Mendix should provide match making functionality to matchmake the recipes to the available device instances.			
	T3.1, T3.3	SAG	UC-4	High
LC-006	Mendix should enable user to instantiate the recipe with selected devices, configure it and deploy it on the field or Edge.			
	T3.1, T3.3	SAG, DELL	UC-4, UC-3	High
LC-007	Mendix should run on the cloud or the edge.			
	T3.3	SAG	UC-4	High

LC-008	Mendix should deploy the applications instantiated from recipes on the cloud or the edge of a system.			
	T3.1, T3.3	SAG, DELL	UC-4, UC-3	High
LC-009	The low-code platform should support various communication protocols to execute the interactions required for a recipe, e.g., the low-code platform must support OPC-UA as a communication protocol in UC-4, MQTT/rest, Kafka in UC-5, and DDS in UC-3 It must be extendable to implement connectors for further protocols.			
	T3.1, T3.2	SAG, IMC, DELL	UC-4, UC-5, UC-3	High
LC-015	Semantic integration of the data from different sources.			
	T3.1, T3.2	Aalto, SAG, IMC, DELL	UC-2, UC-4, UC-5, UC-3	Medium

Table 2.3: Requirements for Continuous Semantic Integration (Section 3.6.7 in D2.1)

ID	Task	Related Use Case(s)	Priority
CSI-001	SmartEdge must provide a mechanism so that (swarm) devices/vehicles can receive information from the environment.		
	T3.1, T3.2	UC-1, UC-2, UC-3, UC-5	High
CSI-002	SmartEdge must provide standardized semantic interfaces to access any data from the Edge. This applies for new IoT devices as well as for field devices.		
	T3.1, T3.2	UC-1, UC-2, UC-3, UC-4, UC-5	High
CSI-005	SmartEdge must provide mechanisms to formalize external knowledge, (e.g., traffic rules or physiotherapists' rules, therapies, tasks), that are applicable for the current scene.		
	T3.1, WP5	UC-1, UC-2, UC-3, UC-5	High
CSI-008	We should have availability of static information about the environment in standardized format. That is, there should be a way to check physical parameters in the field (e.g., in UC-2 the location of lanes, what is their logical connection, what lanes are controlled by what signal heads, in UC-5 the status and location of a person indoors/outdoors, air quality).		
	T3.1, T3.2, WP5	UC-2, UC-3, UC-5	High
CSI-010	It must be possible to link the knowledge of the environment derived in CSI-005 with the recipe criteria defined be the Low-Code toolchain.		
	T3.1, T3.3	UC-2, UC-3	High
CSI-011	An ontology must be provided that allows the common SmartEdge concepts, such as nodes, smart-nodes and swarms, to be modelled in a knowledge graph, which can be deployed either in the Cloud, or in the swarm smart-nodes.		
	T3.1	UC-2, UC-3, UC-4	High
CSI-012	Domain specific ontologies must be defined (or reused) that can be layered on top of the core SmartEdge ontology that allows domain specific concepts to be modelled in the knowledge graph. For example, specific characteristics of an AMR.		
	T3.1, WP5	UC-2, UC-3, UC-4, UC-5	High
CSI-013	A swarm smart-node must have the ability to correlate sensor data from different sources on the same device in order to enhance the semantic understanding of the environment being observed, e.g. it should be possible to combine sensor streams from LiDAR, cameras, etc. in order to semantically		

	annotate objects and other features in an environment in the smart-nodes internal knowledge graph, the LiDAR giving the physical location of the object or feature and the camera facilitating the classification based on the same frame of reference.		
	T3.2, WP5	UC-2, UC-3	High
CSI-014	The same requirement as CSI-013 by integrating sensor information derived from other nodes in the swarm.		
	T3.2, WP5	UC-2, UC-3	High
CSI-015	A task, defined by a recipe, is instantiated by an application. That is the recipe is a template and a recipe is executed at runtime by an application.		
	T3.3	UC-2, UC-3, UC-4	High
CSI-016	A recipe has a clear objective or outcome. When the application achieves the outcome, the application terminates.		
	T3.3	UC-3	High
CSI-017	A recipe will have zero, one, or many start criteria. These are criteria that must be met before the application can execute.		
	T3.3	UC-3, UC-4	High
CSI-018	The steps of a recipe are defined by goals and primitives. A goal is a like a sub-recipe, in that it has an objective and potentially start criteria. A primitive is some base behaviour that a swarm node innately knows how to perform. Goals are broken down into sub-goals and primitives until the sub-goals are completely decomposed into primitives; at which point the steps necessary to execute a recipe are completely defined by primitives.		
	T3.3	UC-3	High
CSI-019	Ideally an abend strategy should be defined for each recipe, so that if an application should fail during the execution of a recipe, the abend strategy should be put into action to mitigate, or ideally correct the failure.		
	T3.3	UC-3	Medium
CSI-020	By virtue of CSI-018 a recipe must know the primitives necessary to execute the application on a swarm. A mechanism must exist to match up the primitives to the characteristics of possible nodes in the swarm, and in this way define the types of nodes that will be required by a swarm to execute an application.		
	T3.3	UC-2, UC-3	High

This section also refers to an objective, which is in scope of WP3.

Obj.2: Middleware and tools for continuous semantic integration allowing the SmartEdge solution to interact with devices according to a (i) standardized semantic interface, via a (ii) continuous conversion process based on declarative mappings and scalable from edge to cloud, and (iii) providing a declarative approach for the creation and orchestration of apps based on swarm intelligence.

The three parts of this objective are addressed in Task 3.1 (see Section 3), Task 3.2 (see Section 4), and Task 3.3 (see Section 5), respectively.

KPIs relevant for WP 3 are shown in Table 2.4. The goal of this deliverable is to provide design of tools for Continuous Semantic Integration. Thus, the progress towards KPIs will follow in the first implementation of this work, i.e., in D3.2.

Table 2.4: WP3 Key Performance Indicator

KPI number	Description
K2.1	Semantic integration should be provided for at least 4 brownfield protocols and more than 3 green field devices.
K2.2	Message conversion performances increased by at least 80% wrt. to the baseline.
K2.3	Semantic integration scalability (in terms of maximum concurrent requests and data velocity) increased by at least 50% wrt. to the baseline.
K2.4	Reduced complexity and configuration time (at least 70%) of swarm intelligence Apps through the automatic instantiation and orchestration of template-based specifications.

3 STANDARDIZED SEMANTIC INTERFACES FOR SMARTEDGE

This chapter presents the design of standard interfaces for the SmartEdge middleware. It reports on our activities from Task 3.1. The contribution includes:

- SmartEdge Schema
- Recipe Model (concept and implementation)
- Domain Specific Ontologies for each use case
- Standardized semantic interfaces.

Figure 3.1 shows the overview of the semantic models that will be designed and developed in WP3. This figure also shows how the semantic models are related with each other. At the bottom level are the device semantic models which represent the semantic models of devices used in all the SmartEdge use cases. On the right-hand side, we see the domain models, which refer to the existing domain models that can be used for semantic enrichment of the device semantic models to describe the capabilities of devices. On the other hand, the domain models are also used in Recipes to describe the capabilities required for a Recipe. A Recipe formally describes an application template. It specifies the capabilities required for an application and data flow between the capabilities to realize the application. A Recipe can be instantiated and deployed on the devices which can fulfil the recipe requirements. An instantiated Recipe can be seen as a swarm in SmartEdge. SmartEdge schema is used to describe the runtime behaviour of a swarm, also to monitor a running swarm. Each of these semantic models is explained in detail in the next sections.

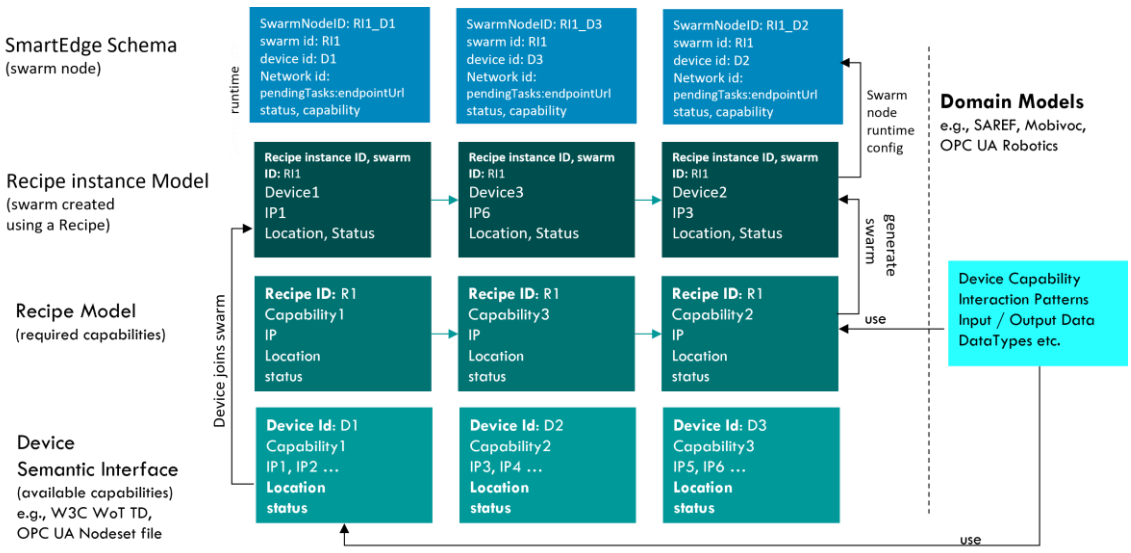

Figure 3.1: Overview of semantic models in SmartEdge

3.1 SMARTEDGE SCHEMA

SmartEdge schema aims to formally define the important concepts of the SmartEdge architecture which are used in swarm formation and execution. It addresses the functional requirements CSI-011 mentioned in Table by providing an ontology which defines the common concepts of SmartEdge and enables them to be represented and stored in a knowledge graph.

Purpose of SmartEdge schema is to enable following swarm functions:

- It can be used during design time for configuration of a swarm;
- It can be used in run time for identifying the nodes with matching skills which can join a swarm;
- It can be used to monitor the execution of a swarm (e.g., entry of a node into swarm, exit of a node and replacing a node in swarm);

It defines the concepts that are common to all swarms regardless of the use case applications, such as swarm coordinator, its interactions with a swarm orchestrator, industrial knowledge graph to discover nodes with required capabilities and nodes in the swarm. The role of a swarm coordinator and orchestrator and the relation between them is explained in Figure 3.2. The section below explains these concepts.

Swarm: A swarm can be seen as an application that is instantiated from a recipe. It executes the tasks prescribed in a recipe and achieves the objective of the recipe. The swarm or tasks of a recipe can be executed centrally by the swarm orchestrator on Mendix runtime. In addition, the swarm orchestrator can also distribute the tasks to swarm nodes, where the tasks can be executed in a decentralized fashion using TUB runtime that is being developed in WP5. In order to formally define the common terms related to swarm and their relationship with each other, in this WP we develop SmartEdge schema which addresses Requirement CSI-015 from Table .

Swarm Coordinator: The role of a swarm coordinator is to do resource coordination for the swarm by allocating required nodes to the swarm and manage the swarm to ensure its successful execution. It is the key component of the swarm as it interacts with different components inside and outside of the swarm for its successful execution. Figure 3.2 represents the role of a swarm coordinator in swarm execution. The tasks of a swarm coordinator are the following:

- Discovering a node that has capabilities to run recipe tasks.
- Connecting to a swarm node.
- Assigning tasks to a node.
- Replacing a node in case a swarm node wants to leave the swarm.
- Monitoring swarm tasks etc.

The swarm orchestrator requests the co-ordinator to provide the swarm nodes required for swarm execution. For this purpose, the co-ordinator first connects to an industrial knowledge graph where the device semantic descriptions are stored and runs the matchmaker (which matches a recipe's required capabilities with available device capabilities) to discover the suitable devices with matching skills which could take part in swarm execution. Secondly, the swarm co-ordinator checks the availability of discovered devices, connects to them, requests them to join the swarm and onboards the node to the swarm. Once a device joins a swarm, it becomes a swarm node, and the coordinator provides the swarm node to the orchestrator. In some cases, the swarm coordinator and orchestrator can reside on the same node.

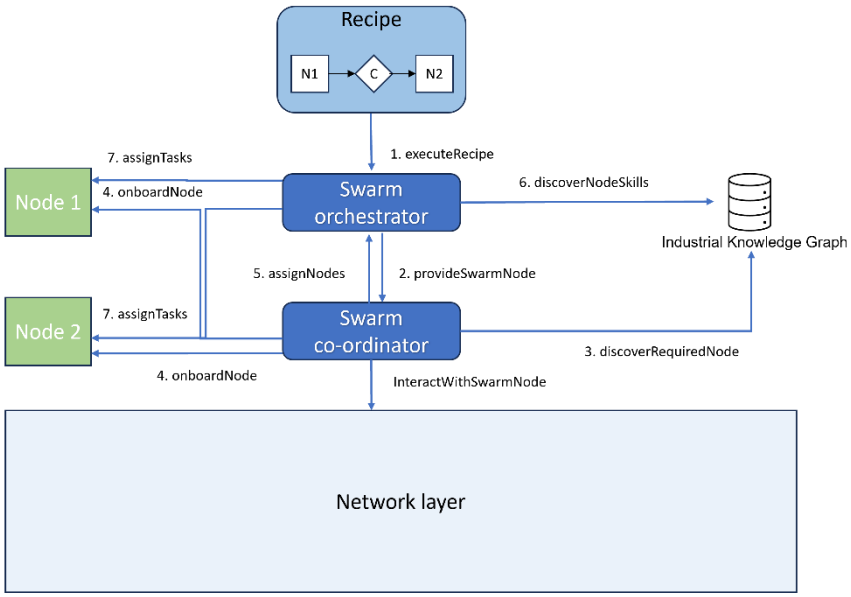

Figure 3.2: Role of a swarm coordinator and orchestrator in swarm execution

Industrial Knowledge Graph: Industrial Knowledge Graph is one of the key components of the SmartEdge architecture. It is an RDF repository which is used to store several semantic artefacts developed in SmartEdge project such as:

- SmartEdge schema for the swarm in execution
- Recipe models developed for use cases
- Device semantic models
- Domain ontologies
- Other semantic artefacts required for the use cases.

The purpose of the knowledge graph is manyfold. It is mainly used for the following purposes:

- For semantic discovery of the artefacts e.g., using SPARQL interface,
- For recipe matchmaking to discover the assets with matching skills to the recipe etc.

One of the good candidates to implement knowledge graphs in the SmartEdge project is the Web of Things Thing Description Directory (TDD).[1] It can be used to store all the semantic artefacts and it provides two interfaces for querying such as: TD interface and SPARQL endpoint. TD interface is specially used to discover Thing Descriptions for the repository, whereas SPARQL endpoint can be used for general SPARQL querying.

Swarm Orchestrator: A swarm orchestrator executes the tasks defined in a recipe in the swarm using swarm nodes. It connects to the swarm nodes with required skills, which are discovered and provided to it by the swarm coordinator. In a centralized approach the orchestrator executes the tasks by interacting with the swarm nodes. In a distributed approach, it assigns the tasks to each swarm node as prescribed by the recipe and executes the tasks by transferring the messages or input / output data from the swarm nodes and executing the constraints defined in the recipe.

In case of dynamic swarms, during the execution of a swarm, if a swarm node may leave the swarm then the orchestrator requests swarm coordinator to provide another available node

[1] https://github.com/thingweb/thingweb-directory

with required skills in order to execute the swarm. The role of a swarm orchestrator is depicted in Figure 3.2.

In the SmartEdge project, the swarm orchestrator can run on Mendix in a centralized approach e.g., UC4, in this case Mendix can be used to create and run the recipes. In a distributed approach, the orchestrator can run on TUB runtime that is being developed in WP5. Swarm orchestrator addresses the required CSI-015 from Table , as it takes care of the execution of tasks in a swarm that are prescribed by a recipe.

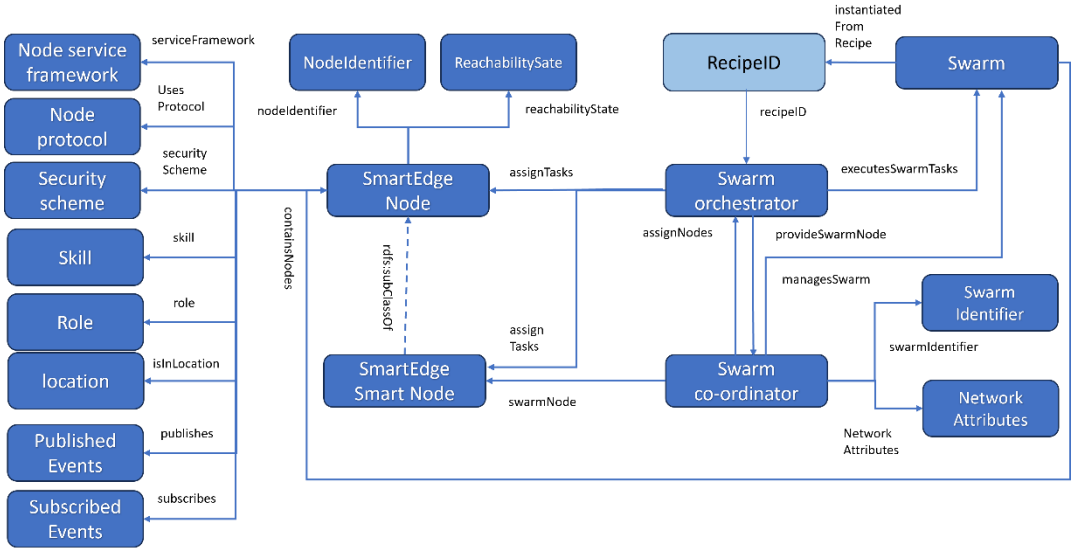

Figure 3.3: Overview of SmartEdge Schema

Figure 3.3 represents the concepts in the SmartEdge schema and the relationships between them. The main concepts in the schema are the SmartEdge node, SmartEdge smart node, swarm co-ordinator and the swarm orchestrator. SmartEdge smart node is a subclass of SmartEdge node where the smart node has the capability to dynamically join or leave the swarm. Each of these nodes has certain attributes and relationships with other nodes which is depicted in Figure 3.3.

Each SmartEdge node has the attributes such as: node id, node capabilities, network attributes, location, events it publishes and subscribes, security scheme to connect to the node, its reachability state etc. which are characteristics of a node. A swarm coordinator has attributes such as swarm-id, network attributes etc. as it manages the swarm and connects to the nodes in the network. Swarm orchestrator has a relationship to the recipe which it runs through the swarm.

3.2 Recipe Model

3.2.1 Definitions

Capability: Capabilities are production-relevant abstractions of functions applied in the context of a process step. Capabilities are implemented by means of skills. E.g., “drilling a hole with a depth of max. 20 cm, diameter of max. 10 mm and with tolerance +/- 0.1 mm into certain types of metals”.

Recipe: A Recipe is a template that specifies the requirements of an application that can be created by composing one or more things or IoT offerings [Thuluva17], [Thuluva20]. Recipe

specifies the capabilities of things or offerings required to execute the application. Additionally, it also specifies the data flow between things or business logic on how the things should interact with each other to achieve the goal specified by the application. Therefore, a recipe template mainly consists of two parts: required capabilities and interaction between capabilities as shown in Figure 3.4.

A capability describes the functional requirements of an application such as the capability of a thing required for an application. For example: drilling capability, capability to lift a product, capability to move a product from a to b, capability to detect an anomaly etc. Optionally capability can also describe non-Functional Properties (NFPs) such as price to access an ingredient, its location, and others. A thing can have one or more capabilities. It is possible to link the knowledge of the environment derived in CSI-005 with the recipe criteria in the form of NFPs. That is, the constraints about the environment can be modelled as NFPs of a recipe capability that must be fulfilled by a matching device or offering in order to instantiate a recipe. They are defined in the device semantic model of a thing using standardized domain semantics. In SmartEdge the capabilities of a thing can be specified as interaction patterns in case of things using Web of things standard, or as FX capabilities in case of things using OPC UA standard as shown in Figure 3.4.

An interaction defines how two capabilities should interact with each other to fulfil a task or achieve a sub-goal of a recipe. It specifies the source and destination capabilities of an interaction. It also defines the operations (e.g., Retrieve, Create, publish, subscribe etc.) that should be executed on each capability to get the required information/output from a capability and execute an application. Furthermore, an interaction also specifies the constraints or conditions for interaction. Lastly, the business logic that should be executed for the application based on the information retrieved from the capabilities can be defined as part of an interaction.

Therefore, the objective of an application can be defined using a recipe by describing the tasks and goals of the application. A task can be defined in a recipe using its capabilities and interactions. One or more tasks can be used to define a goal.

A recipe can be created using low-code application development tools such as Medix, Node-RED etc. In these tools the capabilities are represented as graphical nodes. A user can drag and drop the nodes and they can implement the interactions between the nodes (business logic) as scripts in any programming language. The capability nodes and interactions can be interlinked with each other in a desired way to create a recipe. The graphically created recipe can then be saved for later use. Furthermore, a recipe semantic model can be generated from the graphically created recipe which can be stored in an industrial knowledge graph and used during the instantiation of a recipe.

A recipe semantic model is a formal description of a recipe in RDF format. It mainly contains the semantic description of the required capabilities, and the dataflow between the capabilities that is represented through the interactions. However, the business logic present in the interactions is not part of the recipe semantic model. The main purpose of the model is two-fold. Firstly, it is used to discover the recipes required for an application. Secondly, it is used during matchmaking to discover the things or IoT offerings which have the capability to implement the recipe.

In SmartEdge project there are few requirements for a recipe where it should specify the following requirements:

- objective of the task
- roles (capabilities & constraints) needed to perform the task

- any required starting conditions
- goals and subgoals necessary to complete the task
- conditional transition between goals or sub-goals
- topics related to the transitions
- event messages that will be published when the transitions occur
- any abend conditions and actions.

These are the requirements for recipes in SmartEdge (these requirements are taken from D 2.1: swarm Recipes). As explained in the previous paragraphs a recipe can specify the objective of an application or a swarm, its goals, tasks and sub-tasks. The starting or triggering conditions for an application can also be specified using a recipe. Topics or event messages can be published by capabilities, and they can be subscribed by other capabilities in a recipe. Moreover, one or more recipes can be composed together to achieve a broader goal.

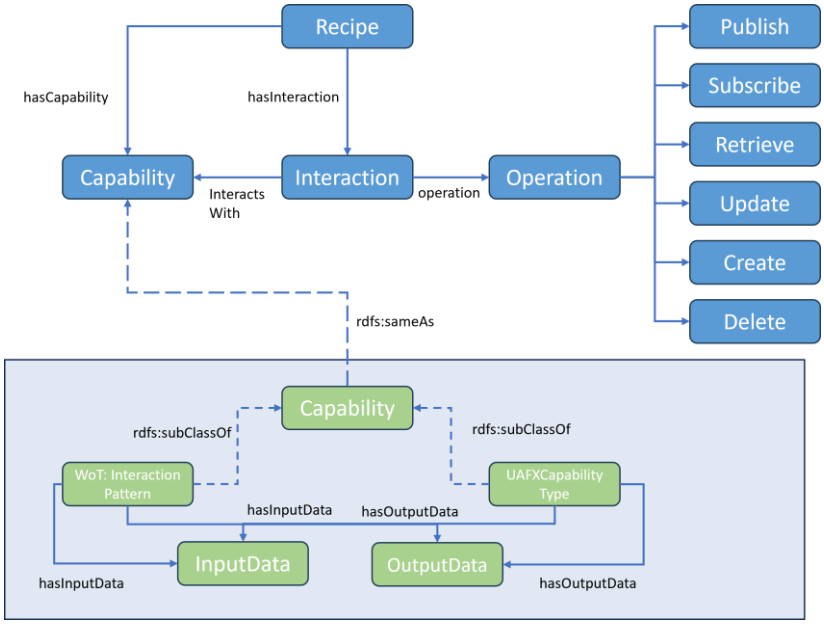

Figure 3.4: Recipe Model

Here we present a sample recipe for use case 3 to show how requirements for an application, its goals, tasks, and constraints can be specified in a recipe. The recipe specifies the requirement for the application to move a manufactured product which is ready for pick up to an appropriate mobile rack. For details about the application please refer to D2.2. Figure shows the recipe with its required capabilities and interactions. In the next paragraphs we explain the recipe in detail.

The objective of the recipe is to place the manufactured product in an appropriate mobile rack. To fulfil this objective, the recipe should achieve the following goal: move the product from the end of the conveyor belt to the chosen mobile rack. Several tasks should be done to achieve this goal such as the following:

- Task1: Identify the mobile rack where the product should be placed as per the product description.
- Task2: Pickup the product when the product is ready for pick up.
- Task3: Move the product from pickup station to mobile rack position.
- Task4: overcome obstacles (if any) while moving the product.
- Task5: place the product in the mobile rack.

A task is specified in a recipe with one or more capabilities and interactions between the capabilities. In our sample recipe, consider Task 3, Figure shows how this task is specified in the recipe. It is specified using the capabilities *"FindMobileRack"*, *"MoveProductToRack"* and the interactions between them. In this way all the tasks mentioned above are specified in the sample recipe using its capabilities and interactions.

The starting or triggering condition means the condition that should be met to start the recipe execution. It is specified in the sample recipe using the capability "*ProductReadyForPickUp"* and the interaction below it. It means that when the "*ProductReadyForPickUp"* event occurs then the product is at the end of the conveyor belt, and it is ready to be moved to the mobile rack. Therefore, the recipe execution should start to move the product.

Conditional transitions are represented in the interactions. Every capability in a recipe should support certain operations such as create, retrieve, update, publish, subscribe etc. using which information can be sent or retrieved from the corresponding thing or IoT offering. Using these operations, we can define the publishing of event messages from a thing or subscribing to event messages from a thing in a recipe.

In this way, a simple recipe model can specify applications and its requirements. In the first phase of the project, we focus on the centralized approach where a recipe will be executed centrally on the Mendix runtime for all the use cases. For this approach, the current recipe model is sufficient. Recipe model can be extended in the later phases of the project to suit the requirements of a decentralized or distributed approach which will be implemented on TUB runtime in WP5.

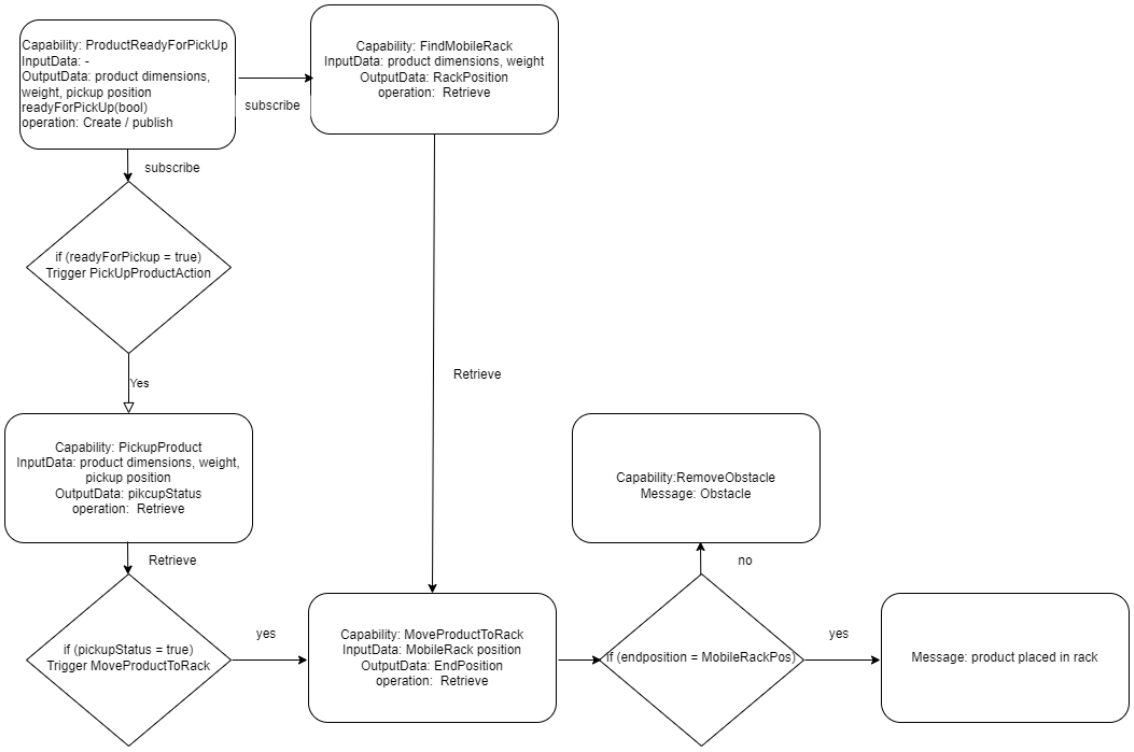

Figure 3.5: Sample Recipe for UC 3

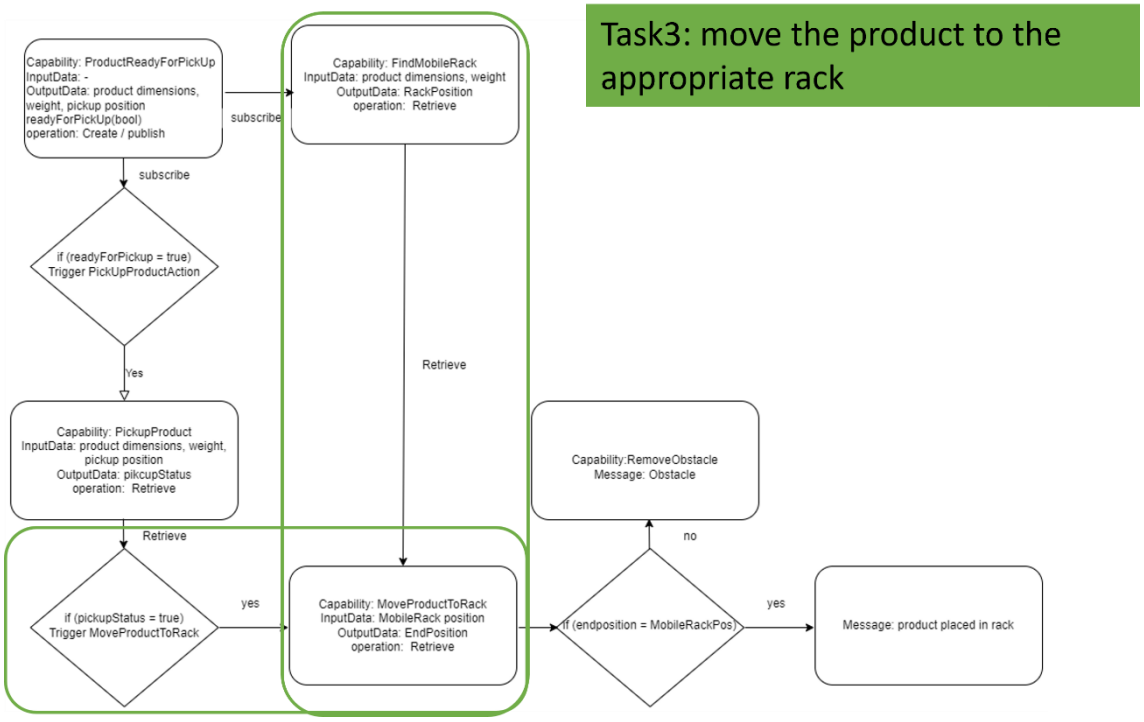

Figure 3.6: Recipe task definition: Move product to mobile rack

In SmartEdge project there are 5 use cases. To understand and address the requirements of all the use cases for recipes, we performed requirement analysis for each use case. In the next section we present the results of the analysis.

3.2.2 Requirement Analysis

There is a lot of diversity in the use cases of SmartEdge project, each one of them is focusing on different domains and heterogenous applications. Use case 1&2 focus on smart traffic management domains, whereas as use cases 3&4 focus on smart manufacturing domains. On the other hand, use case 5 focuses on the health care domain.

Therefore, it is essential to understand the requirements of each of these use cases to understand the applications they would like to demonstrate, identify the capabilities required to implement the applications. Moreover, to understand the physical and virtual devices and assets (or IoT offerings) that will implement the applications. With this purpose we conducted requirements analysis for each use case. For this task we posed some questions to each use case to identify the applications in the use case. Additionally, we also provided them some templates to understand their requirements regarding capabilities and devices required for their applications. Use case owners proactively participated in the requirements analysis and provided the required information. In addition to this, we conducted interviews with each use case team to determine the requirements further. As a result of this entire process, we could identify the semantics required for device semantic interfaces of the devices, capabilities offered by the devices and capabilities required for the applications. We further identified the recipes for each use case. These results will be used throughout WP3 for the design and development of recipe model, device semantic models, and capabilities. Below we present the questions, templates, and results of the analysis for each use case.

Questions:

1. Which applications do you envision in your use case?

2. What are the capabilities required to realize the application?
3. What are the devices used in the use case?
4. What are the capabilities implemented by the device?

These questions are addressed using a table as shown below.

Detailed description about each use case can be found in D2.2. In this deliverable we would like to focus on the requirements of use cases for semantic models such as recipes that the use case owners like to implement to showcase their use case. The capabilities that should be semantically modelled to use in recipes. The assets (devices, things and offerings) that should be modelled using standardized models such as Web of Things Thing Description, OPC UA etc. and the domain semantic models that should be used for the use case are the focus of this deliverable.

3.2.2.1 Requirement Analysis for Use Case 1

Table 3.1 presents sample application(s) from use case 1 and the capabilities required to run the application. An application can be formalized as a Recipe which is the composition of one or more capabilities. Recipe specifies the task, or an application and the capabilities required to implement the task on assets. A capability is implemented on assets or things. Table 3.2 lists the assets from use case 1 which implement the capabilities required specified in for the application.

Table 3.1: Use Case 1: Applications and Their Required Capabilities

Application	**Required Capabilities**
SmartEdge integration with a virtual environment	1. Detecting brightness level 2. Enabling / Disabling a light source
Semantic assessment of ADAS systems using sensor fusion	3. Semantically describe the current state of a system of combined sensors. In UC1 this would be a car and its sensors and different cameras.
Changing scenes to generated alternate test cases	Same as previous application scenario

Table 3.2: Use Case 1: Mapping Capabilities to Things (SmartEdge Nodes)

Capability (from above table)	**Things (nodes) that have the *skills* to implement the capability**
1. Brightness Sensor	Measure the environment's brightness, for example for automatic light control in tunnels or during nights. The measurement should be fine grained enough to differentiate between day and night times as well as direct or indirect sun light (e.g., driving through a tunnel). The thing is provided as virtual sensor within the virtualization environment of UC1 but could be based on a real sensor.
2. (Light) Switch	Providing the ability to switch an actuator between two states whereas the two states are on / off of a light source in UC1 (like a street light that can be turned on and off). The thing

	will be provided as virtual actuator within the virtualization environment of UC1 but could be based on a real one.
3. Car	In UC1 a car can be seen as the composition of multiple car specific things. The things are simulated in the virtual environment and will provide data via smart edge semantic interfaces. The following things are used to create a semantic scene based on sensor fusion. - LiDAR: Provides point cloud measuring in terms of RGBD Video Streams or Images - Ultrasonic Sensors: Short distance object detection - Radar: Detection of objects on roads - Camera: Providing video stream from a car's environment, a car may have multiple cameras - Location: Latitude, Longitude, Altitude, Direction, Speed based on changes in the geolocation over time - Speed: The current speed of the car which could be different to the speed that is based on geolocation data

3.2.2.2 Requirement Analysis for Use Case 2

Similar to use case 1, the semantic requirements for recipes, capabilities and device semantic models for use case 2 are presented in Table 3.3.

For example, use case 2 would like to implement an application such as "Option zone monitoring and optimization of vehicle flow" to monitor and optimize the option zone near the traffic light signals at intersections. This application can be implemented as a recipe where the required capabilities are "getting real-time location and speed of vehicles", "calculating vehicle count in the option zone", "calculating vehicle to vehicle distance and speed" etc. as listed in Table 3.3. These capabilities can be implemented on assets such as Sensor Node (an edge device connected to radars and cameras), Vehicle Node (with built-in sensors and V2X OBU), etc. Therefore, the semantic requirements for this use case are to define the mentioned capabilities, device semantic models for nodes such as sensor node, radar, camera, built-in car sensors, and traffic controller, using standardized domain semantic models.

Table 3.3: Use Case 2: Applications and Their Required Capabilities

Application	**Required Capabilities**
Option zone monitoring and optimization of vehicles flow: Especially when the traffic signal state is yellow, in which case each driver/vehicle in the option zone may decide either to accelerate and pass before the red sign appears, or to brake and slowdown. Such different decisions among vehicles may increase the risk of back collisions near the traffic lights. To tackle such issues, UC2 monitors the option zone traffic	1. Real-time detection of vehicles near the traffic light signal. Detecting vehicles that are moving on specific road and/or specific lane towards the stop line. This needs real-time probe of vehicles at high frequency resolution (10 Hz) 2. Obtain current traffic light signal status (whether it is yellow, green, red, etc.) 3. Calculate real-time traffic indicators per each traffic light: a. Vehicle count in the option zone b. Vehicle to vehicle distance and speed c. Number of approaching vehicles

situation near intersections, decides on the required actions, and sends action commands to the vehicles and traffic lights to optimize the traffic.	d. Queue length (Number of stopped vehicles in the queue or the length in meters) 4. Decide on the required actions for the option zone 5. Send action commands (SPAT message) from the infrastructure to the vehicles 6. Control the traffic lights (change traffic light statuses/colors if needed)

Table 3.4: Use Case 2: Mapping Capabilities to Things (SmartEdge Nodes)

Capability (Numbers from the above table)	**Things (nodes) that have the *skills* to implement the *capability*.**

No. 1	Smart nodes with the required skills:
	• Sensor Node
	• Vehicle Node
	
	Some Details:
	• Passive detection performed by our Sensor Node that has the:
	  o Radar object measurement
	  o Camera object detection
	• Active detection: Connected V2X-enabled vehicles periodically send their geolocation, speed, acceleration, etc. Via V2I communication. Uses:
	  o Built-in car sensors to measure car's location, speed, etc.
	  o V2X on-board unit of the car

No. 2	Smart nodes with the required skills: • Controller Node (Traffic Light Controller)
No. 3	Smart nodes with the required skills: • Traffic node o Metric Twin Some Detail: The Metric Twin sub-component of the Traffic Node calculates traffic indicators based on the data received from the Sensor nodes and Controller node.
No. 4	Smart nodes with the required skills: • Controller Node Some Detail: The decision logic works based on the calculated traffic indicators (by Traffic Node) and the traffic light status.
No. 5	Smart nodes with the required skills:

	• Controller Node Some Detail: The traffic controller node sends SPAT messages via a V2X Module to the Connected Vehicles.
No. 6	Smart nodes with the required skills: • Controller

3.2.2.3 Requirement Analysis for Use Case 3

Similar to use case 2, the semantic requirements for recipes, capabilities and device semantic models for use case 3 are presented in Table. In section 3.2 we explained in detail an example application from use case 3 such as “move product to a mobile rack”. We showed how a recipe can be used to specify the application requirements, goals and tasks. Please refer to Figure and Figure for details.

Table 3.5: Use Case 3: Capabilities and Skills

Use Case	Required Capability	Capabilities that should be implemented on Assets
3	Product mover	• move across floor in 2 dimensions • lift product vertically from underneath • classify objects using camera images • measure distance to surfaces using LiDAR • measure distance to surfaces using stereoscopic camera • receive image data from camera streams • triangulate position and pose of objects based on cameras streams • construct SLAM map from measurement data • construct Semantic SLAM map by fusing SLAM map with object classification • motion planning to move product • motion planning to navigate factory floor • automatically detect obstacles from Semantic SLAM map • avoid obstacles detected in Semantic SLAM map • communicate obstacles to other SMARM nodes • select mobile storage rack based on product type. • forward information on product being moved. • immediately halt if humans detected in local vicinity
3	Mobile storage rack	• store products of specified type in rack slot • record slot of specific product instance in rack • apply rack brake. • detect product in slot

		• provide list of slot and product information • knows next process operational area for products • automatically request storage mover when full to tow to next process operational area
3	Storage mover	• move across floor in 2 dimensions • connect to mobile storage rack • instruct mobile storage rack to release and apply brake • able to tow mobile storage rack • classify objects using camera images • measure distance to surfaces using LiDAR • measure distance to surfaces using stereoscopic camera • receive image data from camera streams • triangulate position and pose of objects based on cameras streams • construct SLAM map from measurement data • construct Semantic SLAM map by fusing SLAM map with object classification • motion planning to navigate factory floor • automatically detect obstacles from Semantic SLAM map • avoid obstacles detected in Semantic SLAM map • communicate obstacles to other SMARM nodes • immediately halt if humans detected in local vicinity
3	Product conveyer	• move product out of processing area for collection • read RFID tag for specific product instance information • request product to be removed from conveyor • forward product specific product instance information
3	Overhead image capture	• capture images using camera • blur human faces • stream images to remote swarm node

3.2.2.4 Requirement Analysis for Use Case 4

Similar to use case 2 and 3, the semantic requirements for recipes, capabilities and device semantic models for use case 4 are presented in Table, Table 3.7 and Table 3.8. Use case 4 is an industrial use case focusing on optimizing the manufacturing process by enabling custom production, anomaly detection and efficient production planning. For these applications, capabilities such as custom configuration of a product, ability to detect anomalies etc are

required. It uses assets such as manufacturing unit, camera, NPU etc to realize the recipes. Semantic requirements for use case 4 are explained in detail in Section 3.2.3.

Table 3.6: Use Case 4: Applications and Required Capabilities

Use Case	Application	Required capabilities
4	customized production Description: Customer can dynamically configure the order of colour blocks for his product on Mendix. There are five colour blocks available for creating a new product on the demo manufacturing unit. They are green, blue, white, red and yellow. The Recipe offers flexibility for a user to dynamically configure the colours and their order for his new product.	• Configuring the colours of block of a product to be manufactured • Manufacture the product. • Place the finished product on the end station. • Notify AGV/user about finished product

Table 3.7: Use Case 4: Capabilities and Corresponding Skills

Use Case	Required Capability	Capabilities that should be implemented on Assets
4	Configuring the colours of block of a product to be manufactured	• Colour blocks • Identify required color block • Lift the block • Identify tray • Place the block in tray
4	Manufacture configured product	• Move the tray
4	Place the product on end station	• Identify the tray containing finished product • Move the product to end station • Notify user to pick up product

The table below further provides the requirements for devices and their characteristics in UC4.

Table 3.8: Use Case 4: Device / Asset with Required Skills and Characteristics

Use Case	Device	Capabilities on Assets / Properties, Actions & Events	Protocol	Service framework	Characteristics of device	Smart Node

4	Robot Arm	Properties • Arm status Actions: • Lift a block, • place a block on a tray, • identify the colour of the block, • scan the block Events: • Block not found notification, • Block placed on the tray notification	OPC UA	Mendix	• Device Fixed/Mobile • Physical dimensions • Weight it can lift etc.	No

3.2.2.5 Requirement Analysis for Use Case 5

Similar to use case 2, 3 and 4, the semantic requirements for recipes, capabilities and device semantic models for use case 4 are presented in Table 3.9, Table 3.10 and Table 3.11. Use case 5 falls under the health care domain. The applications in this use case do not need to be implemented using recipes. However, other semantic models such as device semantic models are relevant to this use case to model assets such as tablet, Nordic thingy52, Raspberry Pi board, pantilt, laser pointer, BLE connectors, mobile app etc.

Table 3.9: Use Case 5: Applications and Required Capabilities.

Use Case	**Application**	**Required capabilities**
5 (HES-SO)	The users of UC5 are intended to be patients (PAT) rehabilitating their neck or individuals undergoing neck sensorimotor assessment, and physiotherapists (PHY). PHY can define and assign the tasks to be performed by PAT. PAT has to put on wearable sensors (number variable according to the needed	• Profiling PAT via tablet/web interfaces • PHY can create sessions and tasks for PAT • The system has to project images/pointers on PAT's surroundings complying with the sessions/tasks created. • The system has to acquire PAT's movement via wearable sensors • The system has to provide run-time feedback to PAT during the tasks' execution (via wearables and tablet) • PHY can annotate the PAT's tasks • The system has to analyze PAT's data • Wearable sensors have to be place-and-play • Wearable sensors can be replaced on-the-fly

	activity) and can select the tasks to be performed on a tablet. PAT is required to follow pointers and/or images projected on their surrounding walls. PAT are profiled and all their sessions/tasks are stored to allow further analysis.	• Wearable sampling frequency and communication rate can be modified at the lunch of the application • Inter-sensor communication has to be seamless and standardized • Wearables Migration should be allowed • Neck movements (angles/speed) should be computed, and acquired-stored

Table 3.10: Use Case 5: Capabilities and Corresponding Skills

Use Case	**Required Capability**	**Capabilities implemented on Assets**
5 (HES-SO)	A. Profiling PAT via tablet/web interfaces	A.1 -The application should allow PHY to enter new patients. A.2 - PAT should have a profile on the system and be able to login and follow the assigned tasks.
	B. PHY can create sessions and tasks for PAT	B.1 - PHY can create tasks: assigning them a name, description, duration, etc. Defining a task comprises specific motions (e.g., angles, directions, pace, ...)
	C. The system has to project images/pointers on PAT's surrounding walls complying with the sessions/tasks created.	C.1 - A laser pointer mounted on a pan-tilt and connected to an embedded board has to enact the movements specified in the tasks.
	D. The system has to acquire PAT's movement via wearable sensors	D.1 - PAT has to follow with their gaze the pointers (executing the task). Wearable sensors placed on PAT will acquire inertial data and send them to a SmartEdge node (I.e., a tablet).
	E. The system has to provide run-time feedback to PAT during the tasks' execution (via wearables and tablet)	E.1 - The tablet has the duty of coordinating the data coming from the self-organized wearables and reconstruct PAT movements.
	F. PHY can annotate the PAT's tasks	F.1 - Once the acquired inertial data will be processed and form a trajectory (space/time), PHY should be able to see and assess/annotate it.
	G. The system has to analyze PAT's data	G.1 - besides PHY's annotation, the system should perform PAT's tasks analysis via pre-defined symbolic rules
	H. Wearable sensors have to be place-and-play	H.1 - The wearables (smartEdge nodes) will have to be turned on an placed seamlessly. It means that after their

		positioning, the wearables must be able to understand their role (i.e., via an inverted kinematic chain).
	I. Wearable sensors can be replaced on-the-fly	I.1 - Sensors might fail (i.e., flat battery) and a seamless sensor replacement on-the-fly must be allowed.
	J. Wearable sampling frequency and communication rate can be modified at the lunch of the application	J - The streaming and messaging frequency of the wearables might depend on the assigned task. Hencheforth, the streaming rate must be variable.
	K. Inter-sensor communication has to be seamless and standardized	K.1 - Wearables must be able to talk with a SmarEdge Orchestrator (to be investaged if a p2p communication is needed).
	L. Wearables Migration should be allowed	L.1 - Besides being replaced, wearable (or any sensor involved in UC5), might be required to migrate from one orchestrator to another and to change the performed tasks.
	M. Neck movements (angles/speed) should be computed, and acquired-stored	M.1 - The SmartEdge orchestrator must be able to retrieve, filter, align, and process the inertial data received by the wearables.

Table 3.11: Use Case 5: Skills and Assets That Implement Them

Use Case	**Capabilities implemented on Assets**	**Hardware/Software Device/Asset**
UC5 (HES-SO)	A.1	H: tablet, PC S: mobile-interface, web-interface, SmartEdge Orchestrator & Node
	A.2	H: Tablet, PC S: mobile app, back-end server
	B.1	H: Tablet, PC S: mobile app, back-end server
	C.1	H: tablet, Raspberry Pi board, pantilt, laser pointer, BLE connectors S: mobile app, and SmarEdge Orchestrator & Node code
	D.1	H: tablet, Noprdic thingy52, Raspberry Pi board, pantilt, laser pointer, BLE connectors S: mobile app, and SmarEdge Orchestrator & Node code
	E.1	H: tablet, Noprdic thingy52, Raspberry Pi board, pantilt, laser pointer, BLE connectors S: mobile app, and SmarEdge Orchestrator & Node code
	F.1	H: tablet, PC S: mobile & web app, and SmarEdge Orchestrator & Node code
	G.1	H: tablet, PC S: mobile & web app

	H.1	H: tablet, Nordic Thigy52 S: mobile app, SmartEdge Orchestrator & Node code
	I.1	H: tablet, Nordic Thigy52 S: mobile app, SmartEdge Orchestrator & Node code
	J.1	H: tablet, Nordic Thigy52 S: mobile app, SmartEdge Orchestrator & Node code
	K.1	H: tablet, Nordic Thigy52, Raspberry Pi S: mobile app, SmartEdge Orchestrator & Node code
	L.1	H: tablet, Nordic Thigy52, Raspberry Pi, S: mobile app, SmartEdge Orchestrator & Node code
	M.1	H: tablet, Nordic Thigy52 S: mobile app, SmartEdge Orchestrator & Node code

3.2.3 Recipe Model for Use Case 4

Use case 4 is an industry use case that demonstrates flexible manufacturing of a product. Therefore, it uses the well-established industry standard called OPC UA (Open Platforms Communication Unified Architecture). The standard provides both the communication protocol and semantic models for modelling the information for interoperability. Therefore, in this use case we use both OPC UA communication protocol and its semantic models which are provided by OPC Foundation as the companion specifications and their information models.

OPC UA FX:

OPC UA FX is one of the standards developed by OPC Foundation. OPC Unified Architecture Field eXchange (UAFX)[2] extends the OPC UA model to enable field device interconnection. It aims to facilitate *controller-to-controller* interactions. Additional interactions – *controller-to-device*, *device-to-device*, and *controller-to-compute* – are intended to be addressed in future releases. The current objective is to provide a standardized field component model (information and interfaces) alongside an information exchange model. The emphasis is on enabling timely data delivery, security, and functional safety. UAFX defines an asset component that can represent a security key or software license. Such assets can be referenced by and/or installed within a controller.

OPC UA FX introduces several significant model entities as shown in Figure :

AutomationComponent

Represents an entity that performs one or more automation functions (e.g., a representation of one or more related field devices).

FunctionalEntity

An information model identifying a set of related FX input data, output data, configuration data, diagnostic information, and methods to manipulate and/or share the data.

Connection

A logical relationship between *FunctionalEntities*.

ConnectionManager

Responsible for establishing (and removing) *Connections*.

Asset

Represents a component with a lifecycle (e.g., versioning).

[2] OPC UA FX is an outcome of the OPC UA Field-Level Communication (FLC) Initiative.

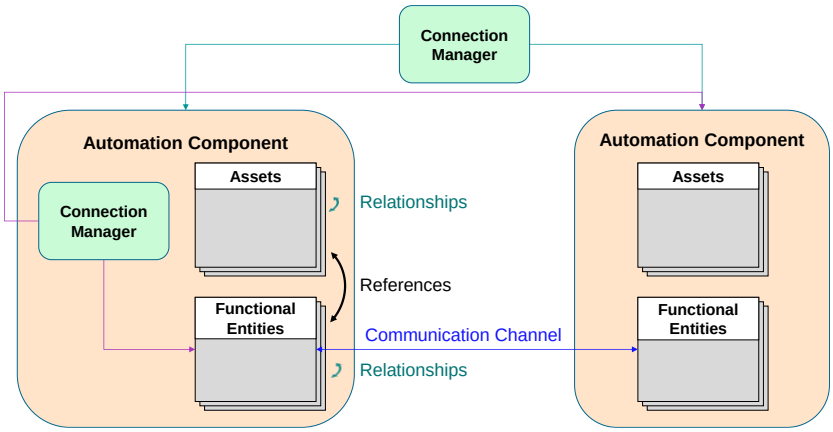

Figure 3.7: FX Model Entities

The significance of these entities is in their ability to provide a level of model specificity previously missing from OPC UA. This specificity is well aligned with concepts needed to support a Skills-oriented perspective.

AutomationComponent

An AutomationComponent is an entity that performs one or more automation functions and provides connection capabilities. The AutomationComponentType is composed of two major sub-information models, asset model and functional model. It also provides information related to offline engineering, general metadata such as communication capabilities and the health status. Figure provides an illustration of this model.

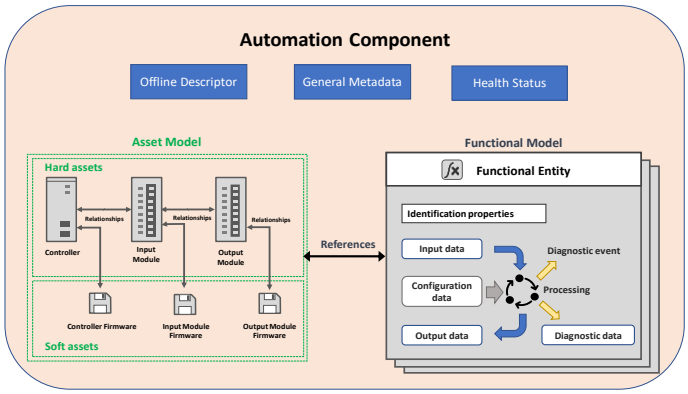

Figure 3.8: A conceptual overview of an OPC UA FX automation component.

The **AutomationComponent** is the base model for an OPC UA FX device/controller/PLC/etc. It includes information related to current asset(s), the available functionality, the capabilities (including communication related capabilities) and any offline information. The AutomationComponent provides for grouping *Asset* instances and **FunctionalEntity** instances. It exposes a Method that is used to establish logical connections between instances of **FunctionalEntity**. An overview of the AutomationComponent information model is illustrated in Figure .

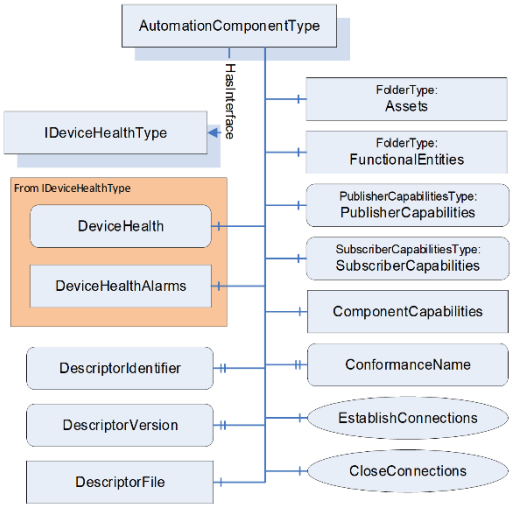

Figure 3.9: An overview of the AutomationComponent information model.

DeviceHealth and DeviceHealthAlarms provide an overall health status of the **AutomationComponent** instance. This includes a summary of all included ***Assets*** and **FunctionalEntities**. Each *Asset* and **FunctionalEntity** might include additional diagnostics that are more specific. For a complete description see OPC 10000-100.

The FunctionalEntity information model is the base model for describing all functionalities in an OPC UA FX information model. Functional entities encapsulate logical functionality. Functional entities are designed in a way that they can describe functionality of any complexity ranging from the acquisition of a single measured value to controlling an entire machine or production line. Functional entities can also be preconfigured and fixed (e.g., a device such as a drive) or they can be dynamically created during engineering or at run time. A logical functionality is viewed as an identifiable process with properties, inputs, outputs, and a configuration that generates events and diagnostic data (see Figure).

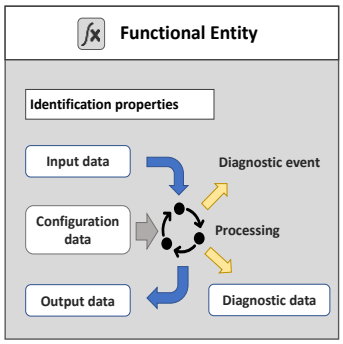

Figure 3.10: Key aspects of the Functional Entity model.

A functional entity can interact with other functional entities by exchanging data. It provides methods for the manipulation or sharing of the data. The exchange of data may be for control or monitoring purposes. Interactions are represented as logical connections (which in turn are modeled as a *Connection*. Inputs and/or outputs can be arranged into logical groups to ease configuration, simplify establishing interactions, and/or to restrict access. Inclusion in one group does not preclude inclusion in other groups. Functional Entity information model is described in detail in Figure .

The functional entity information model is illustrated by Figure with the key members described as follows:

Identification properties:

- Identity - When establishing a logical connection between functional entities, it may be essential to confirm identity, meaning to check that the other functional entity is the one that is expected. Therefore, functional entities provide the information and methods to perform such identity verifications.
- Input data - describes the values that may be provided from another functional entity and consumed by this functional entity.
- Output data - describes the values that are provided by the processing of this functional entity and are available for other functional entities to consume.
- Configuration data - describes any value that is used to set up and configure functionality.
- Diagnostic data - The functional entity maintains information related to the status of its functionality, including the status of any logical connections. This may include the generation of events or alarms related to problems or issues encountered by the functional entity.

A functional entity can be of varying degrees of complexity and can represent different granularity and abstraction levels, from primitive functionality to an entire application. It is expected that the **FunctionalEntity** model will be subtyped by other models. It can have sub-functional entities and relationships to other objects defined in this model or in other models.

For example:
There are primitive functional entities that only generate output data like a temperature sensor or only receive input data like a relay.
More complex functional entities like a motion axis can receive control data, perform a calculation or action, provide status data, and have different operation modes, including closed-loop controls.

There are also functional entities on the process application level representing an entire application such as a paper machine or a boiler, where the functional entity has multiple nested sub functional entities.

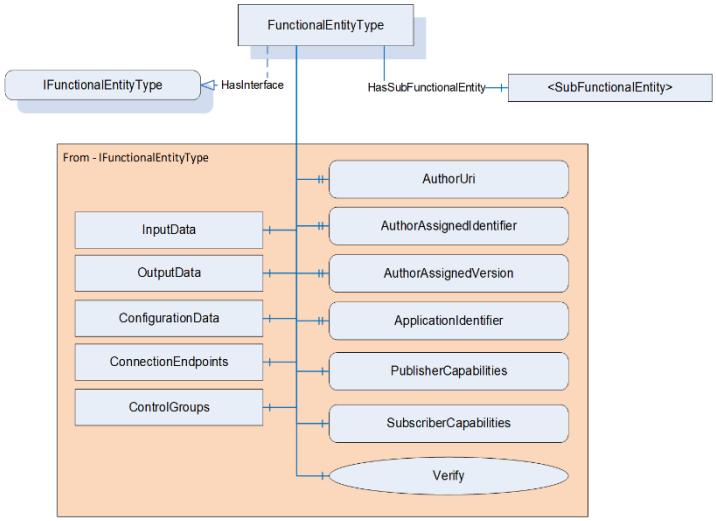

Figure 3.11: Overview of the FunctionalEntity information model.

OPC UA Field level exchange. It describes the field devices, their attributes and functionalities provided by them in a domain agnostic way. Therefore, we use the OPC UA FX information model as a basis to define capability in UC4. It means, the capability model in UC4 is based on OPC UA FX, which is shown in Figure .

Capability Definition:
Capability is an implementation-independent description of the function of a resource to achieve a certain effect in the physical or virtual world. Capability represents required resource functionality from a resources system such as tool, machine, production cell etc. to fulfill a production function for a manufacturing process unit. In our view, so defined Capability matches the role of FunctionalEntity in OPC UA FX. Therefore, our CapabilityType extends FunctionalEntityType as shown in Figure .
In a general sense, Capability defines an automation function. As such, it may have inputs, outputs, configuration parameters, access-rights metadata etc. This information is captured by FX FunctionalEntityType, see Figure . FunctionalEntities encapsulate logical functionality, which can include function blocks, IO module functionality, drive functionality, sensor functionality, actuator functionality, or more complex logical items. From that point of view, FunctionalEntities can be used to realize Capabilities.

The Capability also defines the range of parameters and dependencies or constraints that are partially derived from a product definition and partially from the manufacturing process. The product definition aspects can be aspects of geometry, tolerance, quality inspection etc. For some of this information FX model needs to be extended with other OPC UA Companion Specifications and semantic models (ontologies).

Capability is the key to enable capability-based (continuous) engineering. Capability needs to have description in a machine-readable format and in the right level of abstraction.

Capability, as realized with FX FunctionalEntity, provides the right level of abstraction. It is given in a machine-readable format. Moreover, it is standardized by the OPC Foundation. It can be extended with other models, e.g., OPC UA Companion Specifications, and as such may enhance interoperability across different vendors. As shown in Figure 3.13, capabilities for different companion specifications e.g., Robotics, PackML, Machine Vision, Machine Tools etc. can be created using the FX capability model which enables interoperability across vendors.

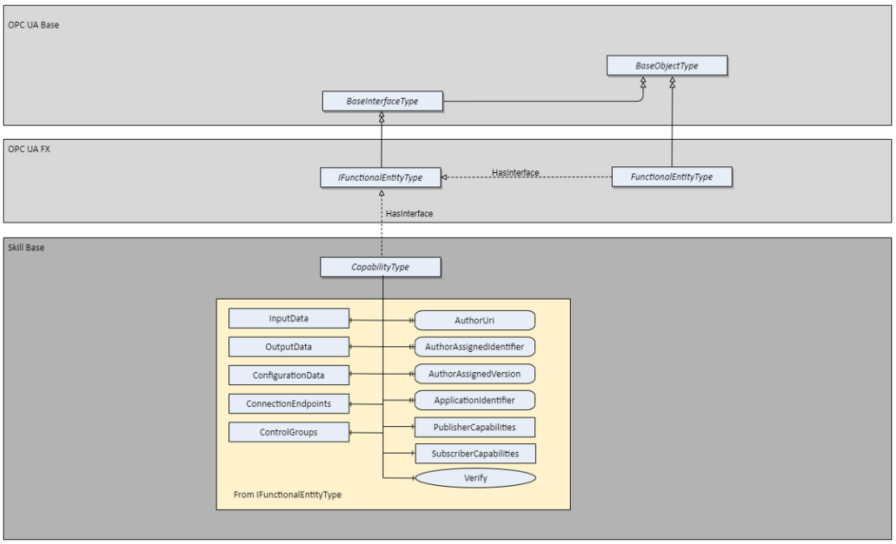

Figure 3.12: CapabilityType Definition using OPC UA FX

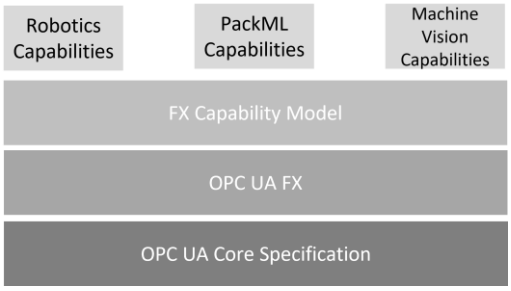

Figure 3.13: Overview of OPC UA Information models layer cake for OPC UA Capabilities

As we mentioned before, in SmartEdge project capabilities and Recipes for UC4 are created using FX capability model. Here we would like to present an example of a recipe for UC4.

The sample application that we would like to showcase here is customized production on the manufacturing unit with production planning. Using this application, a user can configure and customize the product that should be manufactured. The recipe lets the user configure the order and it will check if the manufacturing process of the new product can be fulfilled with the existing resources available on the unit. If the resources are not available, then they will not be manufactured, and the recipe sends a message that the product cannot be manufactured. The recipe is illustrated in Figure .

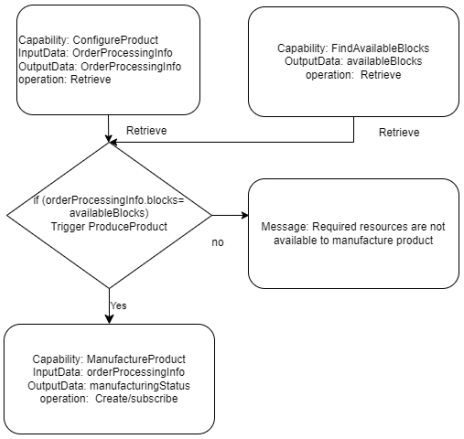

Figure 3.14: Sample Recipe for UC4

3.3 DOMAIN SPECIFIC ONTOLOGIES

In this section we review few domains specific ontologies that are relevant for SmartEdge use cases. Our aim is to re-use existing, well defined domain specific ontologies as much as possible. The static knowledge about an environment can also be modelled using existing domain ontologies. If the suitable ontologies do not exist then new ontologies can be introduced in order to model the external knowledge or static knowledge that is required for the SmartEdge use cases (this addresses the requirements CSI-005 and CSI-008). These ontologies can be used in SmartEdge in multiple ways. They can be layered on top of device semantic descriptions; they can be used in a recipe to define capability constraints. They can be used in SmartEdge ontology (for example to model specific characteristics of an AMR.) etc., to allow domain specific concepts to be modelled in the knowledge graph, this addresses the requirement CSI-015.

3.3.1 The IEEE Standard for Autonomous Robotics

The IEEE Standard for Autonomous Robotics (AuR) Ontology (IEEE, IEEE Standard for Autonomous Robotics (AuR) Ontology, 2022) extends the Core Ontology for Robotics and Automation (CORA) (IEEE, IEEE Standard Ontologies for Robotics and Automation, 2015) to provide a standardized representation of knowledge specific to autonomous robotics. This extension enables the clear identification and understanding of the components essential for building autonomous systems capable of functioning in diverse environmental conditions. The components of an ontology include individuals, classes, relations, and axioms, typically expressed in first-order logic (FOL) or web ontology language (OWL). In IEEE standard, ontologies are categorized as upper-level, reference, domain, or application, as illustrated in Figure 3.15.

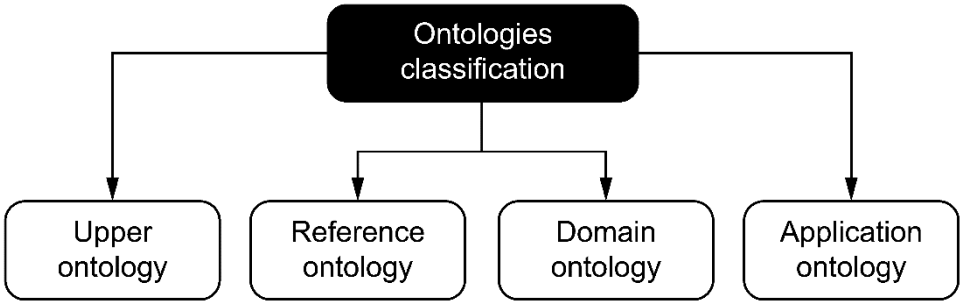

Figure 3.15 IEEE Standard for Autonomous Robotics (AuR) Ontologies Classification

Upper-level Ontology:

- Focuses on widely applicable concepts like object, event, state, and quality, along with high-level relations.
- Addresses fundamental and generic concepts, applicable across various domains.

Reference Ontology:

- Concentrates on a specific discipline, ensuring high reusability within that field.
- Provides a standardized framework for a particular field.

Domain Ontology:

- Focuses on a more limited area, such as autonomous or collaborative robotics.
- Contains vocabulary specific to a domain, covering concepts, relationships, activities, theories, and principles.
- Specializes concepts from upper-level and reference ontologies.

Application Ontology:

- Includes definitions necessary for modeling knowledge in a particular application, such as a robot grasping system or SLAM.
- Tailored to the requirements of a specific application domain.
- Provides a practical implementation of ontological concepts for real-world use.

In SmartEdge, AuR is used to model data elements related to SLAM (Simultaneous Localization and Mapping). SLAM uses collaborative maps to capture the surroundings and determine the positions of multiple robotics in UC3. Leveraging ontologies from the AuR ontology, particularly application ontologies, SmartEdge enhances SLAM by providing a tailored knowledge representation for the simultaneous localization and mapping process, called Semantic SLAM. These ontologies ensure precision in modelling intricate relationships and concepts, contribute to interoperability through reference standards, and facilitate seamless integration with various robotics-related disciplines. Semantic SLAM would include following ontology-based data elements (Cornejo-Lupa, 2021):

- **Robot Information:** Captures the robot's characteristics, capabilities, and location within the environment.
- **Environment Mapping:** Represents the surrounding objects, their features, and their positions.
- **Timely Information:** Records the robot's movements and the duration of its actions.
- **Workspace Information:** Defines the overall characteristics of the mapped area and domain-specific entities.

3.3.2 OPC UA Information Model for Robots

The OPC UA Robotics Companion Specification extends OPC UA standard to the field of robotics. [3, 4] In essence, it provides a framework for seamless communication and integration between different robotic devices and systems. It defines standardized methods for exchanging information related to robotic capabilities, status, and control commands. This specification

[3] https://opcfoundation.org/markets-collaboration/robotics/
[4] https://reference.opcfoundation.org/Robotics/v100/docs/

aims to create a common language for robots and automation systems to understand and interact with each other, promoting a more flexible and collaborative industrial environment.

The OPC UA Robotics Companion Specification includes an Information Model that serves as a structured representation of the data and functionalities related to robotics within the OPC UA framework.

The Information Model defines a standardized way to represent information about robotic devices, their components, capabilities, and states. It organizes data into a hierarchical structure, allowing for a clear and consistent description of the robotic system. This model covers aspects such as kinematics, dynamics, and other relevant properties of robots.

By using the OPC UA Information Model, robotic devices can share a common understanding of their environment and capabilities. This enables seamless communication between different robotic systems, as well as with higher-level automation and control systems. It promotes interoperability by ensuring that all devices adhere to the same data representation standards, fostering easier integration and collaboration in industrial settings.

3.3.3 OPC 30050: PackML - Packaging Control

The OPC UA PackML (Packaging Machine Language) Companion Specification[5] extends the OPC UA standard to the domain of packaging machinery, providing a standardized way for these machines to communicate and integrate within industrial systems. It provides the following models for packaging machinery; however it can be used to define communication between machines in a manufacturing unit (which may not be packaging machines). For example: communication between different modules in the unit, communication between a conveyor belt and a module etc.

State Model:

Defines a standardized state model for packaging machines based on the PackML state model.

Represents different states of the packaging machine, such as Idle, Starting, Execute, Stopping, and Aborting.

Information Model:

Describes the structure and semantics of data related to packaging machines.

Includes information about production data, equipment status, and other relevant parameters.

Hierarchical organization for clear representation and understanding of the machine's components and their relationships.

Methods:

Specifies standardized methods for controlling and interacting with the packaging machine.

Methods include commands for starting, stopping, resetting, and aborting the machine, among others.

Event Model:

Defines events and alarms related to the packaging machine's operation.

[5] https://reference.opcfoundation.org/PackML/v101/docs/

Provides a standardized way to notify systems about changes in state, errors, or other significant occurrences.

Data Types:

Specifies data types that represent common concepts in packaging machinery, ensuring consistency in data exchange.

OPC UA PackML Information Model:

Production Data:

Includes information about production counts, speeds, and other performance metrics.

Equipment Status:

Describes the current status of the packaging machine, such as running, stopped, or in an error state.

Material Status:

Covers information related to materials used in the packaging process, including availability and consumption.

Job Data:

Provides details about the current job being executed by the packaging machine, including job ID, description, and progress.

Performance Metrics:

Encompasses data related to the efficiency and effectiveness of the packaging machine, such as OEE (Overall Equipment Effectiveness) parameters.

By incorporating the OPC UA PackML Companion Specification and its detailed Information Model, packaging machines can communicate seamlessly with other devices and systems, enabling better coordination and control within industrial automation environments.

3.3.4 OPC 40100-1: Machine Vision - Control, Configuration Management, Recipe Management, Result Management

A machine vision system is any computer system, smart camera, vision sensor or even any other component that has the capability to record and process digital images or videostreams for the shop floor or other industrial markets, typically with the aim of extracting information from this data. Digital images or video streams represent data in a general sense, comprising multiple spatial dimensions (e.g., 1D scanner lines, 2D camera images, 3D point clouds, image sequences, etc.) acquired by any kind of imaging technique (e.g., visible light, infrared, ultraviolet, x-ray, radar, ultrasonic, virtual imaging etc.). With respect to a specific machine vision task, the output of a machine vision system can be raw or pre-processed images or any image-based measurements, inspection results, process control data, robot guidance data, etc. Machine vision therefore covers a very broad range of systems as well as of applications.

The OPC UA Machine Vision companion specification [6][7] supports these broad range of applications mentioned above. It provides an information model to define the data structures to support these applications of a machine vision system. The description covers all aspects relevant for operation.

Interaction of the client with the vision system
A vision system usually has the role of an OPC UA server, i.e. its states are exposed via an OPC UA server. This is what in this specification is described and defined. The client system can control the vision system via OPC UA. The vision system may also be controlled by a different entity through a different interface. The vision system reports important events – such as evaluation results and error states – automatically to a subscribed client. However, the client can query data from the vision system at any time.

State Machine
The state machine model is an abstraction of a machine vision system, which maps the possible operational states of the machine vision system to a state model with a fixed number of states. Each interaction of the client system with the vision system depends on the current state of the model and the state and capabilities of the underlying vision system. State changes are initiated by method calls from the client system or triggered by internal or external events. They may also be triggered by a secondary interface. Each state change is communicated to the client system.

Recipe Management
The properties, procedures and parameters that describe a machine vision task for the vision system are stored in a recipe. Usually there are multiple usable recipes on a vision system. This specification provides methods for activating, loading, and saving recipes. Recipes are handled as binary objects. The interpretation of a recipe is not part of this specification. For a detailed description of Recipe Management, please refer to B.1. Result Transfer The image processing results are transmitted to the client system asynchronously. This transmission includes information on product assignment, times, and statuses. The detailed data format of a result is not included in this specification.

Error Management
There is an interface for error notification and interactive error management.

3.3.5 Domain Models for Smart Traffic

We have adopted the following ontologies to define a customized domain-specific semantic model for real-time traffic management in Use Case 2. The ontology diagrams together with the object naming scheme are presented in D5.1 [8].

- The components defined by the ASAM OpenDRIVE standard [9] (not an ontology) can be used in our use-cases to describe the road network as a composition of interconnected individual sections. Elements include road-segments, lanes, junctions, and features such

[6] https://reference.opcfoundation.org/MachineVision/v100/docs/
[7] https://opcfoundation.org/markets-collaboration/machine-vision/
[8] SmartEdge - D5.1 Design of Low-code Programming tools for edge intelligence.docx
[9] https://www.asam.net/standards/detail/opendrive/

as signals. In addition to the linked road segments, the lanes between roads are also connected which can be used in simulated traffic.

- At the same time, ASAM OpenXOntology [10] and ASAM OpenLABEL are being developed in parallel to define ontologies for road network and traffic, and to define taxonomies for labelling of the network elements.
- In addition, H. Qiu et al have proposed a non-standard but well-researched ontology highly relevant to UC-2, called "Ontology-based digital map integration" [11]
- Finally, to include our traffic sensors and edge devices in the model, SOSA [12] and SSN [13] OWL ontology standards (by W3C) can be used to describe sensors, sensing, measurement capabilities of sensors, the sensing observations results, and sensor deployments. SOSA is the result of rethinking SSN.

3.4 Standardized Semantic Interfaces

In this section we review few standards and open-source implementations, which will be used in the implementation of Standardized Semantic Interfaces, implementing the functional requirement CSI-002.

3.4.1 OPC UA

OPC Unified Architecture (OPC UA) is a set of standards[14] designed as an interoperability in Industrial Automation. OPC UA includes a platform independent service-oriented architecture and a cross-platform standard for data exchange from sensors to cloud applications (IEC 62541). It is developed by the OPC Foundation. It ranges from field devices up to cloud-based infrastructure, regardless of diverse hardware platforms and operating Systems.

While there are numerous communication solutions available, OPC UA has key advantages: security model, multiple fault-tolerant communication protocols, and an information modelling framework (semantics) that allows application developers to represent their data in an object-oriented way.

OPC UA has a broad scope which aims to offer economies of scale for application developers. This means that a larger number of high-quality applications at a reasonable cost is available. For example, when combined with semantic models such as Asset Administration Shell, OPC UA makes it easier for end users to access data via generic commercial applications.

OPC UA defines a protocol to exchange the data in accordance with the proposed architecture. The standard addresses a lot of aspects such as platform independence, communication patterns, security, extensibility, and so on.

OPC UA standardizes a set of comprehensive information models. OPC UA information models (companion specifications) are organized with a layered approach, so that each layer provides

[10] https://www.asam.net/standards/asam-openxontology

[11] https://www.semantic-web-journal.net/content/ontology-based-digital-map-integration

[12] https://www.w3.org/2015/spatial/wiki/SOSA_Ontology

[13] https://www.w3.org/2005/Incubator/ssn/wiki/SSN https://www.w3.org/2005/Incubator/ssn/ssnx/ssn

[14] https://reference.opcfoundation.org/

additional information, ranging from basic OPC UA concepts up to domain-specific information, even vendor-specific information, and so forth. It is not a single model, but rather an information stack that defines how the data is looked up, read, and written, and further, how methods are executed, notifications on data and events are handled, and so forth. Companion specifications exist for various domains such as, for example, for robotics, product packaging, computer vision, machine tools, and many more.

3.4.2 W3C WoT

Digital Twins are software abstractions for the IoT. W3C has released the standard called Web of Things (WoT), which enables interoperability across IoT platforms and application domains. The standard provides the concept of Digital Twin for things connected to the Web. It includes the following specifications: WoT Architecture, WoT Thing Description, WoT Discovery, WoT Security, and WoT Scripting API.

In the scope of this project, we will focus mostly on WoT Thing Description (TD). A TD provides general metadata of a Thing, as well as metadata about its functions (Interactions), protocol usage, security mechanisms, links to other Things etc. Thing's Interactions are specified in a so-called Interaction Model. The model defines three types of so-called Interaction Affordances: Property, Action, and Event. They can be manipulated via a RESTful API.

TD Properties can be used for sensing and controlling parameters, such as getting the current value or setting an operation state. They expose an internal state of a Thing (its data points) that can be, e.g., directly retrieved via GET method of the HTTP protocol or optionally modified via HTTP's PUT method.

TD Actions can model invocation of physical (and hence time-consuming) processes, but can also be used to abstract RPC-like calls of existing platforms. They are functions that may manipulate an internal state of the Thing, e.g., to change states that are not exposed via Properties, modifying multiple Properties, change Properties over time or with a process that should not be disclosed. HTTP's POST is the default method for invoking actions on a URI resource.

TD Events provide a mechanism that enables a Thing to asynchronously push messages. They are used for the push model of communication where notifications, discrete events, or streams of values are sent asynchronously to the receiver. These messages are not stating but rather state transitions (events). Events could be triggered by internal state changes that are not exposed as Properties. Events must follow a consistent delivery approach to ensure that all occurred events are delivered. To that end subscriptions are utilized with HTTP's long polling sub-protocol and enable a sensor to provide a steady feed of data.

A Thing Description is extendable by additional vocabulary terms and ontologies. This mechanism is important when creating domain-specific TDs. For example, it is possible to use a domain-specific ontology (see Section 3.3) to enrich a TD with a standardized semantic vocabulary. A TD must be represented in JSON-LD when additional vocabularies are used.

3.4.3 DDS

The Data Distribution Service (DDS) is an open standard middleware protocol designed for high-performance, distributed, real-time systems. DDS provides a publish-subscribe model for sending and receiving data, events, and commands among the nodes of a network. It is particularly suited for systems where timely and efficient delivery of data is crucial. The protocol defines a high-level API and data model that abstracts the details of network programming, allowing developers to focus on the logic of their applications rather than the intricacies of

network communication. DDS is widely used in industries such as defence, air traffic control, robotics, and large-scale Internet of Things (IoT) applications, and is a protocol specification maintained by the Object Management Group (OMG). There are a number of commercial and open-source implementations including: RTI Connext DDS, eProsima Fast DDS, Eclipse Cyclone DDS, and OpenDDS.

ROS, which stands for Robot Operating System, is a flexible and collaborative framework for building complex robotic systems. The original version of ROS used its own publisher-subscriber message-passing interface for communication between different parts of a robot or between different robots. The message-passing interface is a type of Message Oriented Middleware (MOM) that sits on top of the network layers, such as TCP/IP or cross memory services, if running on the same host. The latest version of ROS, ROS2, replaced its own middleware communications bus with DDS.

DDS works well on low latency low packet loss networks supporting real-time systems, but can struggle when operating over low bandwidth or high latency networks, such as WiFi. Here sometimes unreliable networks are unavoidable, e.g., for mobile robots running ROS 2. In these circumstances DDS can be paired with a more resilient MOM designed to operate consistently over these types of networks. In these cases, DDS is used for the local real-time control and communications, where reliable low latency is required between individual ROS2 nodes, and the more resilient MOM used for high-level coordination and data transfer communications between robots, i.e., each robot is a separate DDS fabric interconnected by another resilient MOM.

3.4.4 Zenoh

Zenoh is a Message Oriented Middleware (MOM) for distributed systems that need to operate over low bandwidth high latency networks, such as WiFi, and provides features such as discovery, routing, and data storage. It is designed to be highly extensible and composable, allowing developers to select the features they need for their application and develop their own custom MOM extensions. Zenoh is written in Rust, but also has a C implementation called Zenoh Pico that is designed to run on micro-controllers, such as Arduino.

The Zenoh ROS Bridge forwards ROS system messages to other systems connected to Zenoh. It is possible for the bridge to subscribe to ROS topics, receive those messages and forward them to the other systems. The same works in reverse for the ROS system to receive published messages. Configuring the topics and messages may be a manual process, but the number of message types that would need to be forwarded over the bridge would be far less that within the ROS system itself.

It is possible to use Zenoh as a bridge between multiple ROS systems. This might seem to be counterintuitive, as we could connect them together directly using a ROS bridge, but connecting two ROS systems can create namespace conflicts. Zenoh handles conflicts by automatically adding a scope prefixing to topics and messages, which avoids namespace conflicts. It can even support wildcards for the scope prefix to receive all messages on a topic regardless of the ROS system that published the message. The Zenoh ROS bridge uses the Cyclone DDS implementation of DDS, and other DDS implementations may not be supported.

If a Zenoh router is installed in the Zenoh communications fabric, applications can communicate in several different modes. Routers are typically installed to communicate over more complex network topologies, the routers forming a backbone for the Zenoh communications fabric. The

applications can communicate either in client mode where all communication goes through the Zenoh router, or peer mode where applications communicate peer-to-peer on the local multicast network but through the router to reach applications on more distant parts of the fabric.

3.4.5 C-V2X

At its current stage, UC-2 has envisioned several sources of data to monitor and control real-time traffic at intersections: a) Radars, b) Cameras, c) Traffic Light Controller, d) V2I messages (measurement data from instrumented vehicles).

Radars send raw data (lacking semantic annotation) to the Sensor Node edge device installed near each intersection. Sensor Node converts the raw sensing data into JSON string with use-case-specific semantic annotations. These converted messages can be sent to other smart traffic nodes via custom MQTT topics. The JSON string messages are also ready to be converted to SmartEdge common ontology format that is understood by other SmartEdge nodes.

Connected vehicles and the infrastructure (including Sensor Node, Controller Node) communicate using standard V2X protocols such as SPaT, MAP, BSM, CAM, that do not hold semantic annotations. These messages are originally in binary format to achieve high performance in networking and computation. For instance, the V2X roadside units (RSUs) enable Sensor Nodes and Controller Nodes to receive V2I BSM/CAM messages from the cars moving near each intersection, providing information about car situations such as accurate geolocations and speeds. On the other hand, the onboard units (OBU) installed inside connected vehicles enable them to receive SPaT and MAP messages from a Controller Node to understand road geometry, rules and suggested driving actions. Similar to the raw radar data, the V2X messages need to be converted from binary non-semantic format to the semantic format defined in SmartEdge and presented as JSON and JSON-LD/RDF strings, before they can be used in the context of UC2. Therefore, *converter* components need to be implemented as well. The proprietary V2X OBUs and V2X RSUs considered for the project already come with SDK for converting the received binary messages into JSON strings. Having this conversion process, it will be possible for us to compare and combine the V2I messages with the object detection data from radars and cameras.

Finally, to brief on the V2X standards, the SAE J2735 [15] and ETSI EN 302 637-2 [16] define standard V2X message types and their data format. SAE's BSM stands for basic safety message, where vehicles periodically communicate their state (location, speed, etc.) at 10Hz to support safety applications such as collision avoidance. Similarly, ETSI's Cooperative Awareness Message (CAM) [17] though having a different message format, has the same application as BSM by communicating vehicle information needed for safety applications. On the other hand, Signal Phase and Timing (SPaT) messages are sent by the Control Nodes that also control traffic signal lights. SPaT messages inform the traffic signal phase and timing state to the nearby vehicles. In addition, Control Nodes can send MAP messages to communicate the road lanes information at

[15] https://www.standards.its.dot.gov/Factsheets/Factsheet/71

[16] https://www.en-standard.eu/etsi-en-302-637-2-v1-3-0-intelligent-transport-systems-its-vehicular-communications-basic-set-of-applications-part-2-specification-of-cooperative-awareness-basic-service/

[17] https://forge.etsi.org/rep/ITS/asn1/cam_en302637_2

each intersection as well as to outline lane geometry as closed polygons. Each lane record can include lane type, connecting lane, related signal group, and allowed manoeuvres at the stop line implying traffic rules.

3.4.6 MQTT with SparkPLug B

MQTT[18] is an OASIS standard. It is a lightweight, publish/subscribe messaging protocol ideal for connecting remote devices (e.g., in IIoT applications). MQTT has a small code footprint. This design allows data to move within a challenging communications environment with resource constraints or limited network bandwidth. MQTT is widely used for device messaging in IIoT communications, with millions of devices leveraging its capabilities.

Publish–subscribe[19] is a messaging pattern where publishers categorize messages into classes that are received by subscribers. This is contrasted to the typical messaging pattern model where publishers send messages directly to subscribers. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

Main characteristics of MQTT are:

- Lightweight - Publish–subscribe architecture is decoupled between a broker and clients. MQTT clients are very small and can be implemented on small microcontrollers. Brokers can be implemented on different kinds of machines (on edge or cloud). MQTT message headers are small to optimize network bandwidth.
- Reliable – Messages are delivered within 3 defined quality of service levels: 0 - at most once, 1- at least once, 2 - exactly once.
- Bi-directional Communications - MQTT enables messaging between device to cloud and cloud to device.
- Support for Unreliable Networks - MQTT supports persistent sessions, which reduces the time to reconnect when an IoT device is connected over unreliable cellular networks.
- Scalability - MQTT can scale to connect with millions of IoT devices.
- Security - MQTT makes it easy to encrypt messages using TLS and authenticate clients using modern authentication protocols, such as OAuth.

MQTT effectively enables device-to-device communication. However, MQTT messaging provides zero context about shared data. MQTT is effective in sharing IoT data, but it is less effective in the management of data, i.e., in managing which data needs to be shared. Mere provision of topic names with some basic structures of topics is not sufficient in complex IIoT applications. The payload can be anything and the message can be anywhere. Thus, it is a challenge to find out what data (exchanged over MQTT) originates from which device, and in which context this data is to be used.

Sparkplug[20] has emerged to extend MQTT for this purpose. It provides the context of industrial data, which is necessary for IIoT architectures and systems when exchanged via MQTT. Therefore, Sparkplug is seen as the main building block in MQTT, which extends operational technology (OT) data with context for seamless integration with information technology (IT).

[18] https://mqtt.org/
[19] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
[20] https://sparkplug.eclipse.org/

Sparkplug is an open-source specification[21] hosted at the Eclipse Foundation, which aims in the future to become an ISO standard.

Sparkplug B introduces the structure for MQTT topics. The structure enables logical grouping of messages, and thus improves the data management. Payloads are binary messages encoded in Google Protocol Buffers[22]. It is also possible to use JSON encoding (for human representation). User defined data types (UDTs) can be defined via special templates. This is said to be a mechanism for defining semantic structures that, for example, can be found OPC UA companions too.

3.5 Standardized Semantic Interfaces in SmartEdge

SmartEdge enables seamless integration of SmartEdge devices via standardized semantic interfaces. Standardized semantic interfaces provide a common way to access the devices' data from the application level. For the different use-cases covered by the project, there is a need for seamless communication across diverse protocols, such as OPC UA, MQTT, and DDS. Ensuring interoperability at the protocol level is essential to make use of these interconnected systems.

To overcome the challenges of multi-protocol device communication and enable interoperability at the protocol layer, we envision using a middleware solution that unifies the messages across different protocols as shown in Figure The messages from different protocols being unified at the middleware layer allow the dataflow vertically and horizontally and, also, enable a unified access to the data from the application layer.

[21] https://www.eclipse.org/tahu/spec/sparkplug_spec.pdf
[22] https://protobuf.dev/

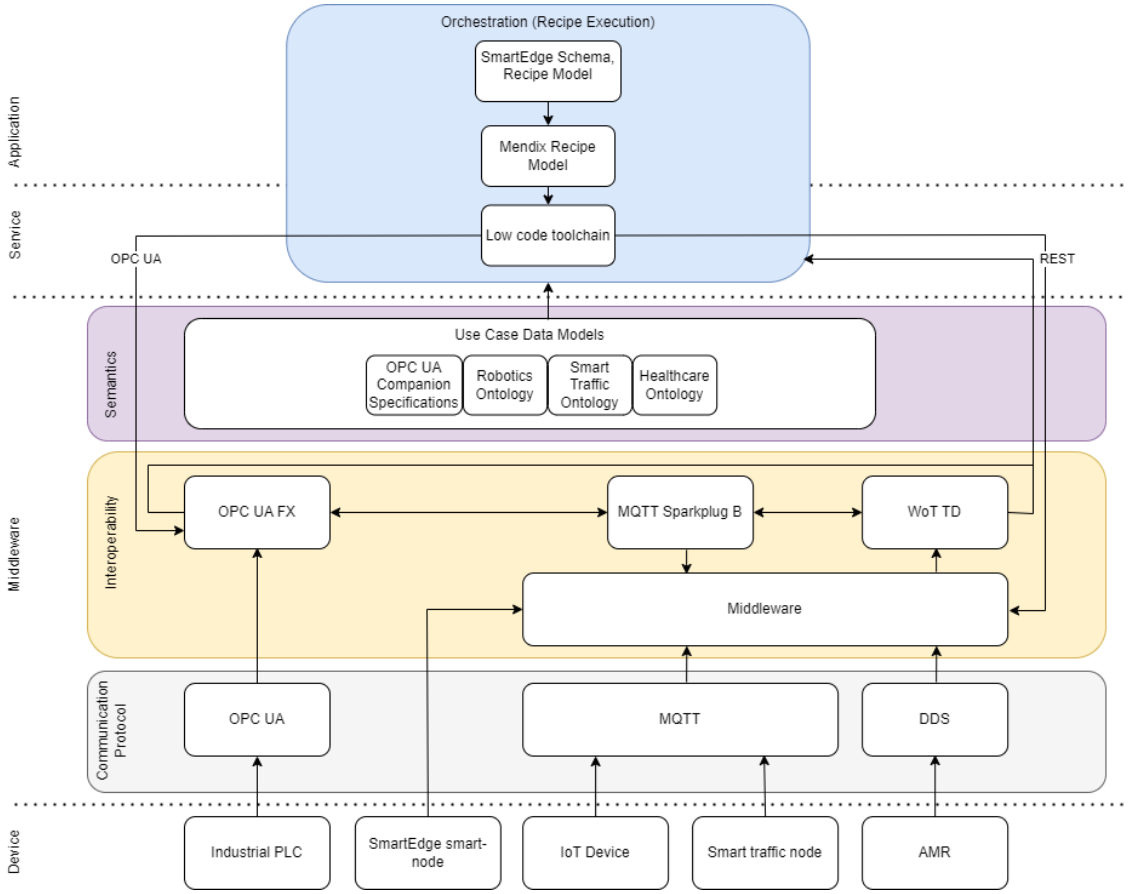

Figure 3.16: Standardized Semantic Interfaces in SmartEdge

Middleware is seen as a mediation layer for the devices, which communicate MQTT, DDS, or middleware-specific protocol. On top, Web of Things Thing Description (WoT TD) will be utilized as a machine-readable standardized metadata description of the devices, their data and services. WoT TD provides detailed information about the capabilities, properties, and interactions that a device offers.

OPC UA-enabled devices make use of information modelling features of OPC UA. SmartEdge relies on OPC UA Field eXchange (OPC UA FX) specifications, which target field device interoperability in the industrial automation domain. OPC UA FX provides a common field component model (information and interfaces) alongside an information exchange model.

In order to achieve the interoperability between the OPC UA-enabled devices and the middleware, OPC UA FX messages can be mapped to MQTT Sparkplug B specification. With that the OPC UA-enabled devices will be integrated at the middleware layer to the rest of the system.

For the semantic description of the data, SmartEdge will make use of domain-specific ontologies, as well as of the OPC UA companion specifications, which define specific data types within the OPC UA information model. The companion specifications are typically developed for particular domains, devices, or applications to ensure interoperability and data consistency.

The use case data models, as well as the WoT TDs and OPC UA information model, describing the devices, will be the basic building blocks for defining devices' capabilities and skills, used to create the SmartEdge recipes. Generic representation of devices' skills and capabilities will be fed into the low-code toolchain, where the recipes will be created and executed. Low-code runtime then executes the recipe by interacting with the middleware.

Table 3.12: Technologies applied in SmartEdge Various Use Cases

UC	Communication protocol	Serialization format	Semantics
1	W3C WoT (different protocols)	RDF Turtle, JSON/JSON-LD	(Use-case specific ontologies)
2	C-V2X/ETSI-G5, MQTT/NATS	JSON/JSON-LD	Smart Traffic Ontology
3	DDS, Zenoh	ROS2 Binary, JSON-LD	Robotics Ontology
4	OPC UA	XML	OPC UA FX and OPC UA companion specifications
5	BLE, DDS, MQTT	JSON	Healthcare Ontology

Table 3.12 summarizes various technologies used at different levels in SmartEdge use cases.

In Use Case 1, W3C WoT and Web RTC will be employed at the communication protocol level. Serialization will be done using RDF Turtle and JSON/JSON-LD. Despite the absence of a specific information model for semantics, messages between vehicles are expected to convey the intention of vehicles and their states, defined in a domain-specific ontology.

For Use Case 2, the term Smart Traffic Node refers to any of UC2 smart nodes such as Connected Vehicle, Sensor Node, or Controller Node. C-V2X and/or ETSI-G5 communication protocols are utilized for V2V and V2I communication among the vehicles and the fixed infrastructure nodes, while NATS publish/subscribe handles data collection and service API communication. UC2 uses NATS only internally, while a “NATS to MQTT interface” handles compatibility with SmartEdge middleware so that Smart Traffic Nodes can fully communicate to the middleware via MQTT standard. Radar detection of vehicles will be serialized in a vendor-specific binary format, while V2X message types will use standard binary formats like SPAT, CAM, CPM, MAP. Sensor Node and Controller Node convert binary V2X and radar messages into semantic formats so that the data from these nodes can be serialized in JSON/JSON-LD.

Use Case 3 will employ DDS for intra-robot communications and Zenoh as middleware for inter-robot and edge device communications. ROS2 will use a binary format for communication over DDS directly between DDS nodes. Whilst there is no official ROS2 message format, several hundred of the most common messages are available online[23]. Messages flowing between robots and edge devices over the Zenoh are SmartEdge or application specific messages and are formatted in JSON-LD. With that, each node specifies the message types it supports, but there is no formal ontology specification. The control and coordination messages passed over Zenoh between robots and other edge devices as either SmartEdge messages or application specific messages.

In Use Case 4, OPC UA will serve as the communication protocol for machine-to-machine communication, with OPC UA FX enabling interoperability between components and describing devices' capabilities. Additional domain specific OPC UA companion specifications will be utilized to describe the data at the semantic level.

For Use Case 5, BLE, DDS, or MQTT will be used for communication. Data will be serialized in JSON and possibly in OMG IDL binary format. The healthcare ontology will be employed for semantics. Messages will follow a generic configuration with specific sub-models/templates

[23] https://github.com/ros2/common_interfaces

based on the situation (e.g., discovery, registration, negotiation, data-streaming). The UC5 knowledge graph will provide details on dynamics and data representation.

4 DataOps tool for semantic management of things and embedded AI apps

This chapter reports the design of the SmartEdge DataOps toolbox supporting the continuous integration of things and applications through the standardized semantic interfaces discussed in Chapter 3. The objective of the DataOps toolbox is to provide a set of technologies to perform data integration from different sources with a specific focus on performance and scalability requirements for both cloud and edge environments. Moreover, in line with the objectives of the SmartEdge solution the design of the DataOps toolbox focuses on low-code approaches to facilitate the configuration and reusability across different use cases.

The content of the chapter is structured as follows:

- Section 4.1 discusses the relevant requirements for the design of the DataOps toolbox considering inputs from SmartEdge deliverable D2.1;
- Section 4.2 provides an overview of the relevant state-of-the-art focusing on data interoperability solutions enabled by a declarative and low-code approach;
- Section 4.3 describes the proposed design for the DataOps toolbox in terms of components identified and relevant technologies for its implementation in SmartEdge according to the defined requirements;
- Section 4.3 outlines a preliminary analysis on how the designed DataOps toolbox can support the SmartEdge use cases.

4.1 Requirements for the DataOps Toolbox

This section discusses the requirements for the DataOps Toolbox considering the ones identified in D2.1 for the Continuous Semantic Integration solution and the analysis of the SmartEdge use cases. The requirements, reported in Section 2, are associated with a need for enabling data interoperability among different nodes composing a swarm.

4.1.1 Data Interoperability

The issue of data interoperability is a significant concern when operating within a multi-stakeholder ecosystem comprising diverse actors [Sadeghi20]. Similarly, within a swarm there is a need for data interoperability among diverse nodes that employ heterogeneous data formats, specifications, and semantics. The ability to exchange data without any loss of meaning among communicating parties is an essential objective, but it is notoriously challenging to achieve also due to:

- heterogeneous information systems that communicate using different protocols and by exposing different interfaces (cf. requirements from D2.1 HP-001, HP-005, HP-006, HP-007, HP-008, HP-009, HP-011, HP-014, LC-009, CSI-001).
- heterogeneous data formats with varying semantic interpretations employed by multiple actors/nodes in the same domain (cf. requirements from D2.1 LC-015, CSI-002, CSI-008, CSI-013, CSI-014). This phenomenon may arise due to several factors that make the establishment of standards difficult, for example, the persistence of legacy applications or the usage of proprietary data formats.

While a message conversion process can offer a valid solution to define transformations across data formats, the integration of such processes considering different data sources and sinks is

something that requires a case-by-case analysis and cannot be solved by a single solution. As an example, to achieve interoperability between a *Device A* that outputs JSON data and a second *Device B* that consumes CSV data it is not enough to be able to convert a JSON payload into a CSV payload. *Device A* may only be capable of transmitting data using a *Remote Procedure Call* protocol while *Device B* may only be capable of receiving data through a *HTTP* API.

To achieve data interoperability, we thus need to address two main challenges reflecting the first two interoperability layers in the European Interoperability Framework (EIF) Toolbox[24]:

- *heterogeneous interface integration* to guarantee ***technical interoperability*** (i.e., the possibility of a data exchange between two systems) through **standardized interfaces**, and
- *payload conversion* to guarantee ***semantic interoperability*** (i.e., ensure that the target node can understand the message received and act appropriately) through **standardized semantics**.

The DataOps toolbox has the objective of enabling data interoperability between heterogeneous structured data sources by leveraging the standardized semantic interfaces discussed in Section 3.

To identify more specific requirements for the DataOps tool, we analyse the types of data exchanges that can be implemented within a SmartEdge swarm between different types of nodes as discussed in deliverable D2.1. Figure 4.1 provides a diagram representing the different cases identified.

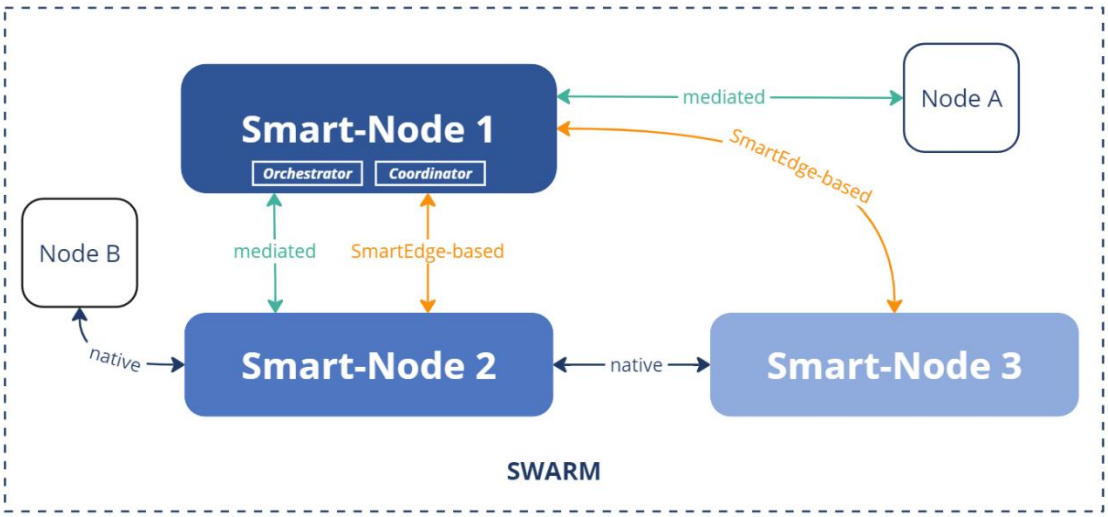

Figure 4.1: Types of data exchanges within a swarm

The simplest case is related to *SmartEdge-based data exchanges* between smart-nodes, i.e., nodes that execute a SmartEdge-enabled component. In this case, the nodes are directly capable of processing and exchanging the data according to the standardized semantic interfaces defined by SmartEdge. The communications between the coordinator/orchestrator of the swarm and a smart-node can be an example of *SmartEdge-based data exchanges*.

As a second case, we can identify *native data exchanges* between different types of nodes, i.e., also considering brownfield devices. In this case, we assume that two nodes in the swarm are

[24] https://joinup.ec.europa.eu/collection/nifo-national-interoperability-framework-observatory/solution/eif-toolbox/6-interoperability-layers

already capable of exchanging data according to common interfaces and semantics and their communication should only be configured. As an example, we can consider a set of cameras emitting video-streams and a smart-node capable of processing such streams. In this case, once the data exchanges between the cameras in a certain swarm and the smart-node are configured (e.g., by providing IP addresses of the cameras to the smart-node), then the data are exchanged by the nodes at runtime without requiring any intermediary component. A *native data exchange* may also exist between nodes that are already integrated to offer certain functionalities in a swarm, e.g., different ROS nodes or an in-cloud backend processing binary messages from IoT devices.

The third case, i.e., the *mediated data exchange* represents the target of the DataOps toolbox and should be enabled for the continuous semantic integration of the nodes involved. In this case, two nodes exchange data according to different interfaces and semantics and their communication should be enabled by SmartEdge. As an example, we can consider the exchange of data from a brownfield device that is not directly supported by a smart-node. For example, *Node A* may deliver messages using MQTT and a custom payload format while the *Smart-Node 1* may have as interface a HTTP API accepting JSON payloads according to a specific schema. In this case, the DataOps toolbox should enable both technical and semantic interoperability to enable the data exchange among the two nodes.

It is important to highlight that the same pair of nodes may require different types of data exchanges, e.g., the *Smart-Node 2* may be capable of exchanging information related to swarm coordination with the *Smart-Node 1* through a *SmartEdge-based data exchange,* while requiring a *mediated data exchange* for exchanging structured information received from a brownfield device (*Node B*).

To generalise, the DataOps toolbox should provide a set of technologies for the implementation of *mediated data exchanges* in different scenarios. The associated requirements are:

- the retrieval of data from heterogeneous data sources with different interfaces (e.g., protocols and interaction mechanisms),
- the conversion of payloads from one data format and/or data model to another one,
- the forwarding of the converted data to the target interface.

In this context, it is important to clarify the distinction between the processing of structured and unstructured information. The processing of unstructured information to extract structured data, e.g., the processing of a video stream by a machine-learning algorithm for object detection and identification, is out of scope of the DataOps toolbox and can be considered as an application executed by a certain node that interacts with the unstructured data stream (e.g., the video stream) and generates structured information (e.g., a JSON message describing the list of objects detected in the video). Considering the provided example, the DataOps toolbox can instead support the processing of the structured data generated by the node to guarantee data interoperability (e.g., convert the JSON message adopting a common vocabulary for the objects detected). For this reason, the fulfilment of requirements CSI-013 and CSI-014 will be supported by the DataOps toolbox, if structured data exchanges should be mediated, and/or by the solutions developed within WP5.

4.1.2 Performance and Scalability

For the definition of more non-functional requirements, it is important to consider the following categorization that can be applied to the described data exchanges:

- *Static* data exchange: exchange of pre-defined datasets with low volatility such as metadata (e.g., self-descriptions) and configurations for a node.
- *Runtime* or *on-the-fly* data exchange: exchange of messages and/or data streams at runtime between nodes.

The two data exchanges are associated with different characteristics and, consequently, different needs in terms of performance and scalability [Scrocca21]. On one hand, *static data exchanges* often require the conversion of datasets of larger size with low frequency, thus requiring the minimisation of resource usage and scalability in terms of the size of the input. On the other hand, *runtime data exchanges* are usually associated with small-size messages with high frequency, thus requiring the minimisation of the latency introduced by the conversion process and scalability in terms of concurrent conversion requests.

The KPIs 2.2 and 2.3 reported in Section 2 target the expected improvements in terms of performance and scalability to be demonstrated in SmartEdge considering the baseline results discussed by Scrocca et al. in [Scrocca21] terms of conversion time (140 *ms* conversion time with 50 *KBytes* XML payloads) and number of concurrent requests handled per second (100 *requests/s* with 50 *KBytes* XML payload on commodity hardware and considering a single converter instance).

Finally, in the context of SmartEdge an additional constraint is introduced by the types of nodes involved in the considered use cases. The resources available for each node, mainly CPUs and RAM, should be taken into account for certain edge devices with minimum specifications.

4.1.3 Deployment

The term *continuous,* associated with the semantic integration process, embraces two aspects that will be supported by the DataOps toolbox: (i) the discussed enablement of data interoperability among nodes with different integration requirements, and (ii) the possibility for the data interoperability solution to support deployments either in the cloud or on the edge.

Indeed, to facilitate a seamless integration between the edge and the cloud, the DataOps toolbox should support different deployment possibilities considering different types of devices (e.g., hardware and operating system) as well as cloud environments such as container orchestrators (cf. requirement from D2.1 HP-018). To facilitate reusability and portability of data interoperability solutions developed through the SmartEdge DataOps toolbox, the definition of the solution should be as much as possible decoupled from the necessary configuration for its deployment. Ideally, it should be possible to reuse with minimal effort the same solution in different deployment environments with minimum modifications required.

4.1.4 Low-code

The design of the DataOps toolbox should also aim at investigating low-code approaches to facilitate the configuration of mediated data exchanges by minimising the amount of code to be written and increasing the reusability of already defined solutions (cf. requirement from D2.1 LC-010).

4.2 STATE OF THE ART

This section provides an overview of the relevant state-of-the-art considered for the design of the DataOps toolbox. In particular, we focus on approaches exploiting declarative mappings to enable semantic interoperability, and frameworks for data integration that could be used to implement solutions for technical interoperability.

4.2.1 Semantic Interoperability through Declarative Mappings

Considering different nodes with a semantic harmonisation need, a naïve approach consists in the implementation of a point-to-point solution to enable a direct payload conversion between each pair of nodes adopting a different data standard (any-to-any approach). However, this approach is not scalable and the amount of mappings to be defined increases as $2n(n-1)$ with n being the number of different standards to be considered. When integrating different software systems that act in the same domain, or more generally share a common set of concepts and vocabulary, it is possible to employ a more effective any-to-one centralized mapping approach [Vetere05]. This approach reduces the number of mappings required for data interoperability enabling a better scalability of the solution. If there are n different formats, the number of mappings increases instead as $2n$ as shown in Figure 4.2. The assumption behind this approach is that it is possible to identify a reference conceptual model O, that models the common semantics of the data standards considered. Each data standard should be only mapped *to* and *from* the reference conceptual model. Using an ontology as a reference conceptual model, we can offer a valid solution to model the common semantics, as we do in SmartEdge, and has the additional advantage of enabling the creation of an interoperable knowledge graph during the conversion process between two standards [Scrocca20]. The advantages and challenges of such an approach in the context of the Web of Things is also discussed in a position paper by Bennara et al. [Bennara20].

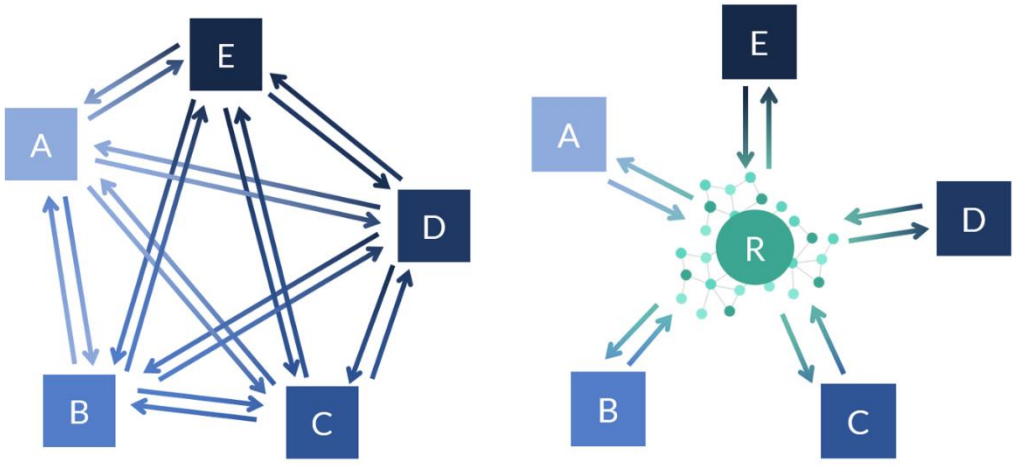

Figure 4.2: Comparison of the any-to-any and any-to-one approaches for interoperability.

Each of the arrows represented in Figure 4.2 is associated with the definition of a set of rules, usually called *mapping rules*, that enable the conversion of payloads from one data standard to another one.

The *payload conversion* can be implemented in different ways. From a literature review [VanAssche22] the following categorization emerges:

- *Hard-coded procedures*: the data conversion process is defined through code in a specific programming language. In this case, data conversion may take place via one-off scripts or by hard-coding specific conversion rules, known ahead of time, inside of another program. An example is a custom Python script that parses a specific JSON stream and generates an associated set of RDF triples. The problem with this approach is that the data conversion solution becomes difficult to maintain and difficult to reuse.
- *Format-specific mappings*: the data conversion process is defined by leveraging a well-defined mapping language to define conversion rules from a specific data format to a target one. An example is the TARQL[25] tool for converting CSV files to RDF, using an extension of the SPARQL language to define the mapping rules, or the R2RML W3C recommendation [Das12] for the mapping from relational dabatases to RDF. The downside of format-specific mappings is that they are optimized for a specific data format. Converting to and from a variety of formats would require using and maintaining a set of different tools, one for each format.
- *Declarative mappings*: overcome the shortcomings of the previously presented approaches by being implemented in a declarative mapping language. Mapping rules are decoupled from the mapping executor and different processors may be used to execute the mappings if they conform to the same adopted declarative mapping language. Additionally, they are not constrained to a specific data format but can convert data to and from various formats. A single solution for the definition and execution of the mappings should be learned and maintained.

Considering an ontology as a reference conceptual model, different declarative mapping languages for the conversion of heterogeneous data sources to RDF have been proposed [VanAssche22]. These declarative mapping languages can be classified as:

- dedicated languages based on R2RML [Das12] syntax (RML [Dimou14], D2RML [Chortaras18], KR2RML [Slepicka15], R2RML-F [Debruyne16], xR2RML [Michel15]),
- dedicated languages with custom syntax (Helio Mapping Language [Cimmino22], D-REPR [Vu19]),
- repurposed languages based on constraint languages (ShExML [García-González18]), extending the ShEx syntax),
- repurposed languages base on SPARQL [26] syntax: XSPARQL[Akhtar08], Facade-X, SPARQL-Generate[Lefrançois16].

Each mapping language provides at least one mapping processor able to execute the specific mappings.

The most widely used of these declarative mapping languages is the RDF Mapping Language (RML)[27]. RML extends R2RML, by adding support for heterogeneous data sources, such as files in the CSV, XML or JSON formats.

RML works by defining:

[25] https://tarql.github.io/
[26] https://www.w3.org/TR/sparql11-overview/
[27] https://rml.io/specs/rml/

- **Logical sources** can be considered as the input of the mapping process. They can reference relational databases or files in one of the supported formats. Depending on the type of file, they define how to access data inside of the file in different ways. For example, a logical source for a JSON file specifies how it accesses data with the JSONPath[28] language, while for an XML logical source XPath[29] is used.
- **Subject maps** which specify how the subject for an RDF triple is to be constructed from the source data. This may be either a constant or an expression that depends on the input source data.
- **Predicate-Object maps** are functions that create a predicate-object association for each item from a logical source. These predicate-object links are used in conjunction with the subjects generated from subject maps to compose a whole subject-predicate-object RDF triple.

In addition to these core concepts RML allows users to declare logical functions [Meester17] and join conditions. Logical functions are declared in RDF by defining their inputs and outputs. Their implementation depends on the RML processor that runs the mapping, but they should follow the contract of the logical function to assure predictable behaviour. Join conditions are instead necessary when the output of the mapping process depends on multiple input logical sources. The join conditions are used like in conventional relational database systems to link together data from different sources.

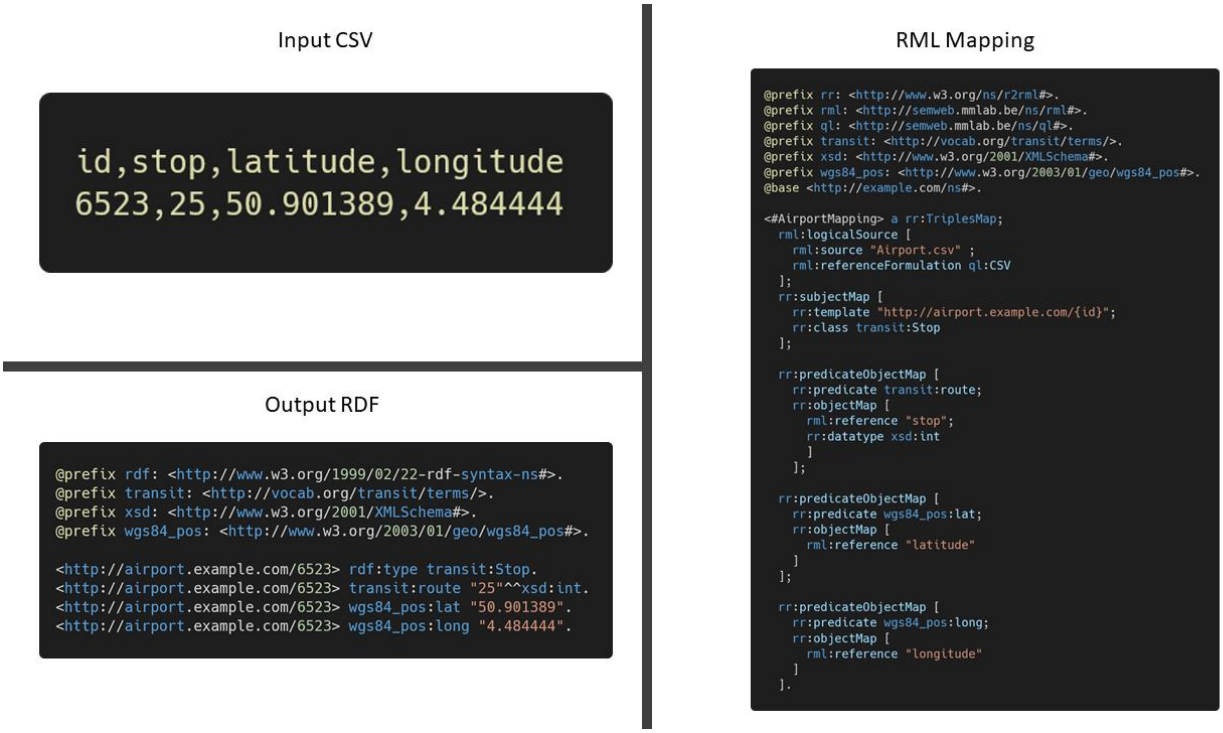

Figure 4.3: Example of an RML mapping from CSV to RDF

Figure 4.3 illustrates the transformation process of an input CSV file into the output RDF format through the application of an RML mapping. The depicted RML mapping constructs subjects for the RDF triples using a subject map that extracts information from the "id" column in the CSV file. As indicated in the subject map, each subject is defined to be of type (utilizing

[28] https://goessner.net/articles/JsonPath/
[29] https://www.w3.org/TR/xpath-3/

the *rdf:type* predicate*) transit:Stop*. Since there is only one row in the input CSV considered, a singular subject is derived. For each subject, the RML mapping defines three predicate-object maps. The first of these maps is used to convey information about the transit route to which the stop belongs. This is done by linking the previously obtained subject with the predicate *transit:route* to the object obtained from the input csv under the column "stop". Furthermore, the same is done for geographic coordinates of the stop using the *wgs84_pos:lat* and *wgs84_pos:long* predicates and getting the corresponding values ("latitude" and "longitude") from the CSV file.

RML is structured to be more machine-readable than human-readable. To increase its usability, YARRRML[30] was introduced. YARRRML is a syntax based on YAML and it offers a more user-friendly way to create RML mappings. It acts as an intermediary step, allowing users to define mappings in a human readable YAML syntax, which then generates the corresponding RML mappings, making the process more accessible.

In Figure 4.4 we compare the same mapping rules defined using the YARRRML language (on the left) and then translated to RML (on the right). We can see that YARRRML is significantly less verbose than the RML language. This increased conciseness without loss of expressiveness simplifies the maintenance of mappings and as such is generally preferred.

Figure 4.4: Example of a YARRRML file (left) and corresponding RML mapping (right)

Recently, the W3C Knowledge Graph Construction community group collected different experts to discuss the further development of RML and propose it as a W3C recommendation. As a result, a new specification based on a modular ontology has been released [Iglesias-Molina23].

RML is a stable solution with widespread adoption and many supported mapping processors exist but can only be used for mappings generating RDF triples as an output. While many solutions exist for the conversion via declarative mappings to RDF, the same is not true for the conversion from RDF to a target data standard [Grassi23]. A more generic approach to define mapping rules is that of using a template-based declarative approach. This is the case for the *mapping-template* component [Grassi23] which defines mappings using the Apache Velocity

[30] https://rml.io/yarrrml/

Template[31] language (VTL). Just like RML, this tool can retrieve input data from various common formats like CSV, XML, JSON, and RDF. It employs specific query languages for each format— for instance, SPARQL for RDF and JsonPath for JSON files. These queries retrieve data from input files and organize it into tabular structures known as *data frames* that are then used to generate the required output. The benefit of employing a template-based language is the flexibility it offers. Users are not restricted to a single output type but can potentially represent any plain text structure. On the downside, the syntax used to define the mappings is bound to the Velocity Template Language (VTL) and a single mapping processor is currently available[32].

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ex: <http://example.com/ns#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix transit: <http://vocab.org/transit/terms/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@base <http://example.com/ns#>.

<#TransportMapping> a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Transport.xml" ;
    rml:iterator "/transport/bus";
    rml:referenceFormulation ql:XPath;
  ];

  rr:subjectMap [
    rr:template "http://trans.example.com/{@id}";
    rr:class transit:Stop
  ];

  rr:predicateObjectMap [
    rr:predicate transit:stop;
    rr:objectMap [
      rml:reference "route/stop/@id";
      rr:datatype xsd:int
    ]
  ];

  rr:predicateObjectMap [
    rr:predicate rdfs:label;
    rr:objectMap [
      rml:reference "route/stop"
    ]
  ].
```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix transit: <http://vocab.org/transit/terms/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ex: <http://trans.example.com/>.

#set( $query = '
    for $stop in /transport/bus/route//stop
    return map {
        "stopId": $stop/@id,
        "stopName": $stop/text(),
        "busId": $stop/ancestor::bus/@id
    }')
#set( $data = $reader.getDataframe($query))

#foreach($stop in $data)
ex:$stop.busId rdf:type transit:stop ;
  transit:stop "$stop.stopId"^^xsd:int ;
  rdfs:label "$stop.stopName" .
#end
```

Figure 4.5: An RML mapping (left) and the same mapping expressed in the VTL template language (right)

Figure 4.5 shows how an RML mapping can be written using the *mapping-template* component approach. Like in RML, data is extracted from the input XML file shown in Figure 4-6. In the *mapping-template* case, a single query written with XQuery[33] is defined to access the input once while, in RML multiple XPath queries are defined. The *mapping-template* approach differs from RML by explicitly saving the extracted data in a support data frame data structure that facilitates the implementation of custom optimisation for accessing and reading the data to be converted. The direct key-based access to values in the *data frame* provided by VTL is then used to write the expected output, i.e., the RDF triples. This output is shown in Figure 4.6.

[31] https://velocity.apache.org/
[32] https://github.com/cefriel/mapping-template
[33] https://www.w3.org/TR/xquery-31/

Figure 4.6: Input XML file and output RDF (Turtle) file for the mappings in Figure 4.5. Note that both mappings produce the same output.

4.2.2 Technical interoperability through Data Integration Tools

The issue of data integration is ubiquitous and frequent. Therefore, several tools and frameworks to address it are available both as open-source or as commercial solutions. Most commonly, tools follow the Extract-Transform-Load (ETL) approach to data integration where the process is defined by three main steps: data is gathered from various sources (Extract), transformed or modified to fit the desired target or system (Transform), and finally loaded into a database, data warehouse, or another destination (Load). Examples of such systems include Talend[34], Apache Fink[35] or Apache Kafka[36].

However, the integration of heterogeneous systems does not require only ETL processes. Other functionalities, such as data filtering, merging or routing, are usually involved. The main categorisation of the components and techniques that can be used in an integration process is defined by the Enterprise Integration Patterns [Hohpe04]. An example framework that is built with these patterns at its core is Apache Camel. Camel is an open-source, configurable and extensible Java integration framework to facilitate the integration with various systems consuming or producing data.

ETL tools can be used to implement the any-to-one mapping approach discussed and provide interoperability among different systems. ETL tools usually support low-code approaches to configure data integration solutions using dedicated declarative languages (e.g., a Domain Specific Language) and/or graphical interfaces.

To support data integration leveraging a global conceptual model in the form of an ontology, the ETL tools should have support for technologies from the Semantic Web field. UnifiedViews [Knap14] and LinkedPipes [Klímek16] have been implemented during the years, providing environments fully based on Semantic Web principles to feed and curate RDF knowledge bases. A different approach is used by Talend4SW[37], whose aim is to complement an already existing tool (Talend) with the components required to interact with RDF data. The Chimera framework [Grassi23] provides additional components for the Apache Camel framework to implement data transformation pipelines through Semantic Web technologies.

[34] https://www.talend.com/
[35] https://flink.apache.org/
[36] https://kafka.apache.org/
[37] http://fbelleau.github.io/talend4sw/

4.3 DESIGN OF THE DATAOPS TOOLBOX

The implementation of mediated data exchanges within a swarm will be supported by the DataOps toolbox designed to facilitate the development of low-code solutions for data interoperability.

In this section, we specify the components identified for the DataOps toolbox and the technologies that will be leveraged for the implementation of such components according to the requirements discussed in Section 4.1.

4.3.1 Components of the DataOps Toolbox

The challenge of data interoperability cannot be universally defined for all scenarios, and as such, there cannot exist a single solution. The DataOps toolbox should provide a flexible and extensible set of components that is adaptable to the requirements of integrating heterogeneous information systems. For this reason, the main design principle that we follow is related to the modularity of the solution. The DataOps toolbox is designed as a set of composable modules that can be appropriately configured and combined within a pipeline to address heterogeneous integration requirements.

4.3.1.1 DataOps Pipelines

To tackle the discussed challenges, starting from the state-of-the-art analysis, we define a set of conceptual steps that are required for the definition of pipelines via the DataOps toolbox.

We consider a mapping scenario where data from a data source, represented according to an input data format and data model (Standard A), should be converted to an output data format and output data model (Standard B) and stored in a data sink. The mapping scenario may involve the integration of additional data sources for the generation of the output, and data transformations to be applied during the process.

The adoption of an any-to-one mapping approach with a reference ontology as global conceptual model represents a solution for the core transformation required by the presented mapping scenario. Such transformation should be supported by a *semantic conversion* process. Such process, shown in Figure 4.7, can be represented as a two-step approach:

- *Lifting* step: the information contained in the input data is extracted according to the reference ontology;
- *Lowering* step: the information is accessed relying on the reference ontology and used to build the output message.

As suggested in the literature [VanAssche22], the *semantic conversion* process should rely on declarative mapping languages to foster the maintainability and scalability of the solution. Mapping rules should be provided as a separate input to the *lifting* and *lowering* steps and a mapping processor should be able to interpret and execute the mappings.

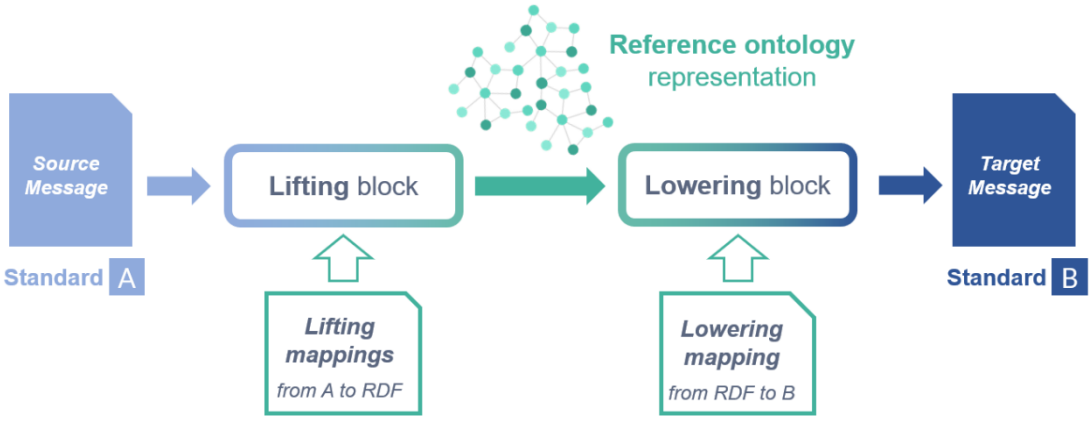

Figure 4.7: Semantic conversion process

Leveraging the standardised semantics identified by SmartEdge, i.e., a (set of) reference ontology(ies) encoding the common semantics for each use case, it is possible to implement semantic conversion processes to support semantic interoperability among different nodes within a swarm.

However, to implement the mapping scenario discussed, a DataOps pipeline also requires the capabilities to obtain the data from the input data source and forward the output to the target data source. We define the **DataOps pipelines** as the composition of different components supporting a mediated data exchange between two nodes in the swarm. A minimum and generic pipeline is shown in Figure 4.8 and is composed by: (i) an input *node data connector* configured to access the source node, (ii) a *mapping processor* configured with lifting mappings, (ii) a *mapping processor* configured with lowering mappings, (iv) an *output node data connector* configured to forward the data the target node. Moreover, a *DataOps pipeline* can be enriched with additional processing blocks to fulfil integration requirements, e.g., interact with an external system to enrich the input data or forward the same input message to multiple recipients.

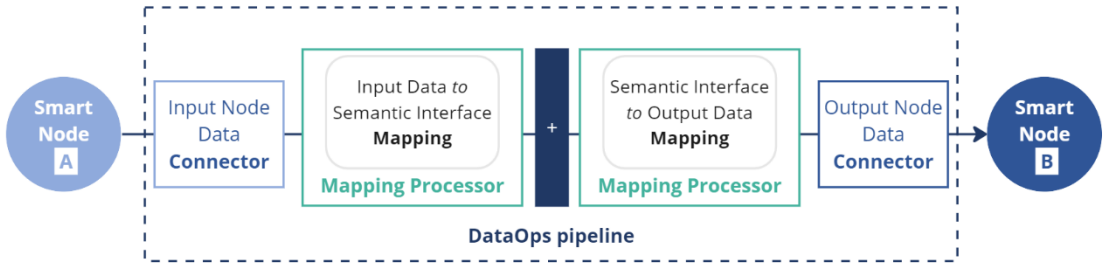

Figure 4.8: High-level representation of a DataOps pipeline

In some cases, the lowering step may not be needed, for example, a SmartEdge smart-node capable of processing RDF data (e.g., processing JSON-LD Web of Things descriptions) may only need the execution of lifting mappings. Similarly, data exchanges meant to enrich the knowledge graph in a SmartEdge swarm only need a lifting procedure to generate RDF triples.

The composition of DataOps pipelines for a specific mediated data exchange should be supported by a declarative approach allowing for the selection and configuration of the required components. The same DataOps pipeline should support different deployment options to address specific constraints.

In the following sections, we specify better the main blocks of a DataOps pipeline, i.e., *node data connectors* and *mapping processors*. In particular, we expand and analyse the aspects that should be possible to declaratively specify via a *mapping language* and the corresponding functionalities that should be supported by the components of the Data Ops pipeline.

4.3.1.2 Node Data Connectors

A *node data connector* component should provide data access from sources and data forwarding to sinks considering the different types of nodes enabling the SmartEdge use cases. Figure 4.9 provides an overview of how a declarative mapping language can support its configuration and what are the functionalities that it should implement.

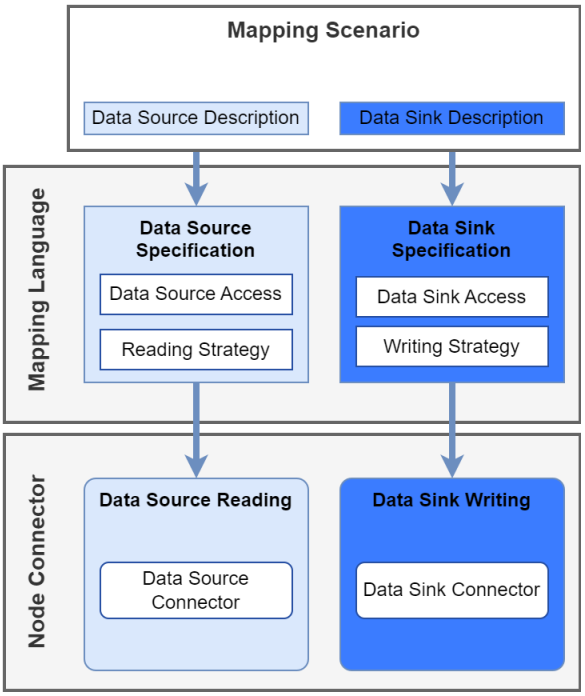

Figure 4.9: Node data connectors overview

Node data connectors should support a *Data Source Specification* that declaratively defines how to access (*Data Source Access*) and retrieve (*Reading Strategy*) the data to be processed by the subsequent blocks in a DataOps pipeline. Different configurations may be needed according to the data source considered, for example considering if it is local or remote, if it is a dataset or a data service. The *Data Source Access* should indicate the location (e.g., URL) to access the data source, the protocol to access the resource, and the security mechanisms restricting the access. The *Reading Strategy* should indicate the type of interaction expected by the data source, e.g., push versus pull mechanism, synchronous versus asynchronous, batch versus stream. Different *Node Data Connector*(s) support different types of data source(s) and the expected interaction in reading data from them, e.g., a *Node Data Connector* may support reading data from data sources adopting a specific protocol.

Similarly, a *Node Data Connector* is needed also to support the writing of data to a target data sink. When writing data, a *Data Sink Specification* declaratively defines how to connect (*Data Sink Access*) and send (*Writing Strategy*) the data obtained as a result of the mapping process. Finally, the result of the mapping process may be split considering different data sinks. The

implementation of this functionality requires the selection or implementation of *Node Data Connector*(s) supporting the target data source and the expected interaction in writing data.

4.3.1.3 Mapping Processors

A *mapping processor* should support the execution of mapping rules defined according to a declarative mapping language. Mapping processors are decoupled from the mapping languages supported and may implement different approaches for the performant and scalable execution of the mapping rules. Starting from an analysis of mapping processors for RDF Knowledge Graph Construction available in the literature, we define the functionalities that characterise a *declarative mapping language* and a corresponding *mapping processor* (shown in Figure 4.10).

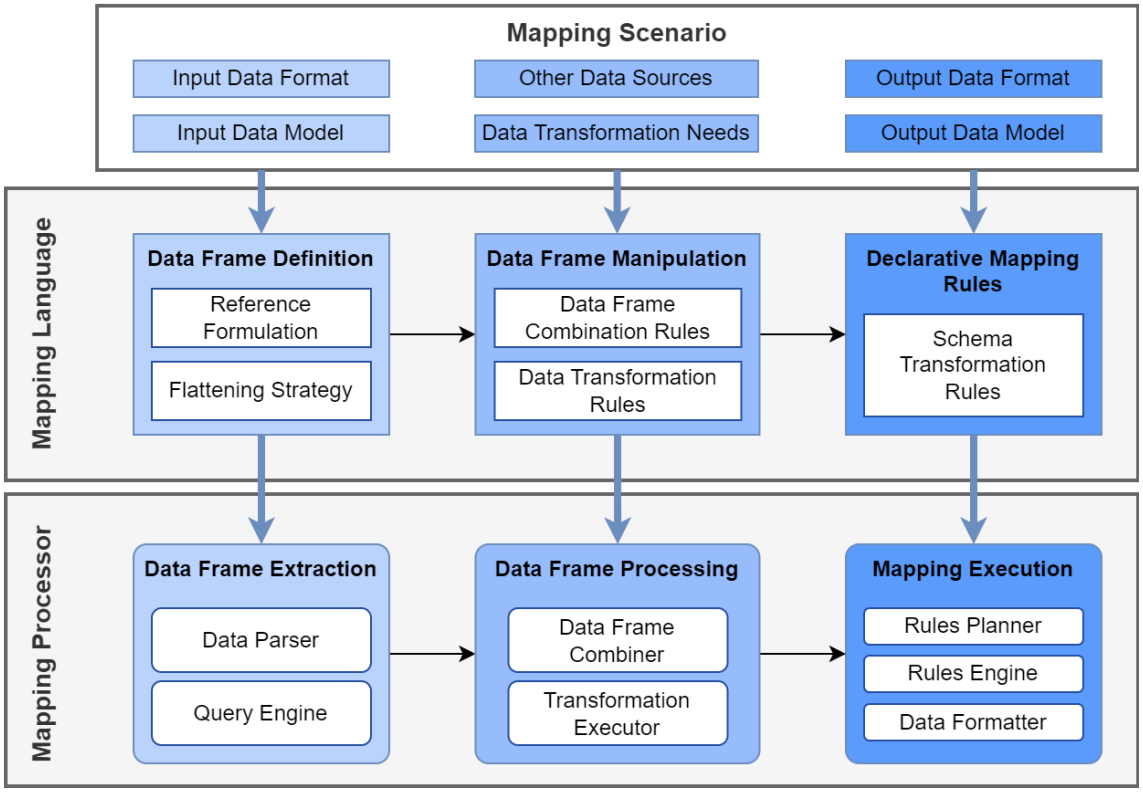

Figure 4.10: Overview of a mapping processor

Mapping processors should be able to access data and internally store them to support the application of data transformation operations according to the considered mapping rules. To decouple the parsing and extraction of data from heterogeneous data sources from the execution of mapping rules, intermediate data structures are usually adopted. Such data structures can be generalised considering the concept of *data frame*, i.e., a two-dimensional data structure made of rows and columns.

The parsing and extraction of data is declaratively defined by a *Data Frame Definition* that specifies: (i) the *Input Data Format* (CSV, XML, etc.) and the corresponding *Reference Formulation* (e.g., SQL, XQuery, etc.) to parse and extract data needed in the data frame and (ii) a proper *Flattening Strategy* for the definition and extraction of a data frame also in the case of hierarchical data sources, e.g., JSON or XML.

The *Data Frame Extraction* functionality should be implemented by a *mapping processor* relying on the *Data Frame Definition* specified. The implementation of this functionality requires the

selection of a *Data Parser* responsible for parsing data received from the data source according to their specific format (e.g., CSV/XML/JSON), and a *Query Engine* capable of extracting the data frame from the parsed data and according to the *Flattening Strategy*. The *Data Parser* and *Query Engine* are usually implemented by the same component. This component can be a SQL query engine in the case of a relational database, a SPARQL query engine in the case of RDF data, or a more generic library extracting a data frame from a JSON object.

The required *data transformations* are specified declaratively via rules that can be interpreted by the mapping processor as operations applied to the data frames (*Data Frame Manipulation)* to (i) combine multiple data frames, or (ii) transform the values contained in a data frame. The capability of applying these transformations is referred to as the *Data Frame Processing* functionality. The implementation of this functionality requires the selection of two components: a *Data Frame Combiner* capable of executing the combination of one or more data frames according to the mapping rules specified, and a *Transformation Executor* capable of applying the data transformation (e.g., a function to modify as lowercase the values in a certain column).

Finally, the required *schema transformations* rules are defined by declarative mapping rules that specify how the data in the data frame should be combined to obtain a valid target output. The execution of such rules (*Mapping Execution*) is implemented by three components: a *Rules Planner* evaluating the dependencies among different mapping rules to schedule their possibly concurrent execution, a *Rules Engine* actually executing the rules, and a *Data Formatter,* validating and formatting the generated output.

Performance and scalability requirements can be met by either selecting a more performant mapping processor for the specific use case or by optimising the mappings (e.g., considering the amount of data frames extracted).

4.3.2 Technologies for the DataOps Toolbox

This section discusses how the requirements defined in Section 4.1 can be addressed by the designed DataOps toolbox. A scouting of available technologies was performed and we present the outcomes that will guide the implementation of the DataOps toolbox in SmartEdge.

4.3.2.1 Data Interoperability through Chimera

For the implementation of DataOps pipelines, we propose the usage and potential extension of the Chimera[38] framework. Chimera [Grassi23] is an open-source solution based on Apache Camel and enables the definition of semantic data transformation pipelines by composing different components for dealing with knowledge graphs. By leveraging the Apache Camel framework, we can employ already existing production-ready solutions to address issues associated with the integration of heterogeneous nodes (i.e., technical interoperability). Moreover, the Chimera components can provide the necessary functionalities for implementing a semantic conversion process. Indeed, Chimera can support the implementation of both node data connectors and mapping processors as outlined in previous sections.

Apache Camel is a robust and stable open-source project that supports out-of-the-box several components, runtimes and formats to access and integrate a large set of existing systems and

[38] https://github.com/cefriel/chimera

environments. Such components can provide the necessary functionalities to implement node data connectors in SmartEdge. We report here a preliminary list of components that may support the requirements of the SmartEdge use cases (e.g., communication protocols):

- the *File Component* to read and write to files on the local filesystem;
- the *HTTP Component* to access or invoke external http resources;
- the *Camel Paho MQTT 5 Component*[39] for MQTT;
- the *Camel NATS Component*[40] for NATS;
- the *OPC-UA Component*[41] for the OPC-UA protocol.

Apache Camel relies on the basic concept of *Route* defining a certain logic to load, extract, integrate, transform, and output data. Each *Route* is a pipeline composed of a set of components that are applied in a specific sequence to a certain *Exchange*, i.e., an entity going through a *Route*. The *Exchange* is identified by an identifier, and it can be thought of as an envelope. It contains the messages (e.g., the data being processed) but also a set of properties that can be used to carry an additional state during the *Route* execution. The components specified within the *Route* play a role in manipulating the data contained in the *Exchange*. These components provide a range of capabilities. For instance, the *File Component* can be employed to generate, duplicate, or remove files. These functions and their settings are configured through a component's Uniform Resource Identifier (URI). A *URI* is a string representation that guides the component in performing operations, abstracting the need for users to interact with the underlying code. By exposing the capabilities of a component through a URI string, users do not need to know the underlying code for a component but can simply choose which functionalities to use. Additionally, the decoupling between the configuration of a *Route* and the code simplifies the introduction of changes to the logic of a pipeline.

Figure 4.11 shows a minimal snippet of an Apache Camel pipeline, defined using the Java DSL, that transfers data from a folder to another one.

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

Figure 4.11: A Java DSL Camel route example that transfers files from the 'inputdir' to the 'outputdir' using the file component's URI arguments.

As shown in Figure 4.12, Chimera provides a set of additional components that can be combined within an Apache Camel pipeline. Chimera introduces the support for several operations on knowledge graphs by leveraging the abstractions and functionalities offered by the RDF4J[42] library. The RDF Graph in Chimera pipelines is an abstraction that can refer to a local knowledge graph (in-memory, filesystem), or a remote graph stored in a triplestore or accessible through a SPARQL endpoint.

[39] https://camel.apache.org/components/4.0.x/paho-mqtt5-component.html
[40] https://camel.apache.org/components/4.0.x/nats-component.html
[41] https://camel.apache.org/components/3.21.x/milo-client-component.html
[42] https://rdf4j.org/

It is important to highlight that additional components can be developed if specific requirements arise and they can be then re-used in different Apache Camel pipelines.

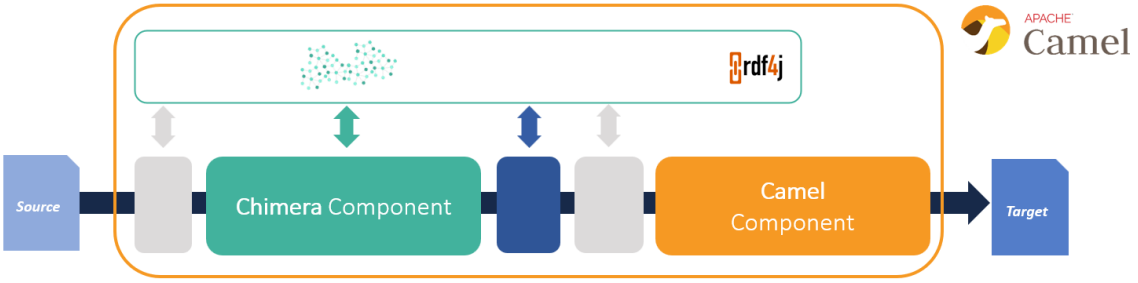

Figure 4.12: The Chimera framework provides a set of Apache Camel components that can be combined in integrated pipelines

Figure 4.13 provides an overview of the functionalities offered by the Chimera components for knowledge graph construction (lifting), transformation (enrichment via RDF graph merge and SPARQL CONSTRUCT, inference), validation (SHACL[43] shapes), and exploitation (lowering).

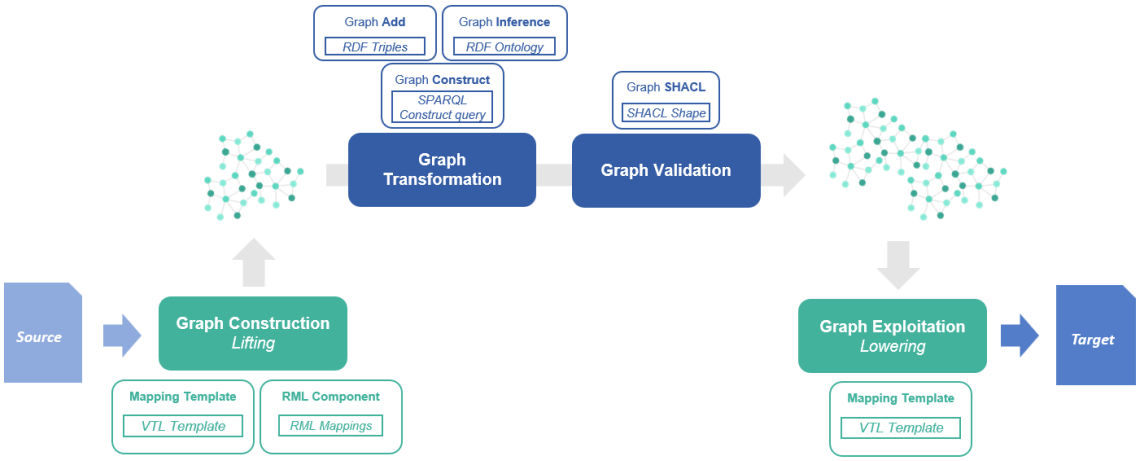

Figure 4.13: Overview of the functionalities implemented by Chimera

Considering the semantic conversion process, Chimera currently offers two mapping processors implementing the state-of-the-art approaches discussed in Section 4.2. It handles the RML declarative mapping language via a dedicated component that can support the implementation of the lifting step, and the template-based approach powered by Apache Velocity that supports the implementation of both lifting and lowering.

In SmartEdge, the configuration of DataOps pipeline can leverage the abstractions introduced by Apache Camel and should be supported via:

- the selection of appropriate Camel components that can serve as *node data connectors*;
- the implementation of appropriate mapping rules using a declarative mapping language and considering the mediated data exchanges needed by each SmartEdge use case;
- the selection of an appropriate *mapping processor* supported by Chimera to execute the mappings;
- the identification of additional components required to implement the pipeline.

[43] https://www.w3.org/TR/shacl/

This process may lead to the implementation of additional Apache Camel components (e.g., node data connectors for protocols currently not supported) and/or Chimera components (e.g., integration of additional mapping processors or adaptation of the existing ones to support performance and scalability requirements). Finally, we plan to work on Chimera to increase the overall TRL of the solution.

4.3.2.2 Performance and Scalability of Mapping Processors

Mapping processors are software components responsible for executing data and schema transformations defined by a declarative mapping language. For this reason, they have a great impact on the overall performance and scalability of a DataOps pipeline. In SmartEdge we will focus on the optimization of the semantic conversion process starting from the literature and the available mapping processors.

In this section, we mainly discuss a dedicated set of mapping processors tailored for the RML language. Indeed, given the broader adoption of RML, many solutions have been developed for this language proposing different approaches and optimisations for the mapping process.

There are various mapping processors available for the RML language, one is RMLMapper [44], a Java based RML processor maintained as the reference implementation of the mapping language by the same group that developed RML language. CARML[45] is an alternative mapping processor implemented in Java focusing on streaming and (potentially) non-blocking processing of mappings. CARML also defines a set of extensions to the RML language to better support stream data sources, XML namespaces and functions for data transformation. SDM-RDFizer[46][Iglesias20] is a Python based project which utilizes streamlined data structures and relational algebra operators to efficiently execute RML triple maps through a multi-thread safe procedure for each set of RML rules. Another Python based project is morph-kgc[47][Arenas-Guerrero22] that is based on the popular Python *pandas* [48] library. Morph-kgc improves performance by introducing the concept of *mapping partitions.* These partitions are composed of sets of mapping rules that generate distinct subsets of the resulting knowledge graph. By processing each group independently, this approach reduces the overall memory consumption and execution time needed for the conversion process. Finally, RocketRML [49] provides a JavaScript mapping processor for RML.

The performance and scalability of different mapping processors is evaluated in the literature ([Scrocca21] [Arenas-Guerrero21] [Arenas-Guerrero22]) but the best solution usually depends on the specific scenario considered, e.g., on different parameters characterising the mapping rules [Chaves-Fraga19]. Several benchmarks exist for evaluation, such as the GTFS Madrid Benchmark [Chaves-Fraga20], however, these are usually targeting the scalability of large-size datasets. To the best of our knowledge, no benchmarking is currently focusing on the performance and scalability of the mapping process over more dynamic scenarios with multiple concurrent requests and low-latency requirements.

[44] https://github.com/RMLio/rmlmapper-java
[45] https://github.com/carml/carml
[46] https://github.com/SDM-TIB/SDM-RDFizer
[47] https://github.com/morph-kgc/morph-kgc
[48] https://pandas.pydata.org/
[49] https://github.com/semantifyit/RocketRML

As these mapping processors are specific to the RML mapping language their usage is limited to situations where the desired output is RDF. The *mapping-template*[50] tool is a Java based solution implementing a more generic mapping approach based on the Apache VTL template language and discussed in Section 4.2.1. The performance of the *mapping-template* is evaluated in the literature only for the lowering phase [Scrocca21]. An evaluation considering the lifting phase and the benchmarks usually applied to an RML processor could be useful for a comparison with other existing solutions. In this direction, we will also evaluate the possibility of enabling a direct translation of RML mappings to template-based mappings executables through the *mapping-template* component.

Considering constrained devices with limited resources (e.g., CPU/RAM), simpler approaches like programmatic string replacement are often adopted for efficient transformation of data to RDF [Naiema23]. However, these approaches do not offer the advantages of declarative mappings and are difficult to scale and maintain. Within SmartEdge, considering the devices involved in the use cases, we will assess and work on optimisations for the performance and scalability of mapping processors based on declarative mapping languages.

4.3.2.3 Deployment strategies for Apache Camel

One of the most prominent advantages of leveraging Apache Camel is its inherent support for multiple deployment options, which could support flexibility in the deployment of DataOps pipelines. The versatility of Apache Camel's deployment capabilities significantly aligns with the diverse needs of SmartEdge, particularly in scenarios where the software component responsible for mediating message formats and semantics may operate across various environments. In the context of the SmartEdge project, this flexibility allows the DataOps pipelines to be executed on Edge devices, more powerful devices or in the Cloud. This adaptability ensures that the definition of a DataOps pipeline is decoupled from its deployment strategy. In this way, the deployment strategy can be tailored to suit the specific demands of different deployment environments specified by the different use cases.

Apache Camel supports different deployment alternatives:

1. **JAR files** that are self-contained executable for devices that can run a Java runtime. This approach can cover a higher number of use cases, but it may not be suitable for all edge devices. In particular, those positioned on the lower end of the spectrum in terms of processing power and resources may not be able to run a Java Virtual Machine (JVM).
2. **Native Executables** using Camel Quarkus, an Apache project that integrates Camel with Quarkus. Quarkus[51] is a Java framework tailored for producing native applications. This allows Java projects to be packaged as lightweight, fast-booting native binaries by employing an ahead-of-time (AOT) compilation approach. Examples of native executables are ELF binaries for Linux and exe files for Windows. This deployment option can be used for those Edge devices that do not or cannot run Java.
3. **Kamelets** using Apache Camel K, a subproject of Camel tailored for simplified deployment of Camel *Route*s on Kubernetes[52]. Camel K simplifies the process of running Camel-based integrations on Kubernetes by allowing developers to create and execute integrations using Camel DSL as native Kubernetes resources. This facilitates the

[50] https://github.com/cefriel/mapping-template
[51] https://quarkus.io
[52] https://kubernetes.io

seamless integration of Camel's extensive library of components and patterns into the Kubernetes environment, enabling efficient and scalable deployment of integration solutions within cloud-native architectures. Camel K introduces the concept of Kamelets, which are abstractions of Camel routes represented as route snippets. These Kamelets define and reveal the interface inputs and outputs. In contrast to Camel, where the Component is the unit of abstraction, in Camel K, the Kamelet encapsulates an entire *Route*. These Kamelets are executable on Kubernetes clusters where they can be used for serverless data integrations.

Figure 4.14 summarises the deployment options described. The Java JAR file can be obtained as a standalone application using Camel Main[53] or Camel SpringBoot[54]. Both the Java JAR file and the Native Quarkus executable can be packaged using an appropriate OCI container[55] and, optionally, deployed using a container orchestrator such as Kubernetes. Kamelets are meant for Camel K and should be natively run on a Kubernetes cluster.

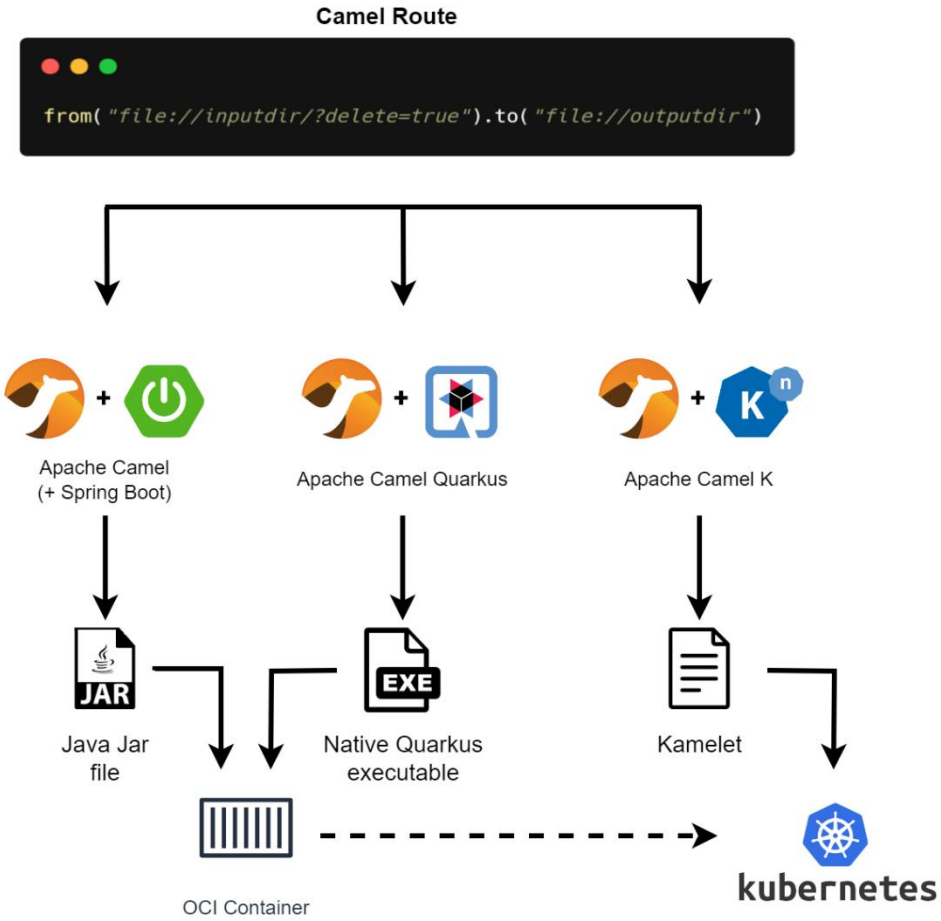

Figure 4.14: Deployment options for an Apache Camel route.

Table 4.1 outlines the advantages and potential issues associated with the different deployment options described.

[53] https://camel.apache.org/components/4.0.x/others/main.html
[54] https://camel.apache.org/camel-spring-boot/4.0.x/spring-boot.html
[55] https://opencontainers.org

Table 4.1: Analysis of PROs and CONs for different deployment options

Deployment	Pros	Cons
JAR file	Easy to build. Easy to deploy to a device, everything necessary is contained in the JAR.	Requires the device to run Java.
Native Executable	Does not require the device to run Java. Faster start-up and execution times than a JAR file.	Creating a native binary demands more CPU power and RAM compared to building a JAR file. While not problematic for one-time route building and deployment, it's essential to consider this when dynamically creating routes. Not all the Java libraries support Quarkus.
Kamelet	Allows an even easier re-use of routes inside of a larger integration. Serverless approach enables scale-to-zero to save resources, and scalability via replication for high traffic loads.	Only makes sense in the context of a Kubernetes deployment.

4.3.2.4 Low-code approaches to define DataOps pipelines

A low-code approach simplifies application development by emphasizing configuration over manual coding. The overall objective is to enable a declarative configuration of components so that users can reduce the need for implementing custom solutions.

For the Apache Camel framework, data integration pipelines are defined using the abstraction of *Route*s rather than direct coding. This abstraction empowers a no-code approach to data integration, as it exposes all available functionalities of Camel components through well-documented URI parameters, which users can configure when creating a route. This approach also means that modifying the data integration pipeline doesn't necessitate rebuilding the entire software artifact that executes Camel routes, it only requires changes to the file where the route is declared.

Routes can be defined using several domain-specific languages (DSL), with the most prominent options being Java, XML, Spring XML, and YAML. Alternatively, routes can be built using a graphical user interface without writing code using Apache Camel *Karavan*[56] and the plugin for Visual Studio Code. This graphical approach significantly eases the process of route definition, as it avoids syntax and logical errors that may happen when manually writing a route in a text file.

[56] https://github.com/apache/camel-karavan

Figure 4.15 shows an example Camel pipeline defined using the graphical user interface provided by Apache Camel *Karavan*. On the left, the corresponding YAML generated by the tool is displayed.

The adoption of Apache Camel for the DataOps toolbox enables a low-code approach for the configuration of pipelines as composition of proper building blocks. We will evaluate the possibility of adding support for Chimera components within Apache Camel *Karavan* to enable also a graphical definition of DataOps pipelines.

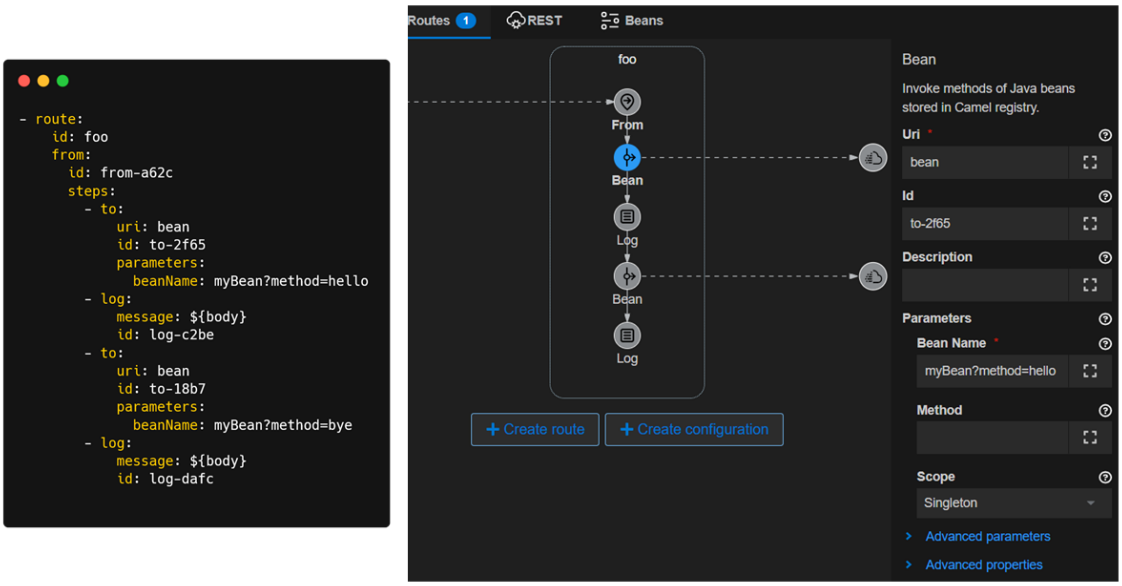

Figure 4.15: An example of a route in YAML (left) and the same route created visually with Apache Karavan (right).

4.4 DataOps Toolbox in SmartEdge

This section discusses how the designed DataOps toolbox can support the different SmartEdge use cases. We identify potential data exchanges that would require a mediation, i.e., a conversion from a structured heterogeneous format (e.g., CSV/XML/JSON) to the standardised semantic interfaces defined in Section 3 or vice versa. Moreover, we provide an initial description of deployment options that the DataOps toolbox should support for its integration considering the architectural constraints of each use case.

4.4.1 Mediated Data Exchanges in SmartEdge

The configuration of different pipelines through the DataOps toolbox will be needed in SmartEdge to support mediated data exchanges. The implementation of such pipelines can be enabled by ensuring the availability for each use case of a set of components:

- *Node data connectors*: requirements for each use case about the data sources to be accessed (both as data providers and as data consumers). Different connectors should be identified or implemented to support heterogeneous protocols and interaction mechanisms (e.g., pull/push).

- *Mapping rules*: requirements for each use case about the data formats and data models adopted and the target standardised semantics. Mapping rules should be declaratively defined to support the conversion of payloads between each pair of input/target data format/model.
- *Mapping processors:* requirements for each use case about the mapping processors considering constraints on performance/scalability and/or about the runtime environment for the execution.

The identification and/or implementation of such components is based on the definition of which mediated data exchanges are necessary in SmartEdge.

Figure 4.16 provides an overview of the data models/formats identified within SmartEdge considering each use case and the standardized semantic interfaces discussed in Section 3. The arrows represent the requirements identified at this stage of the project for the conversion of data from one data model/format to another. The red arrows identify the conversions that can be possibly supported by the DataOps toolbox.

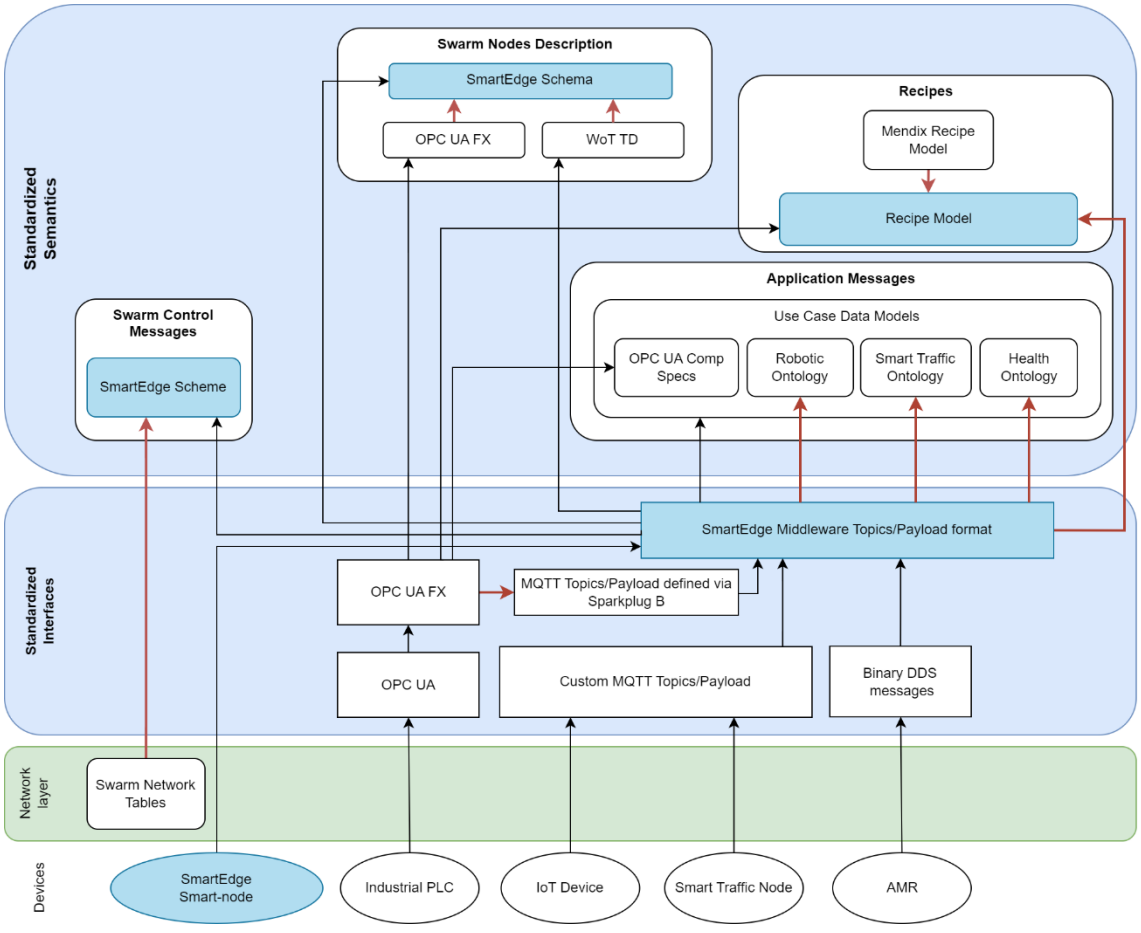

Figure 4.16: DataOps toolbox in SmartEdge

The figure does not consider data models associated with native data exchanges that are supported by default. For example, considering the SmartEdge Use Case 2 (UC2) we excluded

from the analysis the following native data exchanges toward the Traffic Sensor Node fixed at the road infrastructure:

- *Camera/Radar/Lidar* to Sensor Node
- *Connected Vehicle* to/from *Sensor Node*
- *Controller Node to Connected Vehicle*
- *Controller Node to Traffic Light*
- *Sensor Node to Controller Node*: Sensor measurement data sourced from radars/cameras are converted into UC-specific JSON by the Sensor Node before sending out to other nodes

At the bottom of the figure, we represent a set of devices and related data models/formats identified as swarm nodes by the different use case:

- Industrial PLC communicating through OPC UA (Use case 4);
- IoT Devices and Smart Traffic Nodes exchanging messages through custom MQTT topics/payload (Use case 1, Use case 2, Use case 3, Use case 4, Use case 5A/5B);
- Autonomous Mobile Robot (AMR) exchanging binary messages through DDS (Use case 3).

Finally, we represented a SmartNode that identifies a node in the swarm implemented by SmartEdge and thus directly able to communicate according to the standardized semantic interfaces defined by the project.

The interoperability among different protocols can be enabled as discussed in Section 3.4 thus reducing the amount of data models/formats to be considered:

- Messages exchanged using a set of topics in the SmartEdge middleware (*format* defined by the middleware selected for the SmartEdge architecture, *semantics* possibly customly defined for each topic);
- OPC UA data exchanges standardised through the OPC UA FX and other companion specifications.

The possibility of defining a single standardised interface through the SmartEdge middleware also for OPC UA data exchanges will be evaluated. In this case, the DataOps toolbox may support the conversion from OPC UA FX to an intermediate representation adopting MQTT and Sparkplug B.

The description of each node, i.e., what are the interactions available for each node and how they can be invoked, should be harmonised according to the definition of skills and capabilities in the SmartEdge Schema. In this context, the DataOps toolbox may support the conversion from OPC UA FX descriptions or from WoT Thing Description to the SmartEdge Schema.

Considering application messages, the DataOps toolbox can be employed for the conversion of custom payloads received via the SmartEdge middleware to the standardised semantics (e.g., Robotic ontology, Smart Traffic ontology, Health ontology) defined for SmartEdge.

For example, within the SmartEdge UC2 the payload generated by the *Sensor Node* is a JSON string in a structured use-case-specific format. The data format aims primarily at seamless programming, network bandwidth and computation performance, therefore does not necessarily abide by standard ontology schemas. The JSON string in Figure 4.17 is an example

of sensing data denoting current states of a bunch of vehicles moving near intersection "*fi.helsinki.270*" in Helsinki.


```
{
  "source": "radar.270.3.objects_port.json",
  "road_segment": "fi.helsinki.270.271",
  "junction": "fi.helsinki.270",
  "tstamp": 1684146235087,
  "nobjects": 2,
  "objects": [
    {
      "id": 92,
      "lat": 60.1598,
      "lon": 24.9214,
      "speed": 0.001,
      "class": 8,
      "lane": "D2",
    },
    {
      "id": 72,
      "lat": 60.1596,
      "lon": 24.9216,
      "speed": 0.3408,
      "class": 2,
      "lane": "A1",
    },
}
```


Figure 4.17: Example JSON string from Use Case 2

Such messages are passed among our nodes via the MQTT pub/sub mechanism and may be made available to other nodes via the SmartEdge middleware. The above *Sensor Node* JSON strings need to be *lifted* into semantic JSON-LD/RDF strings, matching with the defined standardised domain-specific ontology, i.e., the Smart Traffic Ontology.

Considering the runtime interactions for the execution of a recipe and/or the assignment of roles, tasks, and instructions to certain nodes, it may be necessary to map messages to the SmartEdge Schema or Recipe Model in RDF. Also in this context, the DataOps toolbox may support such conversion. Moreover, it may be necessary to map certain information between the RDF representation according to the Recipe Model and the structured data format (e.g., JSON) fulfilling the requirements of the Mendix tool.

Finally, certain swarm control messages will be directly provided by the SmartEdge network layer for scenarios associated with dynamic swarm formation and management. The adoption of the DataOps toolbox to support the conversion of such messages (e.g., from Swarm Network Tables) to the SmartEdge Schema may be evaluated.

4.4.2 Deployment of DataOps pipelines in SmartEdge

Considering the SmartEdge use cases, different constraints may emerge for the deployment of the DataOps toolbox. For this reason, we identify different options that should be potentially supported for the integration of the DataOps toolbox components within the swarm:

- **Within a dedicated smart-node**: a dedicated component for the mediation of data exchanges is deployed and is considered as an additional node in the swarm providing data interoperability capabilities.

- **Embedded in the source/target smart-node:** the component for the mediation of data exchanges is deployed as part of either the source or the target smart-node and can be integrated or as a library, if the source code of the runtime of the smart-node is compatible, or as an external component invoked by the smart-node for each message to be sent/received. In this case, it is preferable to deploy multiple components for each smart-node requiring a mediated data exchange.
- **Embedded in the swarm orchestrator:** the component is deployed as part of the swarm orchestrator and can implement mediated data exchanges for different nodes involved in the swarm and communicating through the orchestrator.
- **Embedded in the middleware/network layer**: the component is deployed as part of the middleware/network layer and can implement mediated data exchanges for different nodes involved in the swarm, possibly even in a transparent way (i.e., without explicit interaction by the nodes).

Such choice may be dependent on specific constraints of each use case, for example, it may be guided by the availability of more performant nodes to reduce the latency or by specific constraints for the deployment/integration of the DataOps toolbox in certain nodes (e.g., licensing or constraints on software installation). For each option, further requirements may be elicited depending on the hardware/software involved.

5 CREATION AND ORCHESTRATION OF SWARM INTELLIGENCE APPS

The chapter provides the current work on design and implementation of an orchestrator for Swarm apps in the SmartEdge environment. These apps will be instantiated from existing templates, targeted to match the existing devices, which would integrate the swarm. The description of the devices and their capabilities would be specified using the declarative and semantic models proposed in Task 3.1. Moreover, these capabilities would be matched with the specification of the templates, thus enabling the orchestration of the app, and initiating the interactions among the swarm devices. The templates themselves will be specified using semantic descriptors, and the data flows will be enacted using the DataOps infrastructure provided in Task 3.2 and instantiated using the Mendix framework.

The contributions of Task 3.3 are:

Recipe Model implementation in Mendix
- SmartEdge Swarm Orchestrator in Mendix

5.1 STATE OF THE ART – ORCHESTRATION OF SWARM EDGE APPS

With the wide availability of IoT/WoT edge devices for sensing and actuation, it becomes increasingly important to facilitate the orchestration and deployment of these systems. Given the complexity of the configuration of edge platforms, including the setup of device interconnections, integration of inputs/outputs, definition of tasks to be performed, fallback scenarios, etc. [Abbas17].

5.1.1 Cloud/Edge Deployment

In many cases the deployment of edge intelligence solutions requires heavy involvement of technical experts that must manually prepare and configure the participating devices. This makes it challenging to introduce changes in the device organisation, replace nodes, or modify the tasks to be performed. In the domain of cloud computing, the deployment of production systems is automatised using frameworks like Kubernetes [57] [Al Jawarneh19], which decouples the execution runtime of the individual nodes from the orchestration of the entire cluster. Containerised services in this type of environments allow high flexibility as it allows for programmable interfaces that developers can use for monitoring, node synchronisation, event management, scaling, failure handling, etc. However, platforms like Kubernetes are better fitted for data centres, which work in very different conditions with respect to edge devices in IoT. Among these we can mention the network reliability, or the dynamicity of incoming data from devices that can abruptly join or quit the system [Alberti13].

In the SmartEdge project, some of the edge nodes are considered to have self-organising capabilities, enabling the formation of Swarms of devices. In this context, making use of intelligence capabilities at the node level is essential for orchestration and deployment. In order to adapt Kubernetes to edge environments, different lightweight frameworks exist, such as K3s[58], Microk8s, KubeEdge [Xiong18], or Kubelets [Goethals20], which are designed to adapt and perform better in scenarios where edge devices and sensor nodes are in charge of the

[57] https://kubernetes.io/
[58] https://k3s.io/

application execution. Other approaches exploit geo-graphical relationships among nodes, like Oakestra [Bartolomeo20] or OneEdge [Saurez21], which optimises scheduling of edge devices, beyond the Kubernetes-based solutions. Other efforts in this line include CloudPath [Mortazavi17], HeteroEdge [Zhang19] or SpanEdge [Sajjad16], more focused on specific aspects such as streaming data applications. Other approaches more related to Fog Computing include open-source project FogLamp [59], while network-specific works like VirtualEdge [Nguyen16] are centred on edge nodes for cellular networks.

5.1.2 Semantic IoT Platforms and WoT APIs

Semantics provide the ability to create abstractions that capture the essential capabilities, goals, roles, and tasks performed by edge and IoT nodes [Thuluva20]. The advantage of using such descriptions is that they provide the means to abstract the actual implementation or device specific functions from the description of the task(s) that the devices must execute. This enables discoverability of devices, as well as enhanced flexibility, so that different nodes (e.g., from different vendors) could fulfil a given task if they comply with the description of the device capabilities. Moreover, the semantic description of IoT devices may allow orchestrators and coordinator nodes to look for, or replace execution nodes when needed, using the semantic metadata as a catalogue, or directory.

At a development level, these semantic descriptions can be created, accessed, or modified through an API. In the BIG IoT platform, for example, a generic API is used as a bridge between existing platforms, delegating interoperability to the semantic model. Other APIs, such as the one provided in the meSchup IoT platform [Kubitza17], have a deeper control over the devices in the system, although with less flexibility regarding device interoperability. Beyond these APIs, other attempts have been made to bridge applications and IoT devices, such as the Semantic Gateway Service [Desai15], or the Semantic Information Broker, which incorporates the notion of discovery in IoT environments, implemented as “smart spaces” [Viola16].

In the realm of semantic integration and orchestration, there has been a considerable amount of work regarding Semantic Web Services [Calbimonte20]. This concept includes the publication of services following the Web standards, including semantic annotations and ontologies that enable their automatic discovery, composition, interconnection, and invocation. The composition of Semantic Web Services requires languages that formally describe inputs and outputs and indicates the way in which services can be integrated. While the Web Service Description Language (WSDL) [60] has been used to address the syntactic aspects of service description and consumption, for the semantics it was needed to extend the language, for example through approaches like OWL-S, WSMO, or SAWSDL.

Following the evolution of Web Service APIs towards a design based on the REST principles, beyond the limitations of SOAP/WSDL, alternatives such as RESTdesc, hRESTs, WSMO-light, or MicroWSMO emerged, offering different levels of service description for Web APIs [Lucky16]. However, most of these solutions are oriented towards back-end services and need to be adapted to WoT environments. Application integration solutions at the application level, such as IFTTT or Node-RED[61] have been used in previous works, especially for composing complex

[59] https://dianomic.com/platform/foglamp/

[60] https://www.w3.org/TR/sawsdl/

[61] https://nodered.org/

workflows in IoT environments, using individual services as building blocks. These concepts of pipelining components can offer a powerful and flexible architecture for loose coupled WoT services. In combination with semantically rich models (i.e., based on RDF vocabularies), descriptions of WoT processes can serve as the basis for a seamless orchestration of autonomous edge nodes. Nevertheless, to this point there is no solution yet that allows this type of orchestration in swarms of IoT edge devices, using templates that provide semantic descriptions of swarm goals, tasks, and background knowledge.

5.1.3 Semantic Descriptions of Devices for Orchestration

Orchestration of edge devices requires having a common representation of their main characteristics, expressed as capabilities, roles, tasks, etc. The usage of semantic models to enable orchestration has been a key element in previous works. Different ontologies have been proposed to tackle this issue, most notably the SSN (Semantic Sensor Network) ontology, and its successor, the SOSA ontology [Janowicz19]. These models are not meant to be used on their own, but in combination with other vocabularies, e.g., QUDT for expressing quantities, or domain-specific ontologies. Examples of these ontologies include Smart Applications REFerence (SAREF)[62], targeting smart appliances, brick for building management, etc.

Specifically for the Web of Things domain, the W3C Web of Things Thing Description 1.1 Recommendation (TD)[63] is a semantic model designed to serve as an entry point of a Thing. The TD is composed of mainly the following elements: metadata descriptions of the Thing, affordances that specify properties, events, and actions possible with the Thing, as well as the data schema, security mechanism information, and links to related Web resources. For the orchestration of Things, the TD affordances provide essential information regarding the ways in which other nodes can use/interact with it. First, the orchestration service can identify the Things that offer capabilities needed by the tasks specified in a Swarm recipe, or template. The TD specification also includes the concept of a Thing Model, a logical description of the potential interactions with a “class” of things. A Thing Model can include the properties, actions, and events exposed by a type of Thing, although it does not contain individual information about an instance of the Thing (e.g., concrete address, serial number, or other specific data).

5.2 Design of the Swarm Orchestration

One of the goals of SmartEdge is to provide low-code tools for configuring and orchestrating Apps that manage and run on a Swarm of edge devices. To achieve this, in WP3 we envision the usage of tools that allow users to visually construct the application with drag-and-drop functionalities, and intuitive interfaces.

5.2.1 Design-time Orchestration Tooling

In order to enable the configuration and creation of recipes, it is needed to use low-code tools that allow designing the flow and components of the Swarm App, as well as its inputs and outputs. In this section we describe the characteristics of the Mendix[64] framework that will precisely be used and extended for this purpose.

[62] https://saref.etsi.org/

[63] https://www.w3.org/TR/wot-thing-description11/

[64] https://www.mendix.com/

The Mendix framework and ecosystem allows IT teams and App developers to accelerate the creation and entire lifecycle of digital solutions including the following features:

- Rapid development of applications: It allows visual development of complex applications, as well as provides support for automated deployment out of the box.
- Extensibility: The Mendix platform is extensible, with the possibility of reusing components, adding widgets, connecting to different data sources, and customizing behaviours through APIs and code snippets.
- Cloud capabilities: *Containerisation* of Mendix apps is provided by default, enabling both cloud and local installations easily as addressed in functional requirements LC-006, LC-007 and LC-008 in Table 2.2.
- Marketplace: A large number of reusable components are available, including widgets, modules, etc. which are provided by the Mendix community.
- Security privacy: Security and governance features are available by default in Mendix, enabling trusted use of its components.

With these considerations at hand, the Mendix platform provides an interesting starting point for enabling the orchestration of SmartEdge Apps using a low-code approach. The two main components of Mendix relative to the orchestration are the Mendix Studio, and the Mendix Runtime.

Mendix Studio Pro (currently in version 10.x) is a visual model-driven IDE with customizable themes, drag-and-drop functionality, reusable components, and full-stack capabilities. This is shown in Figure 5.1 and Figure 5.2. The orchestration in SmartEdge will be configured at design time using this tool, which permits organizing the different data sources (e.g., coming from edge nodes), establishing a flow of tasks and computations that need to be performed, and the nodes that are involved.

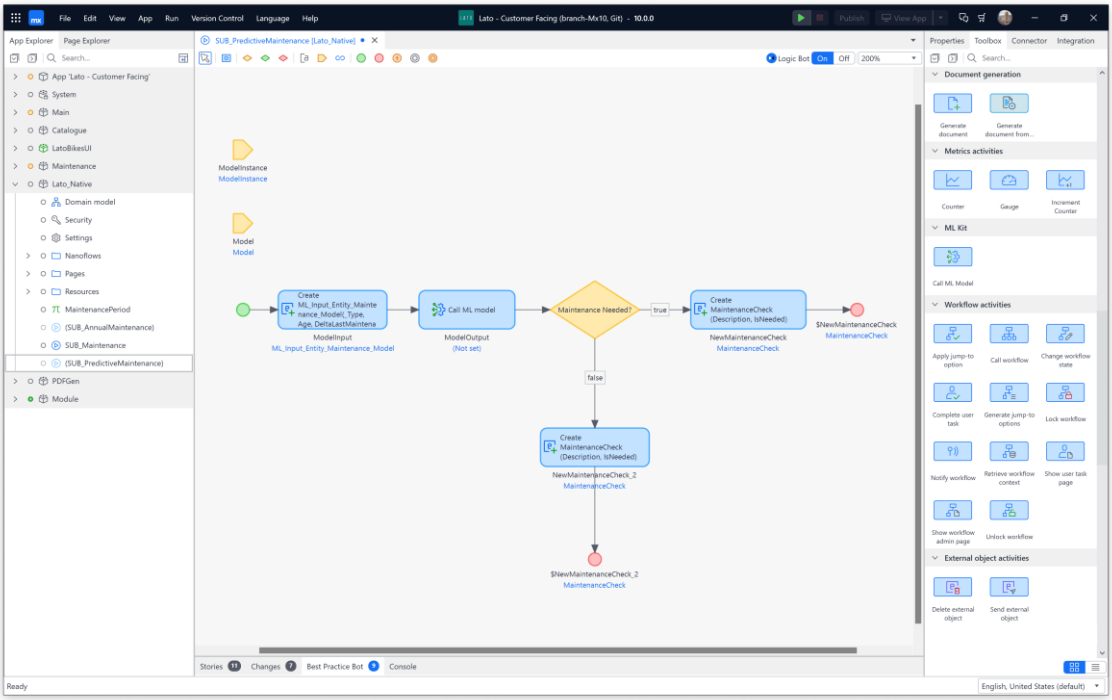

Figure 5.1: Mendix Studio environment for designing the App UI.

However, the current Mendix functionalities do not allow for the usage of ontologies to represent nodes in the system. Although there are different connectors available in the Mendix marketplace (e.g., Bluetooth) for certain types of devices, there is not really a concept of a group of nodes, and even less the concept of a Swarm of nodes. As we will see later, these are functionalities that will be incorporated as part of the SmartEdge WP3 developments.

5.2.2 Low-code Runtime Execution Tooling

Once the design of the App and its components has been prepared, the actual execution is taken in charge by the Mendix runtime component. The Mendix Runtime is essentially an interpreter of a Mendix model, enacting input data ingestion, executing microflows & nanoflows, displaying data results in App pages, etc. The Medix Runtime can be decomposed in two parts, the Mendix Runtime Server and The Mendix Client.

Mendix Runtime Server: It is the part of the Mendix Runtime that is in charge of executing microflows, as well as connecting to different data sources and external services. The Runtime server communicates answering requests to Mendix Clients. The Runtime Server can be deployed on the cloud or locally, e.g., for testing. The requests from the clients are processed and data is returned as a result, following the Mendix model that describes the microflows and the application logic. The Runtime Server follows a stateless service pattern, allowing horizontal scaling.

Mendix Client: It is the runtime component that runs on the devices of the end-users, acting as an interface with the domain-specific Application. The Client is decoupled from the Runtime Server, and thus can execute certain processing flows locally, only requesting the server when necessary. Mendix-based applications can be deployed as Mobile or Web, Mendix Client runs on both cases. In the case of Web applications, the Mendix Client is launched in JavaScript on a single dynamic HTML page. In the case of a Mobile App, the Mendix Client is installed as a React Native[65] application.

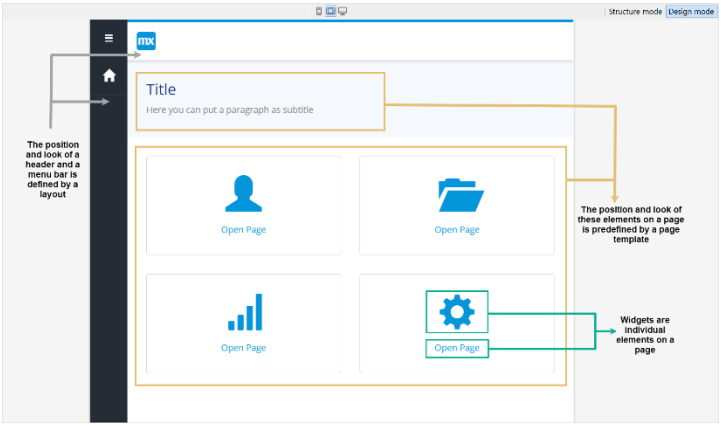

Figure 5.2: Mendix Studio environment design mode.

5.2.3 Swarm Apps Application Logic Design

As explained in Section 3, the SmartEdge Swarm Apps will be configured using the so-called recipes, which are templates or blueprints that specify the goals of a Swarm, and the processes that are needed to accomplish them. In the use cases where Mendix is used as visual

[65] https://reactnative.dev/

development tool, and as engine for the orchestration of the SmartEdge App, the application logic has to be specified using the concepts of microflows and nanoflows. These flows allow performing different types of activities, including branching logic, updates on data, or displaying content, all in a declarative way.

In the case of microflows (Figure 5.3) these run on the Mendix Runtime Server, and thus can access cloud services, external data sources, etc. but cannot run offline. In contrast, nanoflows run directly on the device (or browser for Web apps), therefore being able to be executed offline and autonomously. Depending on App restrictions and autonomy requirements and performance, a combination of microflows and nanoflows can be used in the Studio designer.

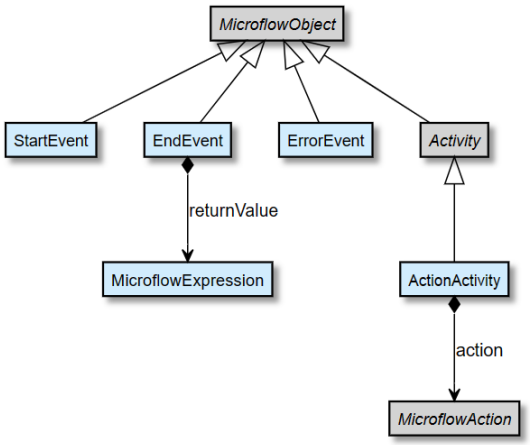

Figure 5.3: Microflow hierarchy in the Mendix model.

The notation of both nanoflows and microflows is based on BPMN (Business Process Model and Notation) [Recker06], which is a well-known standard for representing workflow processes. An instance is shown in Figure 5.4. The elements that compose a nanoflow or microflow can fall under the following categories:

- **Activities**: Elements that represent the actions that are executed in a microflow or nanoflow.
- **Events**: Elements represent start and end points of a microflow or nanoflow, as well as special operations inside a loop.
- **Flows:** Denote the connection between elements.
- **Decisions:** Elements that indicate making choices and merging different paths of a flow.
- **Loops:** Used to iterate over a collection of objects.
- **Parameter:** Elements represent data that is used as input for the microflow or nanoflow.
- **Annotations**: Elements that can be used to add comments or annotations in a microflow or nanoflow.

Besides the fact that nanoflows run on the client, and microflows on the server, there are other important differences between the two. First, in nanoflows the client actions are immediately executed as the steps of the flow are run. Conversely, in microflows the client actions only run after the client runtime receives the response from the server. Also, several types of expressions cannot be used in the same way (e.g., to obtain the current session, etc.). The same applies for action elements. Some are available only for nanoflows, and some only for microflows. Regarding transactions, these are only run in microflows. For nanoflows, in case of errors there is no automatic rollback procedure. Moreover, there are differences linked to the dependencies and limitations of the libraries used by nanoflows (i.e., JavaScript libraries) and by microflows

(i.e., Java libraries), which may also have impact on the functionalities of the flow actions and elements.

Figure 5.4: Example of a Mendix flow including different steps.

5.2.4 Semantic Representation of Swarm App Recipes

The Swarm Apps in SmartEdge are configured using templates of applications' specifications, called recipes. These recipes indicate the requirements of an App, as well as the capabilities of the IoT and other devices that form the swarm. A recipe also specifies and links to the dataflow and the business logic of an App (e.g., using a standard like BPMN), in a declarative format. In consequence, for those Apps in SmartEdge where this design is performed using Mendix, the App flow will be represented using the Mendix Metamodel, and in particular microflows or nanoflows, depending if they run on the Mendix Runtime server, or the Mendix Client.

Beyond the default features of Mendix, SmartEdge recipes are specified using semantically enabled recipes. This means that the recipe itself will be stored in RDF format, including the different elements of the App flow. More concretely, and as discussed in Section 3, it includes: (i) the goal of the task to be executed by the Swarm; (ii) required conditions to start the task; (iii) capabilities of the nodes needed to perform the task; (iv) completion conditions for the task; (v) steps and transitions among these steps specified as a flow; (vi) topics and events messages produced during the execution of the transitions.

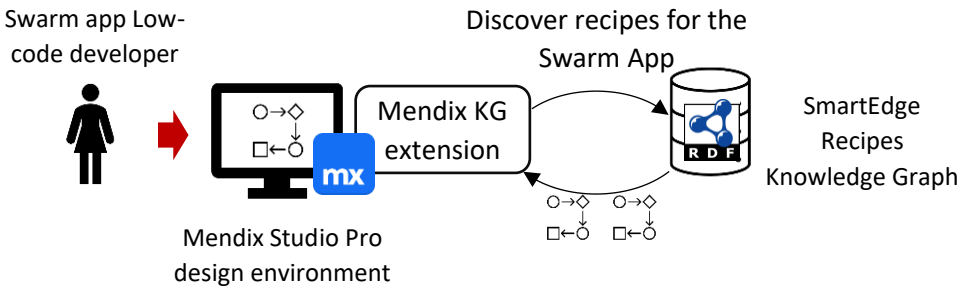

Figure 5.5: Discovery of semantic recipes for the requirements of a Swarm App in SmartEdge.

One of the key elements of the recipes is the specification of capabilities of the Swarm nodes as in Figure 5.5 and Figure 5.6, which are required to complete a given task, and comply with the goals of the recipe. These capabilities represent functional requirements of an application, which can be described as skills. The semantic description of these capabilities can be implemented using semantic models for the description of Web of Things. Concretely, the TD proposes ontology terms for describing affordances. The TD affordances provide machine-understandable metadata of a Thing that shows the possible choices that consumers have to interact with a Thing. As an example, the following snippet in JSON-LD represents an action affordance indicating the possibility of any interaction to turn on or off a connected lamp.

```
  "actions": {
```

```
        "toggle": {
            "description": "Turn on or off the lamp",
            "forms": [
                {
                    "href": "coaps://mylamp.example.com/toggle",
                    "cov:methodName": "POST",
                    "op": "invokeaction",
                    "contentType": "application/json"
                }
            ],
            "safe": false,
            "idempotent": false
        }
    },
```

Therefore, when building a recipe in a low-code environment like Mendix, the KG extensions of SmartEdge will allow specifying capabilities using semantic metadata like the TD affordances. In this way, attached to the Mendix microflows containing the application logic and conditions, it will contain all elements expected in a recipe, i.e., goals, sub-tasks, capabilities, prerequisites, etc.

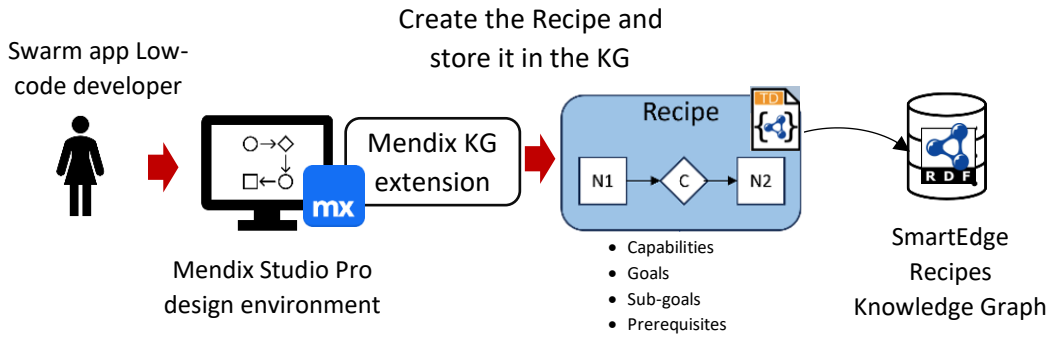

Figure 5.6: Creation of a Recipe using a Low-code environment in SmartEdge.

Nodes can provide different but complementary capabilities, and the recipe should specify how the interactions among them are configured to complete a task. The metadata of these interactions include the source and destination capabilities, which for the Low-code designer will allow identifying Swarm nodes that can provide these capabilities. Moreover, the business logic will be attached to the recipe using a business process language, as it is the case with Mendix. The domain models (as shown in Figure 5.7) will use the semantic extensions to incorporate terms from external vocabularies, to foster interoperability in SmartEdge recipes.

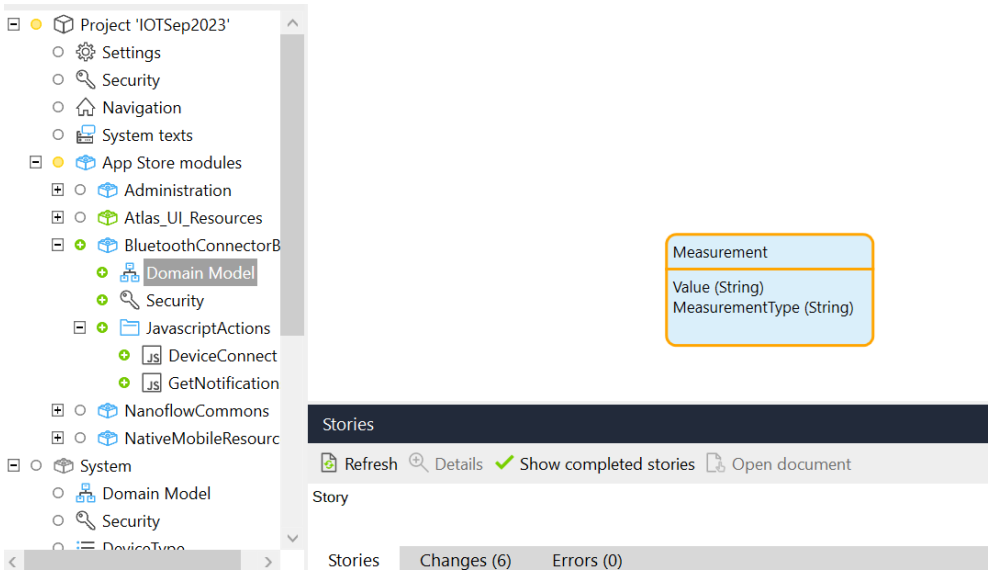

Figure 5.7: Domain model specification in Mendix.

Regarding the recipe metadata, a different number of ontologies need to be used/reused (shown in Figure 5.8). During the design time of the recipe, the following vocabularies and/or ontologies might be referenced:

- As explained before, TD vocabulary will be used and extended to indicate capabilities.
- For sensor and system descriptions, including technical details of the device specifications can be indicated using the Semantic Sensor Network ontology SSN/SOSA.
- To describe node-related metadata, including Swarms, SmartEdge nodes, sensor nodes, orchestrator, coordinator, etc., the newly created SmartEdge ontology will be employed.

Moreover, depending on the use-case, domain specific ontologies will be used to describe the goals, and technical details of the capabilities. This might include generic ontologies for cross-domain aspects like units of measurement (e.g., QUDT, UM), or general knowledge ontologies like Wikidata [66] and Schema.org [67]. Furthermore, specific ontologies for healthcare, manufacturing, or robotics might be referenced when needed.

[66] https://www.wikidata.org/
[67] https://schema.org/

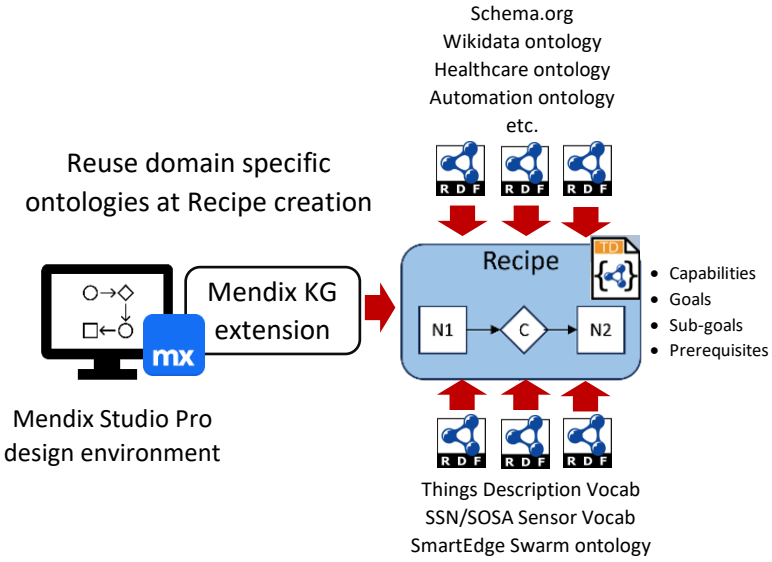

Figure 5.8: Reuse of the SmartEdge ontologies and other external vocabularies at design time.

5.2.5 Instantiation and orchestration of Swarm Apps

With the planned extension for Mendix regarding the creation and search of recipes using semantic metadata for capabilities, goals, prerequisites, etc., it should be possible to instantiate these recipes in a given SmartEdge environment.

The instantiation should be able to provide a matchmaking functionality, in order to map the capabilities needed by the recipe with the capabilities offered by the nodes in the Swarm.

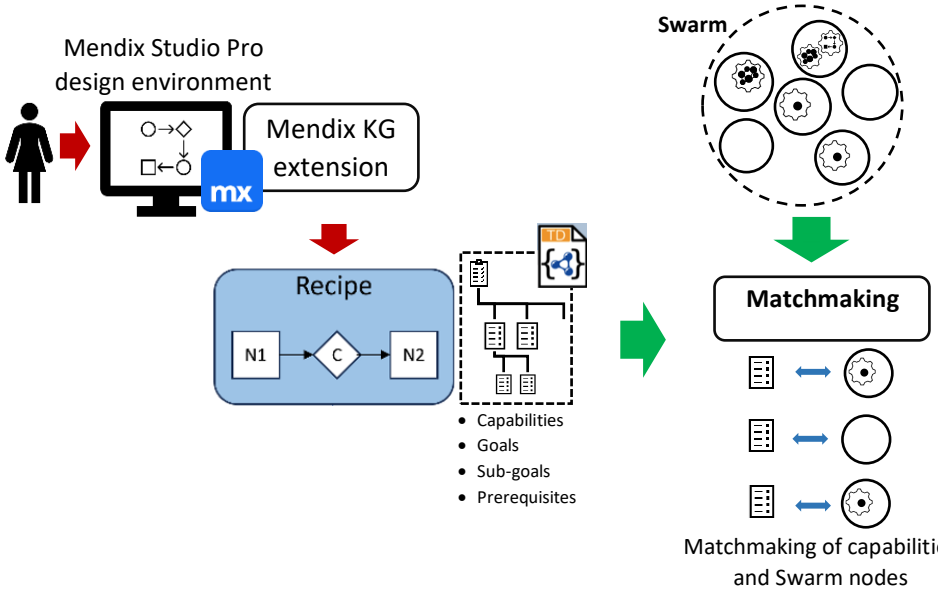

Figure 5.9: Matchmaking between the capabilities required in the recipe and the nodes available in the Swarm at design time.

The matchmaking step that addresses functional requirement LC-005 in Table 2.2 shown in Figure 5.9 will be crucial for the design and implementation of an orchestrator for Swarm apps

in the SmartEdge environment. While in the recipe the capabilities are presented in an abstract form, after the matching there are specific SmartEdge nodes that adopt a given role within the Swarm. These Apps will be instantiated from existing templates (recipes), as functional requirements CSI-010, CSI-015 in Table 2.3, targeted to match the existing devices, which would integrate the swarm. The description of the devices and their capabilities would be specified using the declarative and semantic models developed in Task 3.1. and described in Section 3. As explained above, the node capabilities will be matched with the specification of the recipes, thus enabling the instantiation of the App recipe, and configuring the interactions among the swarm devices. The recipes themselves will be specified using semantic descriptors, and the dataflows will use extensions that allow using the DataOps infrastructure provided in Task 3.2 (see Section 4) and instantiated using the Mendix framework.

In summary, the orchestration will need to follow the steps described below, during the design time:

- SmartEdge capabilities matching – Using the common semantic schema (SmartEdge schema) defined in Task 3.1, the matching component will map application requirements to device capabilities in order to help discover the Swarm nodes that can help contribute to the recipe goals. SmartEdge schema constraints will be specified using the SHACL language.
- Recipe model orchestrator – This component will extend the low-code Mendix app to support the specification of semantic recipe models, which will be used to orchestrate the Swarm apps with the devices matching the needed requirements as addressed in functional requirements CSI-016, CSI-017 and CSI-018 in Table 2.3.
- Dataflow orchestration support – This component will use the Task 3.2 DataOps infrastructure to link the needed data sources (data streams, linked datasets, etc.) to the orchestrated Swarm application.

As an example, if we consider the case of an App that requires capturing sensor data from a Bluetooth connector (Figure 5.10). Using a Low-code app like Mendix, the developer should be able to build a microflow and add connectors for different data sources (Figure 5.11).

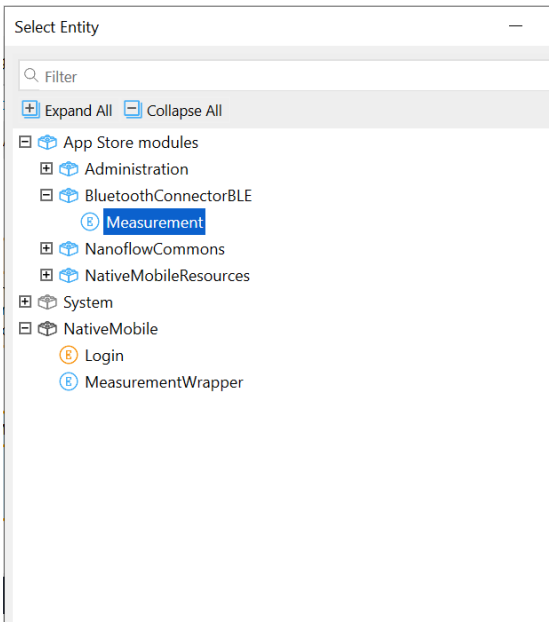

Figure 5.10: Adding a Bluetooth connector in Mendix.

Then, the application logic of the App can be built using different types of elements, including other data connectors, conditions, sequences, etc. At this point, the extension for Knowledge Graph support should allow adding the capabilities in terms of the TD vocabulary. It should also allow connecting to the Knowledge Graph to reuse terminology from domain-specific vocabularies.

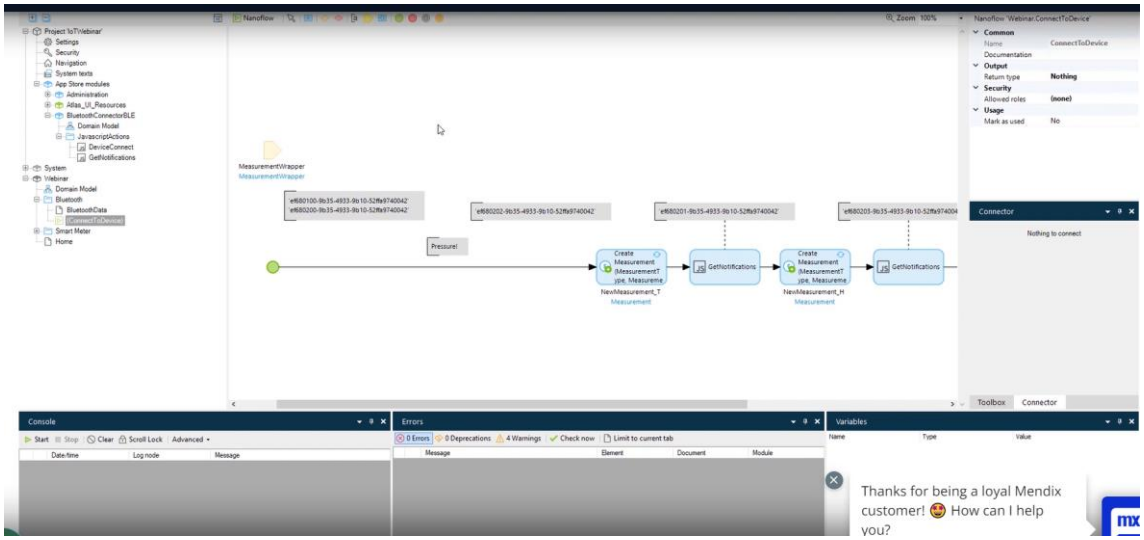

Figure 5.11: Example of a flow development in Mendix.

Through these extensions of the Low-code development environment Mendix, the semantic recipes will be stored in the SmartEdge Knowledge Graph, enabling the discovery of suitable recipes. In this manner, when a new App is developed, it will be possible to find existing recipes that can be reused and/or extended, thus avoiding double work. Within a recipe, a set of roles and tasks and different dataflows can be established, so that there is a common and machine understandable description of what the swarm should do. The semantic vocabularies will be integrated into the SmartEdge ontology (defined in Task 3.1) and will include existing standards such as the W3C Things Description for the Web of Things, as mentioned earlier in this section. The developer will use the Mendix Studio Pro development environment to modify these recipes if any changes are necessary, or if new versions of the recipe are required. All of the recipes will be represented as RDF documents stored in the Knowledge Graph.

Once a semantically enriched recipe is created or selected, it can be instantiated, so that the different roles specified can be assumed by different nodes in the swarm. The instantiation in a first step will consist in matching each of the required capabilities with a Swarm node that is able to comply with the recipe requirements. If there is no available node for a certain mandatory part of the App flow, then the design-time environment should alert that it is not possible to instantiate the recipe successfully.

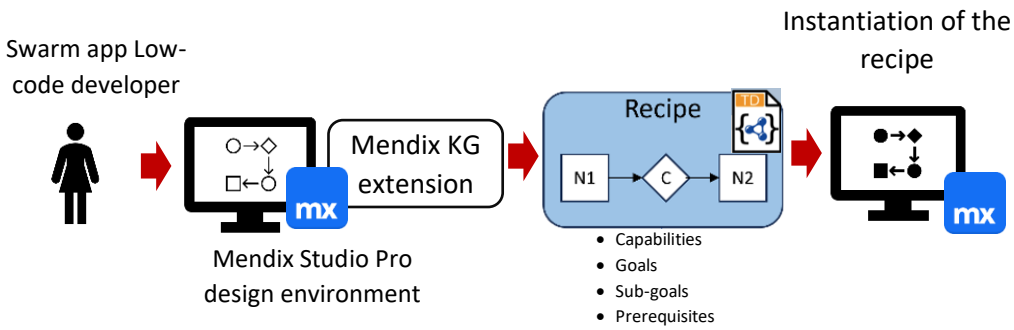

Figure 5.12: Instantiation of a SmartEdge recipe

Otherwise, if all the preconditions and capability requirements are met, the recipe is instantiated as in Figure 5.12, and then passed on to the orchestrator, who will be in charge of executing the recipe in runtime. This will include the coordination of the operations to be executed by different nodes in the swarm. Therefore, the orchestrator will also include the instantiation of the semantic description of the tasks, goals, sub-tasks, and skills established in the recipe. With this information the coordinator will be able to know, for example, that it needs the participation of SmartEdge smart-nodes with certain skills (e.g., stream reasoning over sensor measurements). The coordinator will then need to find and discover which nodes comply with these requirements. In certain cases, the orchestrator may not find the necessary resources to achieve the recipe, and it could either fail or latently wait until the necessary resource can be scheduled. In case of a successful node discovery, then the orchestration itself will be organized as addressed in functional requirements CSI-019 and CSI-020 in Table 2.3.

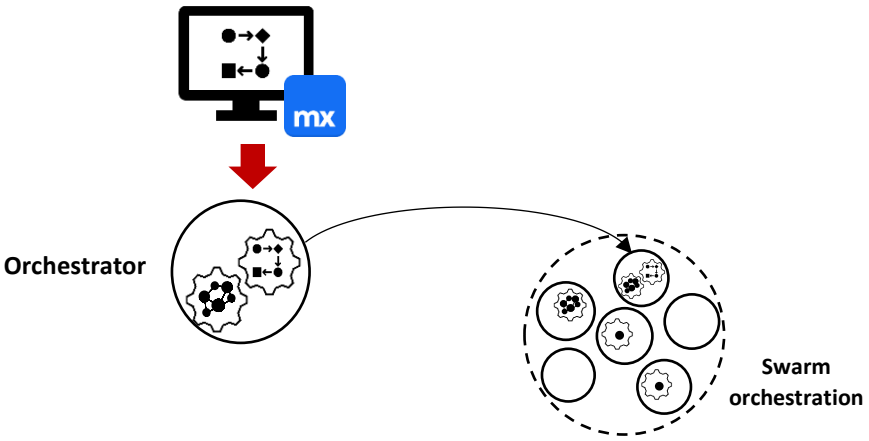

Figure 5.13: Orchestration of the swarm using the instantiation of the recipe received by the orchestrator from the design-time tool.

The orchestrator, as shown in Figure 5.13 may often coincide with the swarm coordinator, and as such will organize the forming and management of the swarm, with different nodes joining or leaving when needed. In certain cases, the SmartEdge nodes may have the capability of holding their own semantic descriptions (e.g., skills or affordances) so that they can be found and incorporated at runtime in the Swarm. In any case the Orchestrator will need to perform a matching operation between the set of available nodes and the different roles established in the recipe, and respect the specification of skills, goals, and tasks. Once this has been established, the different nodes will be able to start operating and exchanging the different data/messages, potentially using the DataOps components of T3.2. After the orchestrator runs

the specified microflows, then the UI of the App will be able to process data and/or display it to the end-user, as in Figure 5.14, with the Bluetooth connector.

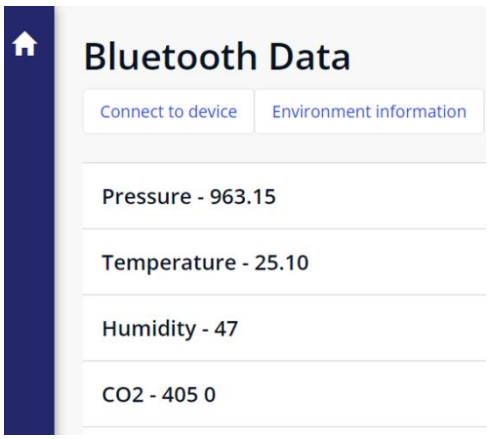

Figure 5.14: Mendix end-user App interface, connecting to data from the edge device.

6 CONCLUSIONS

This document introduced the concept of Continuous Semantic Integration (CSI) in the SmartEdge project (Section 1.1). This concept is broken down into (i) Standardized Semantic Interfaces (Section 3); (ii) DataOps toolbox for semantic management of things and embedded AI apps (Section 4); (ii) Creation and orchestration of Swarm Intelligence apps (Section 5).

This deliverable contributes to Obj.2: Middleware and tools for continuous semantic integration allowing the SmartEdge solution to interact with devices according to a (i) standardized semantic interface, via a (ii) continuous conversion process based on declarative mappings and scalable from edge to cloud, and (iii) providing a declarative approach for the creation and orchestration of apps based on swarm intelligence.

KPIs relevant for this deliverable and Work Package 3 (WP 3) are presented in this deliverable (see Table 2.4). The goal of this deliverable is in the first place to provide design of tools for Continuous Semantic Integration. We will report the progress towards KPIs in the first implementation of this work, i.e., in D3.2. The work presented so far is based on requirements from SmartEdge use cases and the work from D2.1. Our design of CSI will be revisited in deliverable D3.2 and will be based on requirements from D2.2.

Apart from the design, this document provided the first specification of SmartEdge Schema. This schema formally defines important concepts of the SmartEdge architecture, which are used in swarm formation and execution. Further on, we defined a Recipe Model as a means to create swarm applications based on compositions of one or more things or IoT offerings. We have started the implementation of a low-code tool for configuring and orchestrating applications based on the Recipe Model. In order to process data from IoT devices in a unified way, we have identified protocols and standardized information models to be used in our implementation of use case demonstrations. We conducted interviews with use case owners to specify requirements for Standardized Semantic Interfaces (in addition to requirements from D2.1). We also proposed the design of the SmartEdge DataOps toolbox supporting the continuous integration of things and applications through the standardized semantic interfaces. The approach enables interoperability solutions based in a declarative and low-code manner.

The successor of this deliverable, i.e., D3.2 will also provide the first implementation of tools for Continuous Semantic Integration. The first implementation of CSI will be extended with the Low-code Programming Tools for Edge Intelligence from WP5 and will be used for the implementation of use case demonstrations in WP6.

7 REFERENCES

[Abbas17] N. Abbas, Y. Zhang, A. Taherkordi and T. Skeie, “Mobile edge computing: A survey”, IEEE Internet of Things Journal, Vol. 5(1), 2017, pp. 450-465.

[Akhtar08] Akhtar, Waseem, Jacek Kopecký, Thomas Krennwallner, and Axel Polleres. 2008. “XSPARQL: Traveling between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage.” In The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings, edited by Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, 5021:432–47. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-540-68234-9_33.

[Al Jawarneh19] I. M. Al Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari and A. Palopoli, “Container orchestration engines: A thorough functional and performance comparison,” IEEE International Conference on Communications, May 2019, pp. 1-6.

[Alberti13] A. M. Alberti and D. Singh, “Internet of Things: Perspectives, Challenges and Opportunities,” International Workshop on Telecommunications, 2013, pp. 1–6.

[Arenas-Guerrero21] Arenas-Guerrero, Julián and others. 2021. “Knowledge Graph Construction with R2RML and RML: An ETL System-Based Overview.” In Proceedings of the 2nd International Workshop on Knowledge Graph Construction. http://ceur-ws.org/Vol-2873/paper1.pdf.

[Arenas-Guerrero22] Arenas-Guerrero, Julián, David Chaves-Fraga, Jhon Toledo, María S. Pérez, and Oscar Corcho. 2022. “Morph-KGC: Scalable Knowledge Graph Materialization with Mapping Partitions.” Semantic Web Preprint (Preprint): 1–20. https://doi.org/10.3233/SW-223135.

[Bartolomeo20] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan and J. Ott, “Oakestra: A lightweight hierarchical orchestration framework for edge computing”, USENIX Annual Technical Conference, 2023, pp. 215-231.

[Bennara20] Bennara, Mahdi, Antoine Zimmermann, Maxime Lefrançois, and Nurten Messalti. 2020. “Interoperability of Semantically-Enabled Web Services on the WoT: Challenges and Prospects.” In Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services, 149–53. https://doi.org/10.1145/3428757.3429199.

[Calbimonte20] J. P. Calbimonte, S. Martin, D. Calvaresi, N. Zappelaz and A. Cotting, A, "Semantic data models for hiking trail difficulty assessment", International Conference of Information and Communication Technologies in Tourism, January 2020, pp. 295-306.

[Chaves-Fraga19] Chaves-Fraga, David, Kemele M. Endris, Enrique Iglesias, Óscar Corcho, and Maria-Esther Vidal. 2019. “What Are the Parameters That Affect the Construction of a Knowledge Graph?” In On the Move to Meaningful Internet Systems: OTM 2019 Conferences - Confederated International Conferences: CoopIS, ODBASE, C&TC 2019, Rhodes, Greece, October 21-25, 2019, Proceedings, edited by Hervé Panetto, Christophe Debruyne, Martin Hepp, Dave Lewis, Claudio Agostino Ardagna, and Robert Meersman, 11877:695–713. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-030-33246-4_43.

[Chaves-Fraga20] Chaves-Fraga, David and others. 2020. “GTFS-Madrid-Bench: A Benchmark for Virtual Knowledge Graph Access in the Transport Domain.” Journal of Web Semantics 65: 100596. https://doi.org/10.1016/j.websem.2020.100596.

[Chortaras18] Chortaras, Alexandros, and Giorgos Stamou. 2018. “D2RML: Integrating Heterogeneous Data and Web Services into Custom RDF Graphs.” In Workshop on Linked Data on the Web Co-Located with The Web Conference 2018, LDOW@WWW 2018, Lyon, France April 23rd, 2018, edited by Tim Berners-Lee, Sarven Capadisli, Stefan Dietze, Aidan Hogan, Krzysztof Janowicz, and Jens Lehmann. Vol. 2073. CEUR Workshop Proceedings. CEUR-WS.org. http://ceur-ws.org/Vol-2073/article-07.pdf.

[Cimmino22] Cimmino, Andrea, and Raúl García-Castro. "Helio: a framework for implementing the life cycle of knowledge graphs." Semantic Web Preprint (2022): 1-27.

[Das12] Das, Souripriya, Seema Sundara, and Richard Cyganiak. 2012. “R2RML: RDB to RDF Mapping Language.” W3C Recommendation. W3C.

[Debruyne16] Debruyne, Christophe, and Declan O’Sullivan. 2016. “R2RML-F: Towards Sharing and Executing Domain Logic in R2RML Mappings.” In Proceedings of the Workshop on Linked Data on the Web, LDOW 2016, Co-Located with 25th International World Wide Web Conference (WWW 2016), edited by Sören Auer, Tim Berners-Lee, Christian Bizer, and Tom Heath. Vol. 1593. CEUR Workshop Proceedings. CEUR-WS.org. http://ceur-ws.org/Vol-1593/article-13.pdf.

[Desai15] P. Desai, A. Sheth and P. Anantharam, "Semantic Gateway as a Service Architecture for IoT Interoperability," IEEE International Conference on Mobile Services, 2015, pp. 313-319, doi: 10.1109/MobServ.2015.51.

[Dimou14] Dimou, Anastasia, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. “RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data.” In Proceedings of the Workshop on Linked Data on the Web Co-Located with the 23rd International World Wide Web Conference (WWW 2014). Vol. 1184. CEUR-WS.org. http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[García-González18] García-González, Herminio, Daniel Fernández-Álvarez, and José Emilio Labra Gayo. 2018. “ShExML: An Heterogeneous Data Mapping Language Based on ShEx.” In Proceedings of the EKAW 2018 Posters and Demonstrations Session Co-Located with 21st International Conference on Knowledge Engineering and Knowledge Management (EKAW 2018), Nancy, France, November 12-16, 2018, edited by Philipp Cimiano and Olivier Corby, 2262:9–12. CEUR Workshop Proceedings. CEUR-WS.org. http://ceur-ws.org/Vol-2262/ekaw-poster-08.pdf.

[Goethals20] T. Goethals, F. De Turck and B. Volckaert, “Extending kubernetes clusters to low-resource edge devices using virtual kubelets”, IEEE Transactions on Cloud Computing, Vol. 10(4), 2020, pp. 2623-2636.

[Grassi23] Grassi, Marco, Mario Scrocca, Alessio Carenini, Marco Comerio, and Irene Celino. n.d. “Composable Semantic Data Transformation Pipelines with Chimera.”

[Hohpe04] Hohpe, Gregor, and Bobby Woolf. 2004. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional.

[Iglesias20] Iglesias, Enrique and others. 2020. “SDM-RDFizer: An RML Interpreter for the Efficient Creation of Rdf Knowledge Graphs.” In Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 3039–46. https://doi.org/10.1145/3340531.3412881.

[Iglesias-Molina23] Iglesias-Molina, Ana, Dylan Van Assche, Julián Arenas-Guerrero, Ben De Meester, Christophe Debruyne, Samaneh Jozashoori, Pano Maria, Franck Michel, David Chaves-Fraga, and Anastasia Dimou. 2023. “The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF.” In The Semantic Web – ISWC 2023, edited by Terry R. Payne, Valentina Presutti, Guilin Qi, María Poveda-Villalón, Giorgos Stoilos, Laura Hollink, Zoi Kaoudi, Gong Cheng, and Juanzi Li, 152–75. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-47243-5_9.

[Janowicz19] K. Janowicz, A. Haller, S. J. Cox, S. D. Le Phuoc and M. Lefrançois, “SOSA: A lightweight ontology for sensors, observations, samples, and actuators”, Journal of Web Semantics, 2019, Vol. 56, pp. 1-10.

[Klímek16] Klímek, Jakub, Petr Škoda, and Martin Nečaskỳ. 2016. “LinkedPipes ETL: Evolved Linked Data Preparation.” In European Semantic Web Conference, 95–100. Springer.

[Knap14] Knap, Tomáš, Maria Kukhar, Bohuslav Macháč, Petr Škoda, Jiří Tomeš, and Ján Vojt. 2014. “UnifiedViews: An ETL Framework for Sustainable RDF Data Processing.” In European Semantic Web Conference, 379–83. Springer.

[Kubitza17], T. Kubitza and A. Schmidt, “meSchup: A platform for programming interconnected smart things”, Computer, 2017, Vol. 50(11), pp. 38-49.

[Lefrançois16] Lefrançois, Maxime, Antoine Zimmermann, and Noorani Bakerally. 2016. “Flexible RDF Generation from RDF and Heterogeneous Data Sources with SPARQL-Generate.” In Knowledge Engineering and Knowledge Management - EKAW 2016 Satellite Events, EKM and Drift-an-LOD, Bologna, Italy, November 19-23, 2016, Revised Selected Papers, edited by Paolo Ciancarini, Francesco Poggi, Matthew Horridge, Jun Zhao, Tudor Groza, Mari Carmen Suárez-Figueroa, Mathieu d’Aquin, and Valentina Presutti, 10180:131–35. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-319-58694-6_16.

[Lucky16] M. N. Lucky, M. Cremaschi, B. Lodigiani, A. Menolascina and F. De Paoli, "Enriching API descriptions by adding API profiles through semantic annotation", International Conference on Service-Oriented Computing, October 10-13, 2016, pp. 780-794.

[Meester17] Meester, Ben De, Wouter Maroy, Anastasia Dimou, Ruben Verborgh, and Erik Mannens. 2017. “RML and FnO: Shaping DBpedia Declaratively.” In The Semantic Web: ESWC 2017 Satellite Events, 10577:172–77. Springer. https://doi.org/10.1007/978-3-319-70407-4_32.

[Michel15] Michel, Franck, Loïc Djimenou, Catherine Faron-Zucker, and Johan Montagnat. 2015. “Translation of Relational and Non-Relational Databases into RDF with xR2RML.” In WEBIST 2015 - Proceedings of the 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal, 20-22 May, 2015, edited by Valérie Monfort, Karl-Heinz Krempels, Tim A. Majchrzak, and Ziga Turk, 443–54. SciTePress. https://doi.org/10.5220/0005448304430454.

[Mortazavi17] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips and E. De Lara, "Cloudpath: A multi-tier cloud computing framework", IEEE/ACM Symposium On Edge Computing, October 2017, pp. 1-13.

[Naiema23] Hamed, Naeima, et al. "FOO: An upper-level ontology for the Forest Observatory." European Semantic Web Conference. Cham: Springer Nature Switzerland, 2023.

[Nguyen16] K. K. Nguyen and M. Cheriet, "Virtual edge-based smart community network management" IEEE Internet Computing, 2016, Vol. 20(6), pp. 32-41.

[Recker06] J. Recker and J. Mendling, “On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages”, In Proceedings of the Workshops and Doctoral Consortium, Presses universitaires de Namur (Namur University Press), 2006, pp. 521-532.

[Sadeghi20] Sadeghi, Mersedeh, Petr Buchníček, Alessio Carenini, Oscar Corcho, Stefanos Gogos, Matteo Rossi, Riccardo Santoro, and others. 2020. “SPRINT: Semantics for PerfoRmant and Scalable INteroperability of Multimodal Transport.” In 8th Transport Research Arena TRA 2020, 1–10. http://hdl.handle.net/11311/1132635.

[Sajjad16] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers", IEEE/ACM Symposium on Edge Computing, October 2016, pp. 168-178.

[Saurez21] E. Saurez, H. Gupta, A. Daglis and U. Ramachandran, “Oneedge: An efficient control plane for geo-distributed infrastructures“, ACM Symposium on Cloud Computing, November 2021, pp. 182-196.

[Scrocca21] Scrocca, Mario, Alessio Carenini, Marco Comerio, and Irene Celino. 2021. “Semantic Conversion of Transport Data Adopting Declarative Mappings: An Evaluation of Performance and Scalability.” In Proceedings of the 3rd International Workshop Semantics And The Web For Transport, edited by David Chaves-Fraga, Pieter Colpaert, Mersedeh Sadeghi, Mario Scrocca, and Marco Comerio. Vol. 2939. CEUR Workshop Proceedings. Online, September: CEUR. https://ceur-ws.org/Vol-2939/#paper2.

[Scrocca20] Scrocca, Mario, Marco Comerio, Alessio Carenini, and Irene Celino. 2020. “Turning Transport Data to Comply with EU Standards While Enabling a Multimodal Transport Knowledge Graph.” In Proceedings of the 19th International Semantic Web Conference, 12507:411–29. Springer. https://doi.org/10.1007/978-3-030-62466-8_26.

[Slepicka15] Slepicka, Jason, Chengye Yin, Pedro Szekely, and Craig Knoblock. 2015. “KR2RML: An Alternative Interpretation of R2RML for Heterogenous Sources.” In Proceedings of the 6th International Workshop on Consuming Linked Data, edited by Olaf Hartig, Juan Sequeda, and Aidan Hogan. Vol. 1426. CEUR Workshop Proceedings. Bethlehem, Pennsylvania: CEUR. https://ceur-ws.org/Vol-1426/#paper-08.

[Thuluva17] A. S. Thuluva, A. Bröring, Medagoda, G.P., Don, H, D. Anicic and J. Seeger, "Recipes for IoT applications", Proceedings of the Seventh International Conference on the Internet of Things, 2017, pp. 1-8.

[Thuluva20] A. S. Thuluva, D. Anicic, S. Rudolph and M. Adikari, "Semantic Node-RED for rapid development of interoperable industrial IoT applications", Semantic Web, Vol. 11(6), 2020, pp. 949-975.

[VanAssche22] Van Assche, Dylan, Thomas Delva, Gerald Haesendonck, Pieter Heyvaert, Ben De Meester, and Anastasia Dimou. 2022. “Declarative RDF Graph Generation from Heterogeneous (Semi-)Structured Data: A Systematic Literature Review.” Journal of Web Semantics, 100753.

[Vetere05] Vetere, G., and M. Lenzerini. 2005. “Models for Semantic Interoperability in Service-Oriented Architectures.” IBM Systems Journal 44 (4): 887–903. https://doi.org/10.1147/sj.444.0887.

[Viola16] F. Viola, A. D'Elia, D. Korzun, I. Galov, A. Kashevnik and S. Balandin, "The M3 architecture for smart spaces: Overview of semantic information broker implementations," Conference of Open Innovations Association (FRUCT), 2016, pp. 264-272, doi: 10.23919/FRUCT.2016.7892210.

[Vu19] Vu, Binh, Jay Pujara, and Craig A. Knoblock. 2019. “D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF.” In Proceedings of the 10th International Conference on Knowledge Capture, 189–96. K-CAP ’19. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3360901.3364449.

[Xiong18] Y. Xiong, Y. Sun, L. Xing and Y. Huang, “Extend cloud to edge with kubeedge”, IEEE/ACM Symposium On Edge Computing, October 2018, pp. 373-377.

[Zhang19] D. Zhang, T. Rashid, X. Li, N. Vance and D. Wang, "Heteroedge: taming the heterogeneity of edge computing system in social sensing", International Conference on Internet of Things Design and Implementation, April 2019, pp. 37-48.