SEMIoTICS

Deliverable D3.10
Network-level Semantic Interoperability (final)

Deliverable release date	28.02.2020
Authors	1. Ermin Sakic, Arne Bröring (SAG) 2. Jordi Serra, Luis Sanabria-Russo, David Pubill, Angelos Antonopoulos, Christos Verikoukis (CTTC) 3. Konstantinos Fysarakis, Michalis Smyrlis, Emmanouil Chatzimpyrros, Konstantina Koloutsou (STS) 4. Prodromos-Vasileios Mekikis (IQU)
Responsible person	Konstantinos Fysarakis (STS)
Reviewed by	Ermin Sakic (SAG), Nikolaos Petroulakis (FORTH)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Verikoukis Christos) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

Acronyms Table

Acronym	Definition
AAA	Authentication, Authorisation and Accounting
API	Application Programmable Interface
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
ASCII	American Standard Code for Information Interchange
BSS	Business Support System
BAN	Body Area Network
BLE	Bluetooth Low Energy
CAN	Controller Area Network
CB	Context Broker
CP	Context Producer
CC	Context Consumer
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
DTLS	Datagram Transport Layer Security
ETSI	European Telecommunications Standards Institute
E2E	End to End
GRE	Generic Routing Encapsulation
HTTP	HyperText Transfer Protocol
IoT	Internet of Things
IIoT	Industrial Internet of Things
IT	Information Technology
IHES	Intelligent Heterogeneous Embedded Sensors
IETF	Internet Engineering Task Force
IPC	Inter-Process Communication
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IFTTT	If This, Then That
JSON	JavaScript Object Notation
LwM2M	Lightweight Machine to Machine
6LoWPAN	IPv6 over Low-power Wireless Personal Area Network
LHS	Left Hand Side
MAC	Media Access Control
MQTT	Message Queuing Telemetry Transport
MQTT-SN	Message Queuing Telemetry Transport – For Sensor Networks
MANO	Management and Orchestration
M2M	Machine to Machine
NETCONF	Network Configuration Protocol
NBI	Northbound Interface
NFV	Network Functions Virtualization
NFVO	NFV Orchestrator
NFVI	Network Functions Virtualization Infrastructure
NS	Network Service
OASIS	Organization for the Advancement of Structured Information Standards
OFCONF	OpenFlow Configuration
OVSDB	Open vSwitch Database Management Protocol
OSS	Operations Supports System
OGC	Open Geospatial Consortium

QoS	Quality of Service
OEM	Original Equipment Manufacturer
OWL	Web Ontology Language
PNF	Physical Network Functions
PLC	Programmable Logic Controller
RA	Robotic Assistant
REST	Representational State Transfer
RO NBI	Resource Orchestrator Northbound Interface
RR	Robotic Rollator
RPC	Remote Procedure Call
RHS	Right Hand Size
SARA	Socially Assistive Robotic Solution for Mild Cognitive Impairment or mild Alzheimer's Disease
SAWSDL	Semantic Annotations for WSDL and xml schema
SASL	Simple Authentication and Security Layer
SBI	Southbound Interface
SCADA	Supervisory Control and Data Acquisition
SDN	Software-Defined Networking
SEMIoTICS	Smart End-to-end Massive IoT Interoperability, Connectivity and Security
SSC	SEMIoTICS SDN Controller
SPDI	Security, Privacy, Dependability, and Interoperability
SE	Smart Environment
SSWAP	Simple Semantic Web Architecture and Protocol
TD	Thing Description
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UC	Use Case
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VIM	Virtualized Infrastructure Manager
VLAN	Virtual Local Area Network
VXLAN	Virtual Extensible Local Area Network
VM	Virtual Machine
VNF	Virtual Network Function
VNF-FG	Virtual Network Function-Forwarding Graphs
VTN	Virtual Tenant Networks
W3C	World Wide Web Consortium
WoT	Web of Things
WSDL	Web Services Description Language
WSMO	Web Service Modeling Ontology
WS-BPEL	Web Services Business Process Execution Language
XML	Extensible Markup Language
XSD	XML Schema Definition

1 INTRODUCTION

This deliverable is the final output of Task 3.4 ("Network-level Semantic Interoperability"), providing an update on deliverable D3.4 - "Network-level Semantic Interoperability (first draft)" and targeting the third objective of WP3 *("To develop and offer adaptable and dynamic networking services to client IoT applications")*.

As such, it provides the final design and specification of the network programming interfaces that enables the development, optimization and adaptation properties required for the SEMIoTICS framework to support the deployment of network services from all SEMIoTICS layers and its seamless interaction with IoT Applications, as specified by SPDI patterns. As detailed herein, in SEMIoTICS SDN interfaces can be utilized from various levels in the IoT implementation stack (IT & Cloud infrastructures, IoT platforms, the SEMIoTICS framework and IoT applications) for the provisioning of adaptable and dynamic networking services. The pattern-based SEMIoTICS approach is adopted here as well, to specify the use of network services and define specific SPDI properties. Finally, in the context of this task, the adopted key enabling technologies are extensively described to provide an overview of the pertinent technological landscape.

In addition to the State-of-the-Art Analysis and the requirements identified during WP2 activities that where essential on guiding the efforts of this Task 3.4, it is also important to mention the interplay with other WP3 activities, such as the ones taking place in the context of Task 3.1 ("Software defined Aggregation, Orchestration and Cloud Networks") and T3.2 ("IIoT Network Function Virtualization"), since compatibility with (and requirements of) SPDI-driven Task 3.4 interfaces are considered in the first phases of the work carried out in these tasks. Moreover, the thing descriptions and semantic schemas defined in Task 3.3 ("Semantics-based Bootstrapping & Interfacing") are influenced by Task 3.4, since said thing descriptions must also include network-level capabilities and semantic as well as SPDI information of the involved entities. Considering the use of SPDI patterns, there is significant interaction with WP4, and more specifically the pattern language and associated patterns defined in Task 4.1 ("Architectural SPDI Patterns"), as these are driving the network-level semantic capabilities of the interfaces described herein. Moreover, the semantic annotations adopted in Task 3.4 are integral in the work carried out in Task 4.4 ("End-to-End Semantic Interoperability").

From the perspective of the delta compared to the previous version of the deliverable (i.e. D3.4 - "Network-level Semantic Interoperability (first draft)"), the contributions in D3.10 can be summarized as follows:

- Provides the final design and specification of the SEMIoTICS SPDI-driven NBI (see subsections 4.1 and 4.2, respectively).
- Provides final implementation details and testing results (see subsection 4.3)
- Provides full set of Interoperability-focused patterns (see section 5)
- Highlights the use of the Pattern-driven NBI as an enabler for key SEMIoTICS features (see section 6), including role in end-to-end (E2E) semantic interoperability features, definition of IoT Orchestrations and use by external IoT platforms

To address the above issues, the deliverable is structured as follows. Section 2 lists some key aspects that the network interfaces will need to support across all different IoT application/services levels (IT & Cloud infrastructures, IoT platforms, the SEMIoTICS framework, and IoT applications), also considering complex types of interactions that may introduce additional requirements (e.g., Cross-platform, Cross-layer, Cross-application and Higher-level services). Section 3 describes the key enabling technologies considered as well as the subset of those finally adopted to implement the network-level semantic interoperability. Section 4 features the specification of the pattern-driven Network Services API, the development of the described interfaces and the testing results. Section 5 provides the full set of interoperability patterns, while section 6 positions the work presented herein in the context of the SEMIoTICS framework and its key features. Finally, Chapter 7 provides the concluding remarks.

1.1 PERT chart of SEMIoTICS

The role of Task 3.4 is shown in the PERT chart of the project (Figure 1).

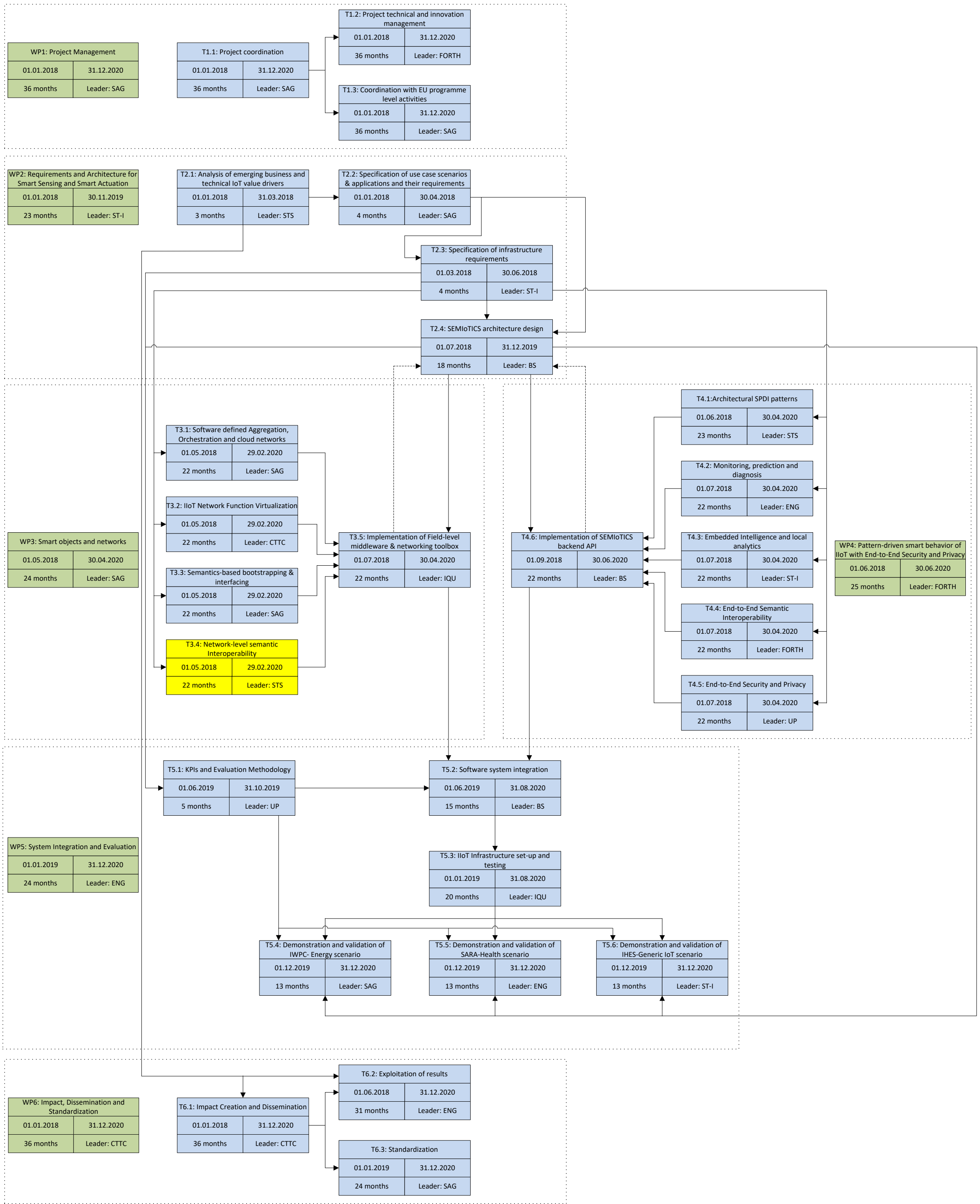


FIGURE 1. TASK 3.4 WITHIN THE SEMIOTICS PERT

2 NETWORK INTERFACING CONSIDERATIONS AND REQUIREMENTS

This section aims to document the key elements and considerations that drove the design and specification of the SPDI-driven network programming interfaces of SEMIoTICS, to enable the deployment of network services from all SEMIoTICS layers and the seamless interaction of the framework with IoT applications.

To identify the relevant requirements for all cases/layers, we also consider complex types of interactions that may introduce additional requirements, such as:

- **Cross-layer**, whereby entities deployed at different layers of the SEMIoTICS framework interface with each other, thus allowing interactions across non-adjacent layers, such as cloud to edge or application to network.
- **Cross-platform**, whereby applications or services access resources from multiple platforms through the common interfaces. This covers requests to different instances of the SEMIoTICS platform and/or 3rd party IoT platforms; effectively providing the means to an application deployed on one platform (e.g., an IIoT wind turbine status monitoring application aggregating information from pertinent sensors) to collect data from other platforms that process related data.
- **Cross-application domain**, with applications or services accessing now information not only from several platforms, but also from platforms that process data from different application domains. Therefore, such an application could potentially collect data about environmental conditions and traffic from a smart city application, in order to propose the least polluted routes to patients with breathing issues covered under a smart healthcare application.
- **Higher level services**, whereby exposed interfaces enable higher level services to orchestrate existing deployments, applications, and the associated services, to provide value-added services, such as providing wind turbine failure predictions or energy demand predictions (to fine-tune energy output) from data aggregated across associated services, enabling effective predictions even for stakeholders/deployments that do not have the breadth of historical data or computational capabilities to extract this knowledge.

To enable the above, two basic properties have to be guaranteed across the deployment, also affecting the design of the networking interfaces:

- **Platform-scale independence**, allowing the integration of resources from platforms at different scale. More specifically, at the Cloud/IoT backend level, platforms can host high volumes of data from a large number of devices. In contrast, field-level deployments (e.g., fog) interact with nearby devices in the field and only maintain limited amount of information. Device level platforms (e.g., at the IoT gateway level) have direct communication with the things, managing heterogeneous data. As a result, in the SEMIoTICS framework, an application should be able to uniformly aggregate information for the different scale platforms (e.g., collect wind turbine status values for a specific area via cloud or minimally processed data via a platform at field).
- **Platform independence**, allowing the integration of distinct platforms that implement the same functionality, like an IIoT wind turbine status monitoring in different wind parks. The platforms may utilize different equipment and techniques in order to monitor the wind turbines (e.g., legacy wired sensors attached to smart gateway or newer wireless sensors); a single application at the backend should be able to interface with all instances in a uniform manner without requiring any changes.

The vision of such a heterogeneous and flexible deployment is sketched in Figure 2, where different applications in an IoT marketplace are interacting with IoT devices via a common API that support various operations, such as discovery, access etc. While driven by the above, in the subsections below more specific requirements are investigated, focusing on particular layers and types of interactions.

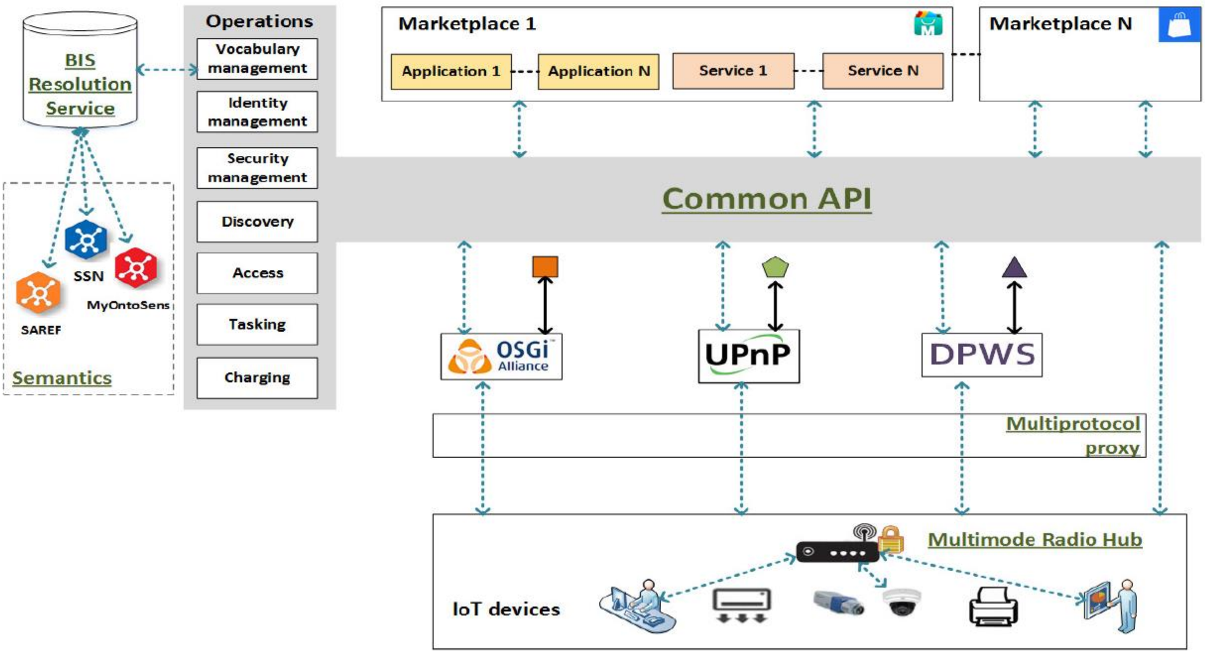


FIGURE 2: INTEROPERABILITY VISION ACROSS ALL 4 LEVELS, INSPIRED BY [2]

2.1 IT & Cloud Infrastructures interfacing

Taking the most advantage out of the physical infrastructure requires the ability of sharing its resources in a secure and dependable way. This is achieved via virtualization, where former physical components such as network devices (e.g., switches or routers), i.e. Physical Network Functions (PNFs), are replaced by software counterparts (i.e. Virtual Network Functions (VNF)) that provide the same functionality on top of the required isolation properties for guaranteeing security and privacy among different tenants/applications.

In this domain, ETSI NFV is part of the European Telecommunication Standards Institute (ETSI), an independent standardization organization that develops NFV standards and proofs-of-concepts, see e.g., [1][3]. Within ETSI's Network Functions Virtualization (NFV) paradigm, the task of abstracting the infrastructure's hardware and its exposure as virtual resources are tasks assigned to the Virtualized Infrastructure Manager (VIM). Such an entity ensures that appropriate network overlays, compute and storage resources are configured according to predefined configurations (descriptors) containing specifics of each VNF and how to interconnect them together to realize a Network Service (NS).

Applications requiring a NS should send requests to the NFV orchestrator the backend/Cloud level. This entity exposes HTTP RESTful APIs via the so-called endpoints, which trigger the allocation of virtual resources from the VIM, as well as SDN Controllers (if any). Particular modifications to components of existing NS are also possible. VNF management tasks such as: start, stop, resume, pause, snapshots are achievable through VIM endpoints tailored to managing compute resources. Moreover, physical or virtual network-related settings can also be modified using similar endpoints at the VIM or the SDN Controller; the same is true for block storage allocations for VNFs.

2.1.1 MANAGEMENT AND ORCHESTRATION

Within ETSI's NFV and SEMIoTICS, the role of the Management and Orchestration (MANO) controller is to provide a higher level of abstraction for the deployment of NS. Such abstraction is achieved by exchanging information about the physical/virtual infrastructure (from here on referred to as Network Functions Virtualization Infrastructure (NVFI)) with the VIM and maintaining a catalogue of VNF and NS descriptors. The associated standards (e.g.,[1]) document some pertinent network related considerations that arise from these standards, such as the need for portability, service continuity, and operational and management requirements.

Based on the above, in the subsections below, we provide descriptions about the information exchange endpoints between the VIM and MANO, and a description of how applications could trigger the deployment of VNFs or NS via the exposed northbound APIs.

2.1.1.1 NFVI ENDPOINTS FOR ORCHESTRATION

Figure 3 shows the reference NFV architectural framework as defined by ETSI. The figure groups VIM and MANO into *NFV Management and Orchestration*. It also defines a considerable set of reference points for information exchange among components. Most relevant for the deployment of NS within SEMIoTICS are the *Nf-Vi*, *Os-Ma-Nfvo*, and *Or-Vi* reference points[3].

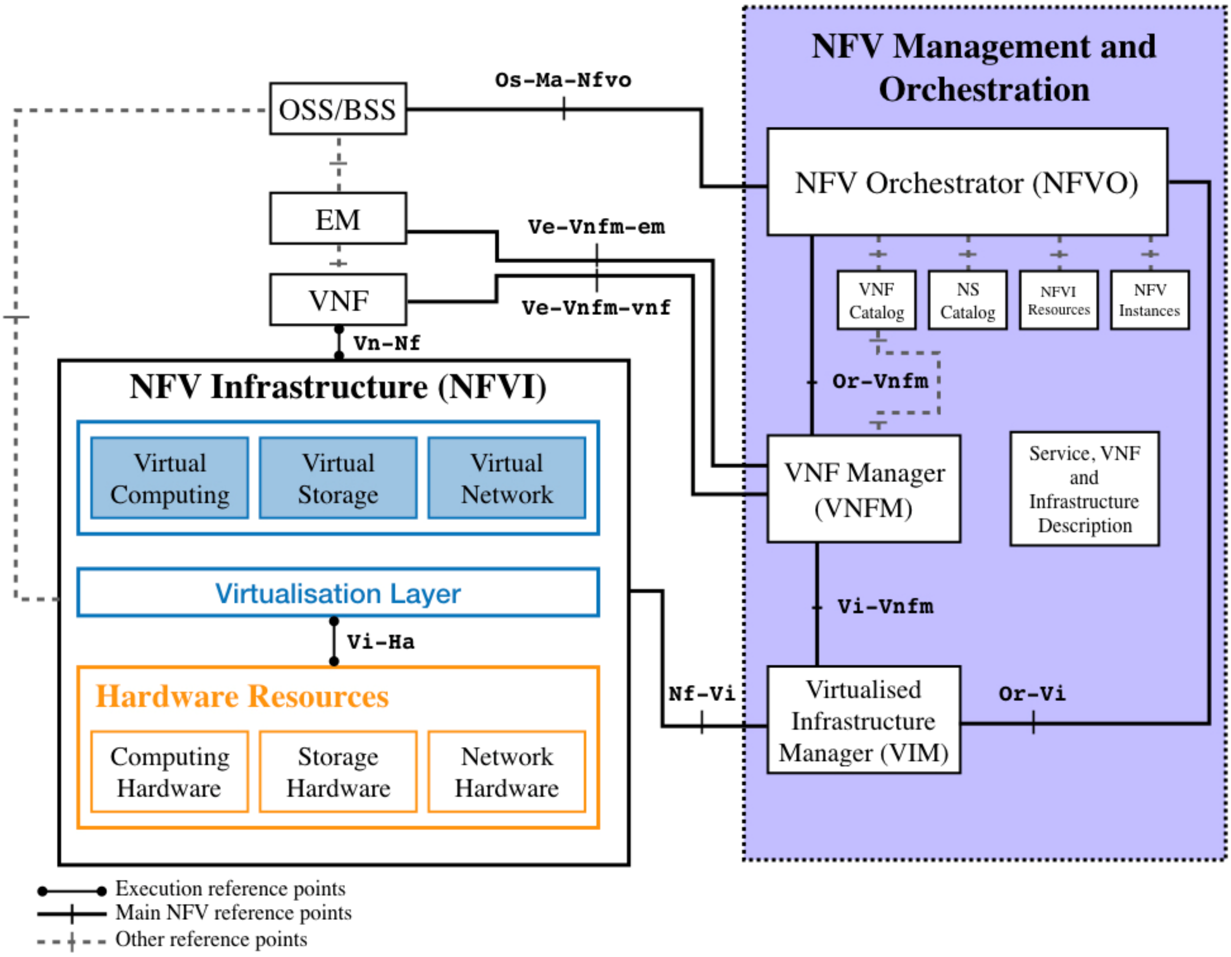


FIGURE 3. NFV REFERENCE ARCHITECTURAL FRAMEWORK

In more detail, these are:

- **Nf-Vi:**
  This reference point is used for NFVI-VIM communication. Particularly:
  - Assignment of virtualized resources after an allocation request.
  - Forwarding of virtualized resources state information.
  - Hardware resources configuration, information exchange and events capture.
- **Os-Ma-Nfvo**
  It realizes Operations Support System/Business Support System (OSS/BSS)-NFV Management and Orchestration communication. It is used for:
  - Request for network service lifecycle management.

  - Requests for VNF lifecycle management.
  - Forwarding of NFV related state information.
  - Policy management exchanges.
  - Data analytics exchanges.
  - Forwarding of NFV related accounting and usage records.
  - NFVI capacity and inventory information exchanges.

  It is valid to assume the use of this reference point to software other than OSS/BSS. That is, any authorized software external to NFV could use this reference point for gathering information of the physical/virtualized infrastructure, as well as signalling the intention to create a NS via the MANO controller.
- **Or-Vi**
  Orchestrator-VIM communication reference point. It is used for:
  - Resource reservation and/or allocation requests by the Orchestrator.
  - Virtualized hardware resource configuration and state information exchange.

  As mentioned before, the *Os-Ma-Nfvo* reference point can be used by OSS/BSS (or other entity such as SEMIoTICS global pattern orchestrator) to gather information of the NFVI and trigger the creation/modification of a NS; but the *Or-Vi* reference point is the one that enables direct communication between MANO and VIM in order to realize such service by allocating resources from the infrastructure.

2.1.1.2 INTERFACES FOR NS MANAGEMENT

From SEMIoTICS Pattern orchestrator's perspective, it is envisioned that the creation and modification of NS could be achieved through the MANO controller Resource Orchestrator northbound interface (RO NBI) [4]. In the global NFV architectural framework this NBI is reached via the *Os-Ma-Nfvo* endpoint.

Applications or any pattern enforcement entity should trigger such HTTP RESTful APIs in order to gather information or act upon the configuration of a new or existing NS. Some of the entities upon which modifications are possible through this NBI are:

- Tenant or applications.
- Gathering information from several VIMs. Notice that the MANO is able to handle many different VIMs, so orchestration of NS across different domains is possible.
- VNFs.
- VNF-FG (VNF-Forwarding Graphs) and topologies. Notice that NS and VNF-FG are two sides of the same concept, the former is an application's perspective, while the latter specifies the actual interconnection of VNFs.
- NS instances. This relates to the re-instantiation or termination of NS already onboarded on the NFVO catalogue.

2.2 IoT Platforms interfacing

SEMIoTICS aims to offer federation and interoperability with other IoT platforms, most notably. IoT platforms generally rely on message-oriented middleware technologies, including reliable message queuing and publish/subscribe messaging. These "brokered" messaging capabilities can be thought of as decoupled messaging features that support publish/subscribe and message decoupling, where clients and servers can connect and perform their operations in an asynchronous fashion. At the core of all IoT platforms we generally find a Message (or Event) Broker which is able to mediate between content producers (e.g., sensors) and the context consumer applications (e.g., Smartphone applications visualizing of the context information provided by the sensors), offering Publish/Subscribe functionality. Publish/subscribe pattern provides a one-to-many form of communication via topics and subscriptions, where each published message is made available to each subscription registered with the topic. Messages are sent to a topic and delivered to one or more associated subscriptions, depending on filter rules that can be set on a per-subscription basis. The subscriptions can use additional filters to restrict the messages that they want to receive. Messages are sent to a topic but are not received from the topic directly, instead subscribers to the copy receive copies from the message. In what follows, we summarize the main actors of an IoT platform:

- **Publish/Subscribe Context Broker**. As already mentioned, the Publish/Subscribe Context Broker (CB) is the main component of an IoT platform. It works as a handler and aggregator of context data and as an interface between architecture actors. Primarily the CB has to control context flow among all attached actors; in order to do that, the CB has to know every Context Provider (CP) in the architecture; this feature is done through an announcement process detailed in the next sections. Typically, the CB also provides a Context Provider Lookup Service and a Context Persistence Service.
- **Context Producer**. A Context Producer (CP) is an actor (e.g., a temperature sensor) able to generate context. The basic Context Producer is the one that spontaneously updates context information, about one or more context attributes according to its internal logic. This communication is between CS and CB is in push mode, from the CP to the CB.
- **Context Consumer.** A Context Consumer (CC) is an entity (e.g., a context-based application) that exploits context information. A CC can retrieve context information sending a request to the CB or invoking directly a CP over a specific interface. Another way for the CC to obtain information is by subscribing to context information updates that match certain conditions (e.g., are related to certain set of entities). The CC registers a call-back operation with the subscription for the purpose, so the CB notifies the CC about relevant updates on the context by invoking this call-back function.

In FIWARE (see Figure 4), the Orion Context Broker fulfils the pub/sub Message Broker functionality and must be federated with SEMIoTICS. FIWARE leverages the NGSIv2 Data Model and API, which relies on JSON representation to make data from multiple providers accessible for data consumers. The interaction with both data providers and data consumers is taking place via the FIWARE NGSI 10 context data API [5]. SEMIoTICS must leverage the API for context queries, context subscription and context updates to interact with the respective context elements (i.e., sensors and actuators) in a FIWARE domain.

On the contrary, for FIWARE to access context elements in other domains (in this case SEMIoTICS) a specialized FIWARE entity, namely the ***Context Provider,*** must be involved. The latter can be registered via its URL as the source of context information for specific entities and attributes included in that registration, using the ORION NGSIv1 and NGSIv2 APIs. In the case of NGSIv2 Data Model, which uses JSON representation, this is provided by the field **provider**:

```
"provider":{
  "http": {
    "url": "http://mysensors.com/Rooms"
  }
}
```

If FIWARE Orion fails to find a context element locally (i.e. in its internal database) for a query or update operation but a Context Provider is registered for that context element, then it will forward the query or update request to the respective Provider. In this case, Orion acts as proxy, while the client that issues the request, the process is transparent. SEMIoTICS must implement the respective NGSI10 API (at least partially) to support query/update operations from FIWARE to a context element in the SEMIoTICS domain.

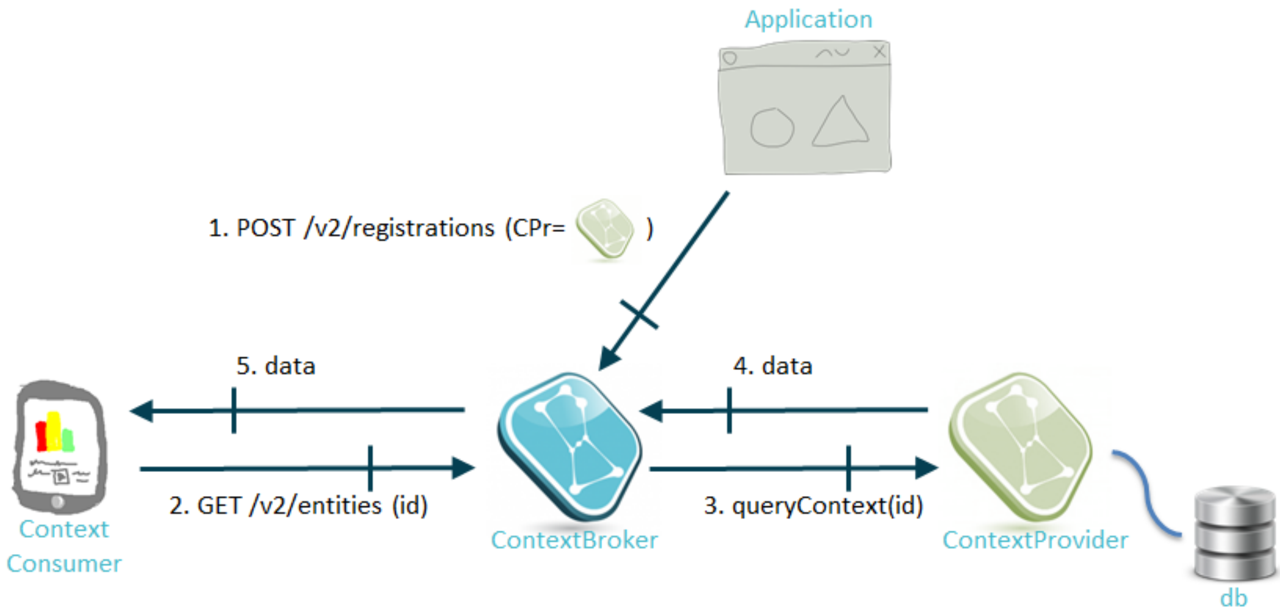


FIGURE 4: FEDERATION WITH FIWARE IOT PLATFORM

2.3 Network-level Interfacing with SEMIoTICS framework

The network level interfacing requirements consider the intricacies of the different envisioned application domains in order to offer a flexible and adaptable network infrastructure within the SEMIoTICS framework. Therefore, each scenarios' requirements are explicitly examined; energy, IIoT integration in wind park control network (D2.2 subsection 2.1); healthcare, socially assistive robotic solution for ambient assisted living (SARA) (D2.2 subsection 2.2); smart cities, Intelligent Heterogeneous Embedded Sensors for future IoT systems (IHES) (D2.2 subsection 2.3). Some important considerations in this context are presented in the subsections below.

2.3.1 IIOT INTEGRATION IN WIND PARK CONTROL NETWORK

In the IIoT SEMIoTICS deployment in the wind park control network the pattern language will have to consider simple interactions between the field devices deployed in the wind park (i.e. the IIoT ecosystem); cross layer interactions with the network management system, enabled via an SDN/NFV capable control network; finally, cross layer interactions with the backend/cloud platforms. Additional complex interactions are explored in section 2.3.4 below, while a high level view of these are depicted in Figure 5.

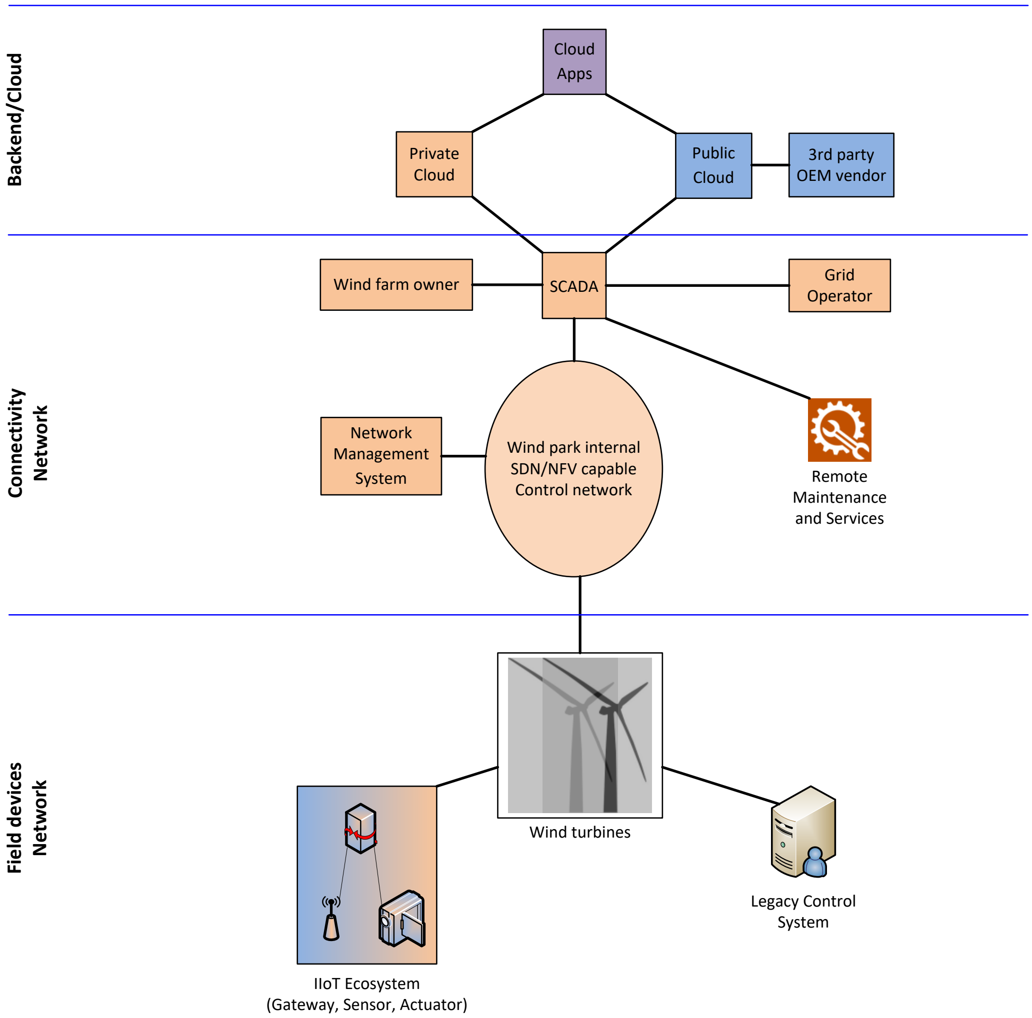


FIGURE 5. INTERACTIONS BETWEEN STAKEHOLDERS AT EACH LAYER IN THE IIOT WIND PARK SCENARIO

Examining closer the field level interactions, we observe the interactions between devices in the IIoT Ecosystem; IIoT gateway communicates with the IIoT sensor and, in turn, with the IIoT actuator, to accomplish that, first a simple but robust (i.e. reliable and secure) registration operation must be completed to pair the IIoT sensor with the IIoT gateway, a procedure known as commissioning. Then, the data collected by the sensor can be processed (features are extracted from data) by the IIoT gateway or relayed directly to the actuator to act accordingly.

In times where the collection of requests from the connected sensors and actuators exhaust the computational resources of the current IIoT gateway, the gateway is expected to *communicate* with the NFV Orchestrator (NFVO) so that the later will consecutively deploy (or migrate) the necessary VMs, allowing the gateway to offload computations to the private cloud, where computational resources are abundant. So, if the gateway suggests (via the analytics algorithm) that a certain action has to be taken, it *sends* this decision to the control center without sending high-frequency and large amount of data.

It's also very important to consider the scalability of the NFVO and SDN controller so that it can keep up with the requests received by the gateways and to be able to provide the integration of the necessary IoT components. For the scaling to function properly, QoS network-related criteria should be identified on application-level (e.g., latency between motion and motion input) and be able to determine if the current network configuration is sufficient. If not, then these criteria are automatically translated to an improved configuration by the SDN controller. Moreover, service quality specific -properties, such as reliability, delay and bandwidth, must be guaranteed by the SDN controller. Therefore, QoS measures such as isolation through VLANs, traffic prioritization, and bandwidth allocation should be supported by the network and the interfaces should be in place to allow for such QoS specific negotiations between involved entities.

Although SDN controllers work excellent in this scenario, there is always room for error or malicious misuse. For example, a malicious entity could launch a denial service type of attack, packet-in flooding, in which the SDN control plane is flooded with **packet_In_messages** potentially making the SDN controller to waste all available resources to process these messages, thus becoming unreachable or ending up in an unpredictable state [29][30][31] (additional attacks for SDN are thoroughly explored in [32]. Such cases should be swiftly identified and excluded from the network; thus, the SDN controller must accommodate such security related services and facilitate their interactions and adaptation actions (e.g., rerouting/isolating malicious traffic to a honeypot system).

2.3.2 SOCIALLY ASSISTIVE ROBOTIC SOLUTION FOR AMBIENT ASSISTED LIVING (SARA)

In the SARA deployment scenario, pattern-based mechanisms will ensure that the various communication between different stakeholders will occur in a reliable and secure manner (further explored in D4.1). Initially, the pattern engine needs to setup up those connections by employing network-level interoperability operations. Considerations for the following communications should be met. The SS (Smart Environment) hub should be able to act as a gateway for Internet access for all four hubs (BAN, RR, RA, SE) and also provide sufficient communication resources between them via ZigBee technology; the smart wearable device within BAN (Body Area Network) should be connected to the BAN hub (i.e. smartphone) via Bluetooth Low Energy (BLE) technology; the BAN hub is connected to the internet via cellular connectivity (i.e. possibly 5G in the future) and should also be able to engage communication with devices within RR (Robotic Rollator) via BLE or WIFI; devices within RR use a Controller Area Network bus architecture (CAN-bus) to exchange information; finally, cross layer interactions between the SARA backend services and the SARA IoT field devices and cross platform interactions between the SARA backend services and the SARA client applications should be accommodated and realized by internet protocols (IPv4/IPv6). The pattern language will resolve those issues by defining and enforcing mechanisms that will guarantee the proper network interfaces are present and functional to achieve those communications.

2.3.3 INTELLIGENT HETEROGENEOUS EMBEDDED SENSORS FOR FUTURE IOT SYSTEMS (IHES)

In the horizontal scenario for IHES, the pattern language needs to support interactions between various components. We describe some interactions in the following paragraph.

For the IIoT Sensing unit, network-level interoperability mechanisms should ensure the communication with the IIoT gateway to notify it for any changes and send any sensor measurement data; the initialization of low latency and reliable communication (further explored in D4.1) between the controllers, when they are deployed at the IoT gateways must be arranged. Low latency and reliable communication could also be needed in some cases (e.g., in a time sensitive scenario where a sensor monitors a safety parameter and must promptly trigger an actuator to avoid life threating incident, such as a wind turbine with very high inclination or a smart vehicle collision or a patient fall)

it's important to receive a measurement from a sensor in time or activate an actuator); cross layer interactions are explored in subsection 2.3.4.2. The network-level interoperability mechanisms need to also deal with those concerns.

2.3.4 COMPLEX TYPES OF INTERACTIONS

Some additional types of more complex interactions are explored in the subsections below.

2.3.4.1 CROSS-PLATFORM

Cross-platform interactions between various components in different SEMIoTICS instances in addition with interactions with corresponding entities in other platforms (e.g., FIWARE or MindSphere) should be supported. For the **Wind Park scenario**, we can identify such interactions on the Backend/Cloud layer (e.g., Cloud applications exchanging data with the private/public cloud).

Considering the **SARA scenario,** the SEMIoTICS framework should be able to communicate with the AREAS Business Framework to access various features such as the AREA suite (e.g., Patient Health Record) and its corresponding management services (e.g., Identity & Access Management, Storage).

For the **IHES scenario**, and in all the applications built on top, we can foresee cross platform interactions of the SEMIoTICS Backend with Open IoT Platforms (e.g., FIWARE) and domain specific IoT platforms (e.g., MindSphere), providing seamless interactions across heterogeneous devices via the Semantic Mediator components and associated interfaces which will be present on the SEMIoTICS field gateway, as well as its backend.

2.3.4.2 CROSS-LAYER

Cross layer interactions are possible when data is exchanged between components of a different layer (e.g., field devices to cloud).

For the **Wind Park scenario**, we need to consider the IIoT ecosystem's capability to exchange information with the cloud, as data coming from the edge, through the IIoT gateway, to the cloud apps. Additionally, the communication of the IIoT gateway with the backend must be ensured, as the former will need to send a request whether the current computing environment doesn't cover its needs (e.g., the combined requests from connected sensors and actuators) and the latter will need to deploy/configure the required VMs to meet those needs. The SEMIoTICS framework should also support the communication of the IIoT gateway with the remote-control center, as the gateway needs to send the results of the actions taken towards the remote-control center.

The SCADA system integrated in this scenario needs to establish communication with the individual turbines, the sub-station and the meteorological stations so that the wind park operator can supervise (e.g., monitor) it effectively. Another cross-layer interaction within the scope of automated configuration needed to be considered, covers the connectivity between the control devices (SDN controller) and IoT cloud components. Additionally, the interaction between the grid operation, that will send instructions regarding the stability/efficiency of the grid based on demand-response (i.e. to adjust the power generation), to the SCADA system must be established.

Finally, the interaction from cloud to edge, between the third-party OEM vendors and their assets (e.g., turbines), should be taken into consideration, since during the maintenance period of the wind park the vendors commonly need to access the turbines or any other equipment they provided.

Regarding the **SARA scenario,** the SEMIoTICS framework should provide the connection between the AREAS Cloud services and the IoT infrastructure and consecutively the interaction between the IoT infrastructure and the actual IoT devices via the four SARA hubs. Moreover, the communication between the specialized artificial intelligence services and the SARA hubs should be supported, as the latter rely on those functions (e.g., object & people/face recognition, natural language processing etc.). Finally, the software client interfaces allowing the various actors (e.g., General Practitioners) to access the management functionalities are provided by the AREAS suite, hence the SEMIoTICS framework does not need to provide additional interfaces on that matter.

Considering the **IHES scenario**, we identified that the SEMIoTICS framework should support the exchange of information, aggregated by the AI Sensing IIoT gateway, between itself, the cloud and the sensing units.

2.3.4.3 CROSS-APPLICATION & HIGHER-LEVEL SERVICES

SEMIoTICS should also facilitate **cross-application** interactions; e.g., application A exchanging data with application B, for the latter to calculate some values based on the output of application A. While a direct interaction falls out of the scope of SEMIoTICS, there could be cases where the two applications are interfaced through different instances of SEMIoTICS. We foresee that in such cases, the cross-platform considerations detailed in subsection 2.3.3.1. The SEMIoTICS framework will be able to interface with **higher level services**, exposing interfaces to enable the deployment of higher-level services that orchestrate existing applications/services/infrastructures. To allow for these value-added services, the SEMIoTICS network's northbound API, as well as the backend interfaces, will have to expose the necessary resources that may be needed for the creation of such services (e.g., network connectivity view, network resource view, computing resource view).

2.4 Interfacing with IoT applications

Typically, applications are hosted on end-user devices, e.g., smart phone or desktop computer, and they interact with the cloud-layer of a system. Examples of such applications are messenger or email clients, mapping tools, shopping, or personal health apps. A cloud backend provides the interface (API) to the pool of data (e.g., emails, maps, products). In these cases, the application is clearly restricted to the upmost layer of the system stack [5][6].

IoT applications are defined and characterized differently. We consider here applications that are hosted by an IoT device (i.e. a thing), a gateway to multiple IoT devices, or by a device residing on the Edge of the network. For example, this could be a data analytics application that analyses the data generated by the hosting device (e.g., process monitoring and optimization, predictive maintenance, or functional safety). Other examples for such IoT applications are user interfaces (if the device has a display) and visualizations. We also consider the provision of protocol bindings as IoT applications. Such a provision of a protocol binding is an IoT application that offers an interface to other applications for accessing data or functionalities of the device. Here, the interface of the application needs to be clearly defined. An example could be an application that implements an HTTP or CoAP REST interface to interact with a device. The Web of Things activities [5] at the W3C are working on a standard for such an interface.

Figure 6 shows the architectural model proposed by the W3C Web of Things group as an execution environment for IoT applications. Thereby the concept of "servient" is central. It can be applied to the different layers, i.e., a servient could be hosted by an IoT device or even on the Cloud. In the W3C approach, the Thing Description [6] provides comprehensive metadata about the possible interactions of the servient and thereby describes the interface of the IoT application.

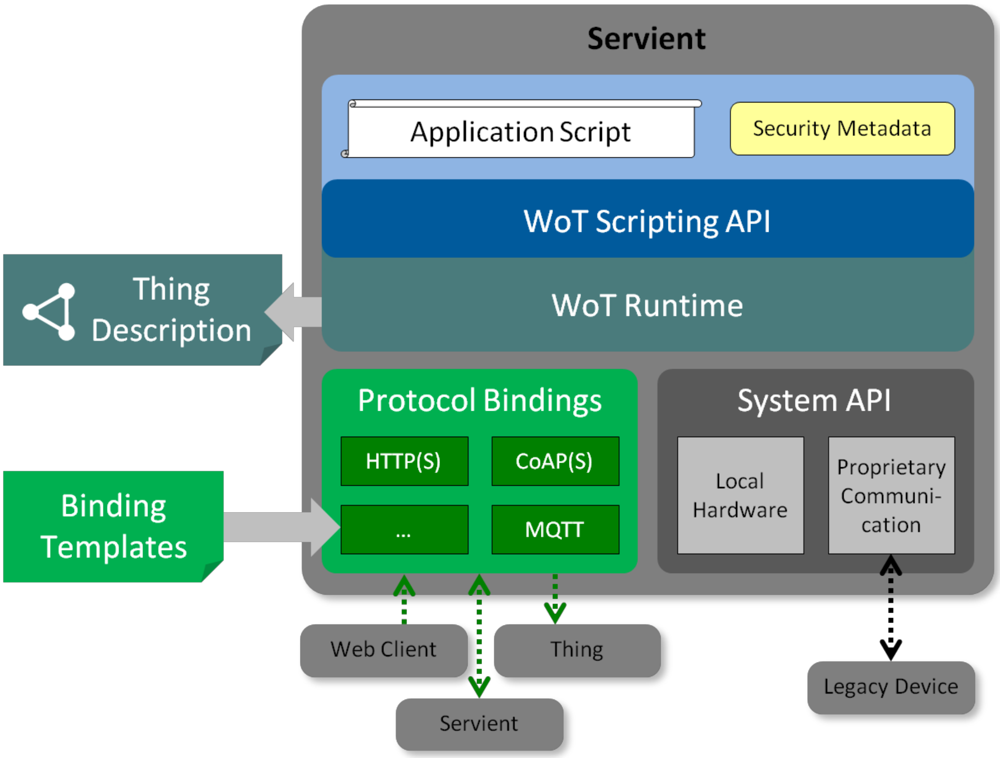


FIGURE 6. ARCHITECTURE FOR IOT APPLICATION EXECUTION USING W3C WEB OF THINGS [5]

Besides the above described case of single IoT applications hosted on a dedicated device, there are use cases where multiple IoT applications, distributed on multiple devices, interact and collaborate with each other towards a common goal. These IoT applications can be on different layers of the system stack (see cross-layer interactions above). An example for such a scenario is described in Figure 7 below. The figure illustrates a data flow from three devices, a microphone, accelerometer and camera, which could be deployed in a wind turbine of the SEMIoTICS Wind Park scenario. The data output of these devices and their installed IoT applications flows then further to other IoT applications hosted by e.g., edge devices. In Figure 7, the data from the IoT devices is going into analytics components and then to a data correlation component to determine whether the PLC controlling the turbine needs to be stopped. This collaboration of IoT applications is similar to microservices architectures.

An important requirement for SEMIoTICS is to enable this definition of flows between IoT applications. Also, it is crucial to be able to define QoS constraints between the flows from one IoT application to the next. These high-level application QoS constraints need to be translated into network-level QoS constraints. The IoT applications interface designs as well as the flows need to be able to capture all relevant information to support this definition and translation of QoS constraints. To realize these requirements, we rely on and extend our previous work on *recipes* for IoT application, which we describe in Section 2.3.2 below.

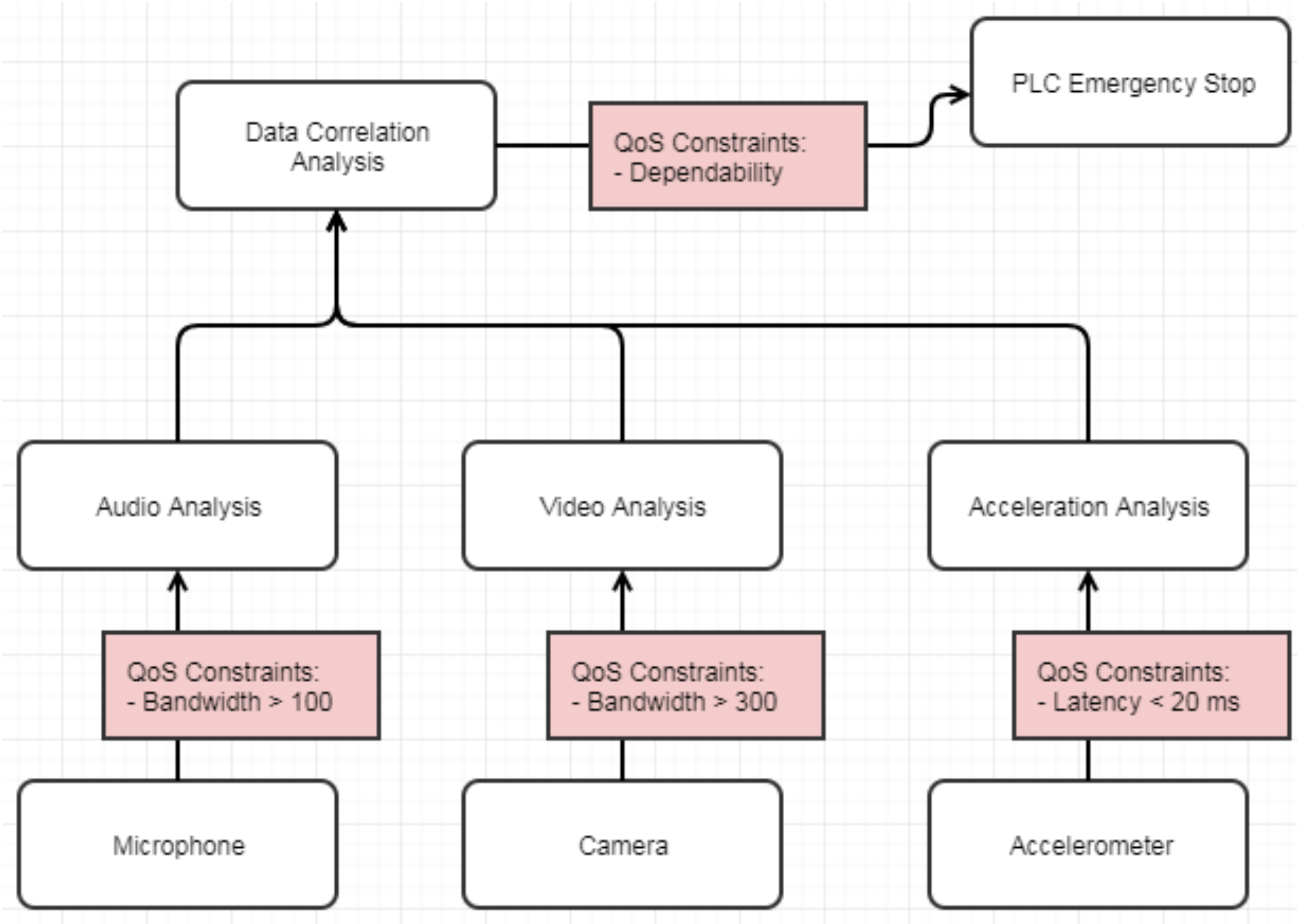


FIGURE 7. EXAMPLE OF A DATA FLOW BETWEEN IOT APPLICATIONS OF A WIND TURBINE

2.5 Requirements Specification considerations

Table 1 below features some essential network interfacing requirements, as defined in deliverable D2.3 (“Requirements specification of SEMIoTICS framework”) that need to be examined concurrently with designing the pattern language.

TABLE 1. NETWORK INTERFACING REQUIREMENTS

SEMIoTICS Requirement: Req. ID	SEMIoTICS Requirement: Description	Network interfacing considerations	Reference
R.NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities	These are core requirements for the development of the network interfacing capabilities of the SEMIoTICS framework. In this regard, the developed interfacing mechanisms must, by design, be tailored to and support the SPDI-driven approach that is at the core of SEMIoTICS, enabling both the SPDI reasoning as well as the transmission of SPDI-related information across layers.	The developed solutions feature a dedicated network layer module integrated in the SDN controller module that is able to process and reason on SPDI patterns, as well as communicate these to the backend (see Section 4). Moreover, the network interface exposed by said controller module is driven by SPDI parameters (see detailed specification in subsection 4.2).
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation		
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	The network interfaces must, define and enforce (via the pattern engine) mechanisms that guarantee the establishment of E2E connectivity (e.g., by 5G cellular network, Bluetooth BLE) between different types of devices (e.g. SARA hubs, sensors, backend servers), actors (e.g., human operators, applications) and interaction type (e.g., maintenance, medical staff, simple user/patient). Additionally, the networks should support various more complex interactions such as cross platform (e.g., cloud apps <-> private cloud), cross layer interactions (e.g., field devices <->backend), cross application (e.g., SDN controller <-> remote management service) or interactions with higher level services (e.g., Third-party	The networking capabilities of the SEMIoTICS framework as a whole (the focus of WP3 activities) do, in tandem, cover these requirements and guided the design and specification of the Pattern-driven controller NBI (Section 4).
R.UC1.1	Automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders.		
R.UC2.3	The SEMIoTICS platform SHOULD guarantee proper connectivity between the various components of the SARA distributed application. The SARA solution is a distributed application not only because it uses different cloud services (e.g., AREAS Cloud services, AI services) from different remote computational nodes, but also because the SARA application logic itself is distributed across various edge nodes (SARA Hubs).		

		entities); more in section 2.3.3.	
R.GP.5	Interaction between SDN controller and IoT backend cloud through a dedicated interface (called northbound software interface)	The northbound interfaces will provide applications, such as the NFV orchestrator or heterogeneous IoT backend platforms, with access to information (e.g., links' traffic load, reliability and latency per link). The southbound interfaces will dynamically reconfigure network nodes. A functional requirement for SDN is to provide southbound interfaces with switches, IoT gateways and routers (e.g., OpenFlow, NETCONF, OFCONF, OVSDB). Both of these complex interactions will need to be supported by interoperability mechanisms defined in the pattern language.	Covered by Pattern-driven NBI (Section 4) and interfacing capabilities of SEMIoTICS SDN Controller (SSC).
R.GP.6	Interaction between SDN controller and network nodes (e.g. switches, routers or IoT Gateways) through dedicated interface (called southbound software interface)		Covered by Southbound interfacing capabilities of SSC
R.NL.9	Interface between the VIM and the SDN controller to allow VTN.		Covered by Pattern-driven NBI (Section 4), interfacing capabilities of SEMIoTICS SDN Controller (SSC), and backend MANO features.
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication)	The IoT Orchestration mechanisms (i.e., the Recipes) in tandem with the pattern language, should define and provide the communication between various IoT devices through their interfaces. Further, the interaction with edge devices should also be assured.	The Pattern Language, along with the associated Pattern Engine components (see deliverable D4.1) encompass reasoning on SPDI and QoS properties. By extension, the pattern-driven NBI interface (presented in Section 4 herein)
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern		
R.GP.7	SDN controller giving feedback for a future generation of SPDI patterns to avoid using the same pattern in case of failure		
R.UC1.3	There MUST be enabled the definition of network QoS on application-level		

	and automated translation into SDN controller configurations.	Finally, using pattern-based operations SEMIoTICS should translate high-level application SPDI and QoS constraints to network-level QoS constraints, enforcing the requirements by triggering adaptations where needed (e.g., to provide fault tolerance).	covers these aspects as well and acts as an enabler for the end-to-end deployment and reasoning on the SPDI and QoS properties of the IoT orchestrations. Moreover, the integration of a "Recipe"-based IoT Orchestration (see subsection 5 and deliverable D4.1) leveraging standardized semantic models facilitates interoperability with existing works.
R.UC1.4	Network resource isolation MUST be performed for guaranteed Service properties – i.e. reliability, delay and bandwidth constraints.		
R.UC1.5	Fail-over and highly available network management SHALL be performed in the face of either controller or data-plane failures.		
R.UC1.12	Standardized semantic models for semantic-based engineering and IIoT applications MUST be utilized.		
R.UC2.15	"The SEMIoTICS platform SHOULD provide low latency connectivity between the SARA hubs and cloud services (i.e. AREAS cloud services and AI services) to allow offloading of near real-time computation intensive tasks to the cloud.)		
R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.	The network interfaces developed in the context of T3.4 will need to feature strong and unambiguous security controls, including encryption, authentication and logging, to minimize risk of unauthorized use and compromise.	Covered by Network Interface Security considered by design (see subsection 4.4), and work in the context of Task 4.5, focusing on E2E security and privacy.
R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.		
R.S.7	The negotiation interface of the SDN Controller SHALL be secure against network-based attacks		
R.NL.11	Secure communication with the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules), as well as the communication between VIM, SDN Controller, and MANO, with data paths acting as computing nodes for VNF spinoff		
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised		

2.6 Associated KPIs

In addition to the requirements stemming from the project's concept and approach (as described in subsections 2.1 to 2.4), as well as the formally defined project requirements (subsection 2.5), an additional aspect considered are the overarching Objectives and associated KPIs. These are detailed in Table 2.

TABLE 2. CONSIDERATIONS AND RELATION TO OVERARCHING PROJECT OBJECTIVES AND ASSOCIATED KPIS

Objective		KPI		
#	**Description**	**ID**	**Description**	**Relation to network interface**
1	Development of patterns for orchestration of smart objects and IoT platform enablers in IoT applications with guaranteed security, privacy, dependability and interoperability (SPDI) properties.	KPI-1.1	Delivery of 36 verified SPDI patterns covering the 6 core property types for 3 data states and 2 cases	The interface specification and processing capabilities at the network layer (see section 4) is by design compatible with the pattern language developed within SEMIoTICS and is able to process the associated patterns, as well as reason on the associated properties locally (enabling local embedded intelligence at the network layer). 6 of the developed patterns focus on Interoperability aspects.
		KPI-1.2	Machine-processable pattern language	
2	Development of semantic interoperability mechanisms for smart objects, networks and IoT platforms	KPI-2.3	Validated semantic interoperability between the SEMIoTICS framework and 3 IoT platforms	The NBI specified within T3.4 is pattern-driven, and the backend Pattern Orchestration elements are integrated with the semantically rich "Recipes" approach for defining IoT Orchestrations and the associated semantic components (see sections 5 and deliverable D4.1). Therefore, through this integration, and the presence of semantic mediator components, the semantic interoperability is facilitated. The pattern-driven NBI presented herein is a key enabler in this (see section 6).
3	Development of dynamically and self-adaptable monitoring mechanisms supporting integrated and predictive monitoring of smart objects of all layers of the IoT implementation stack in a scalable manner.	KPI-3.2	Delivery of a monitoring language capable of defining platform agnostic monitoring conditions (as part of SPDI patterns), correlations of different IoT platform events that are necessary for this, and predictive monitoring checks	The pattern language includes monitoring of SPDI properties and related features (see deliverable D4.1). Moreover, the translation of SPDI and QoS property requirements in pattern-driven orchestrations to monitoring policies is provided (see deliverable D4.2). While the monitoring and reasoning of said properties is catered for by the relevant components developed in the context of T4.2 and T4.1, respectively, the needed instantiation of communications is enabled by the interface specified within T3.4.
4	Development of core mechanisms for multi-layered embedded intelligence, IoT application adaptation, learning and evolution, and end-to-end security, privacy, accountability and user control.	KPI-4.2	Delivery of adaptation mechanisms that support proactive and reactive, as well as horizontal and vertical adaptation actions, related to network, smart objects and IoT platforms with an adaptation time of 15ms	Pattern-driven adaptations of the SEMIoTICS platform are, in the context of the network layer, enabled by the interface specified in the context of T3.4/D3.4, in tandem with the Pattern Engine (reasoning) components developed in the context of T4.1 that is integrated into the SDN controller. Said component interfaces with components within the controller for path instantiation and for getting a real-

				time view of the network conditions (see subsection 4.1 herein).
		KPI-4.6	Development of 3 new security mechanisms/controls enabling the secure management of smart devices and sensors over programmable industrial networks	The Pattern-driven management and adaptation of the networking infrastructure, enabled by the pattern mechanisms (T4.1/D4.1) and the relevant network interface presented herein (T3.4/D3.4) is a core part of this KPI.
6	Development of a reference prototype of the SEMIoTICS open architecture, demonstrated and evaluated in both IIoT (renewable energy) and IoT (healthcare), as well as in a horizontal use case bridging the two land-scapes (smart sensing), and delivery of the respective open API	KPI-6.1	Reduce Required Manual Interventions	The Pattern-based specification of the network and its properties (through work carried out in T3.4) enables the pattern-driven management of the networking aspects of the IoT infrastructure and, in line with the SEMIoTICS semi-autonomous operation, will allow for the reduction of manual interventions required for maintaining the required properties of the IoT deployment (see subsection 3.4, section 6, and the pattern-driven IoT Orchestrations approach detailed in D4.1 – with final version due to appear in D4.8).

3 ENABLING TECHNOLOGIES

In this section, we discuss about the key enabling technologies that drove forward the development and adaptation properties required for the SEMIoTICS framework to support the deployment of network services from all SEMIoTICS layers. The section covers the various networking protocols in order to identify their strengths and weaknesses, and then elaborates on the data formats that can be used for the delivery of the measurements from the field layer to the backend and vice versa.

3.1 Networking protocols

Continuous innovations in hardware, software and communication solutions in the last decade have led to the expansion of the Internet of Things (IoT) with the number of connected devices growing vigorously. The huge amount of data generated by these devices require to find a system architecture able to both process and store all the data. Hence, several networking protocols have been proposed in order to properly manage the large amount of data. In this section, we discuss about the most prominent networking protocol solutions, including HTTP, AMQP, CoAP, MQTT and YANG.

3.1.1 HYPERTEXT TRANSFER PROTOCOL (HTTP)

This protocol is the fundamental client-server model protocol used for the Web, and the most compatible with the existing network infrastructure. Currently, the most widely accepted version of this protocol is HTTP/1.1. The communication between a client and a server is established via a request/response messaging, with the client sending an HTTP request message and the server returning a response message, containing the resource that was requested if the request was accepted. Recently, HTTP has been associated with Representational State Transfer (REST), a guideline for developing web services based on a specific architectural style in order to define the interaction between different components. Because of the success of RESTful Web services, there has been a lot of effort in bringing this architecture into IoT based systems by combining HTTP and REST. The combination of HTTP protocol with REST is commendable, as it is very easy to create, read, update, and delete data (the so-called CRUD operations). According to this mapping, the operations for creating, updating, reading and deleting resources correspond to the HTTP POST, GET, PUT and DELETE methods, respectively. For developers, the fact that REST establishes a mapping of these CRUD operations with HTTP methods, means that they can easily build a REST model for different IoT devices. The presentation of the data is not pre-defined and as such, the type is arbitrary, with the most common being JSON and XML, as we will discuss in the following section. In most cases, IoT standardizes around JSON over HTTP. In Figure 8, we illustrate a typical REST HTTP request/reply interaction.

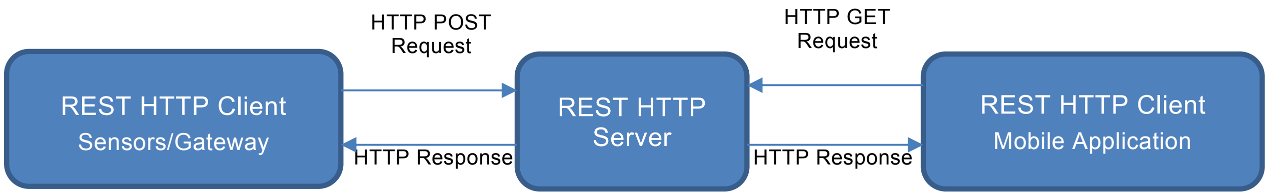


FIGURE 8. REST HTTP REQUEST/REPLY INTERACTION MODEL

Regarding the transport protocol used, HTTP uses TCP. While using TCP provides reliable delivery of large amounts of data, which is an advantage in connections that do not have strict latency requirements, it creates challenges in resource constrained environments. One of the main problems is that the constrained nodes most of the time send small amounts of data sporadically and setting up a TCP connection takes time and produces unnecessary overhead. Moreover, for quality of service (QoS), HTTP does not provide additional options, but instead it relies on TCP, which guarantees successful delivery as long as connection is not interrupted.

Regarding security, HTTP uses the very well-known TLS for enabling secure encrypted communication channel, resulting in a secure version of HTTP, also known as HTTPS. The first part of securing the client-server data exchange is a TLS handshake, implemented as an exchange of a 'client hello' and a 'server hello' messages where they have to agree upon a cipher suite, which is a combination of algorithms they will use to assure secure settings. After that, the client and server exchange keys based on the agreed key exchange algorithm. The result is an exchange of messages encrypted with a shared secret key. The data is encrypted to prevent anyone from listening to and understanding the content.

To summarize, the REST HTTP is not enough for the IoT-cloud communication, due to its complexity. It uses the request/reply paradigm, which is not suitable for push notifications, where the server delivers notifications to client without client request. Moreover, the amount of additional data over the protocol TCP is too large because of multiple provided options, which is unnecessary for simple computing nodes in the lower levels of IoT architecture. HTTP does not explicitly define QoS levels and requires additional support for it. This meant that it was necessary to explore other messaging protocols or to improve the HTTP itself. To this end, the new version of the protocol, HTTP/2.0, introduces a number of improvements, some of which are especially relevant in IoT context. It enables a more efficient use of network resources and a reduced latency by introducing compressed headers using a very efficient and low memory compression format and allowing multiple concurrent exchanges on the same connection. These features are particularly interesting for the IoT as it means the size of packets is significantly smaller, making it a more adequate option for constrained devices. Additionally, it introduces the so-called server push, which means the server can send content to clients with no need to wait for their requests. The drawbacks of HTTP/2.0 are not known yet, as there are no implemented solutions reported in the literature, as of today.

3.1.2 ADVANCED MESSAGE QUEUING PROTOCOL (AMQP)

AMQP is an open standard protocol, standardized by OASIS, designed to enable interoperability between a wide range of different applications and systems, regardless of their internal designs. It was originally, developed for business messaging with the idea of offering a non-proprietary solution that can manage a large amount of message exchanges that could happen during a short time in a system. This AMQP interoperability feature is significant as it allows that different platforms, implemented in different languages, can exchange messages, which is crucial in heterogeneous systems.

AMQP has been implemented in two very different versions, AMQP 0.9.1 and AMQP 1.0, each with a completely different messaging paradigm. AMQP 0.9.1 implements the publish-subscribe paradigm, which revolves around two main AMQP entities, both part of an AMQP broker: the exchanges and the messages queues. The exchanges represent a part of the broker that is used to direct the messages received from publishers. The publishing of messages to an exchange entity is the first step in the process, and after that, the messages are routed into one or more appropriate queues. This depends on whether there are more subscribers interested in a particular message, in which case the broker can duplicate the messages and send their copies to multiple queues. A message will stay in the queue until it is received by a subscriber. This routing process, that actually links exchanges and queues, depends on so called bindings, which are predefined rules and conditions for message distribution. On the other hand, the newer version of the AMQP protocol, AMQP 1.0, is not tied to any particular messaging mechanism. While the older versions of the protocol used the publish-subscribe approach with an architecture that consists of exchanges and messages queues, the new AMQP implementations exploit a peer-to-peer protocol and can be used without a broker in the middle. A broker is present only in the communication that needs to provide a store-and-forward mechanism, while in other cases direct messaging is possible. This option of supporting different topologies increases the flexibility for the possible AMQP based solutions, enabling different communication patterns such as client-to-client, client-to-broker, and broker-to-broker.

AMQP uses TCP for reliable transport, and in addition it provides three different levels of QoS. QoS-0 delivers on the best effort basis, without confirmation on message reception. For example, a temperature sensor sends data every few minutes. For this kind of telemetry information over a longer time period, it is acceptable if sometimes the messages are missing, because the average temperature is still known since most of the message updates have been received. The next level of guarantee is QoS-1, which assures that messages will arrive, so a message confirmation is necessary. This means receiver must send an acknowledgement, and if it does not arrive in a defined period of time, the publisher will send a publish message again. The third option, QoS-2, guarantees that the message will be delivered exactly once without duplications. Since in resource constrained nodes the battery life is more important than reliable communication, QoS-0 is a valid option. For the message exchange between more powerful nodes, QoS-1 and QoS-2 are obviously better options. Finally, the AMQP protocol provides complementary security mechanisms, for data protection by using TLS protocol for encryption, and for authentication by using SASL (Simple Authentication and Security Layer).

With all the features it offers, AMQP has relatively high power-, processing- and memory-related requirements, making it a rather heavy protocol, which has been its biggest disadvantage in IoT-based ecosystems. This protocol is better suited in the parts of the system that is not bandwidth and latency restricted, with more processing power.

3.1.3 CONSTRAINED APPLICATION PROTOCOL (COAP)

This protocol was designed by the Constrained RESTful Environments (CoRE) working group of IETF for the use in constrained devices with limited processing capabilities. Similar to HTTP, one of its most important characteristics is its use of tested and well accepted REST architecture. With this feature CoAP supports request/response paradigm just like REST HTTP, and especially for constrained environments. CoAP is considered a lightweight protocol, so the headers, methods and status codes are all binary encoded, thus reducing the protocol overhead in comparison with many protocols. It also runs over less complexed UDP transport protocol instead of TCP, further reducing the overhead. When a CoAP client sends one or multiple CoAP requests to the server and gets the response, this response is not sent over a previously established connection but exchanged asynchronously over CoAP messages. The price of this reduction is reliability. It should be noted that because of the unreliability that comes with using UDP, which proved to be a problem for some environments, IETF created an additional document adding the possibility of CoAP running over TCP.

CoAP relies on a structure that is divided into two logically different layers. One of the layers, dubbed as request/response layer implements the RESTful paradigm and allows for CoAP clients to use the same methods as HTTP when sending requests. Thus, clients can use GET, PUT, POST or DELETE methods to manage the URI identified resources in the network. The same procedure is followed in HTTP when requesting to obtain data from the server, for example sensor value, client will use method GET with a server URL, and as a reply will receive a packet with that data. The request and responses are matched through a token; a token in the response has to be the same as the one defined in the request. It is also possible for a client to push data, for example updated sensor data, to a device by using method POST to its URL. As we can see, in this layer CoAP uses the same methods as REST HTTP. What makes it different, is its other layer. Because UDP does not ensure reliable connections, for reliability CoAP relies on its second structural layer - message layer, designed for retransmitting lost packets. This layer defines four types of messages: CON (Confirmable), NON (non-confirmable), ACK (Acknowledgement), and RST (reset). The CON messages are used for ensuring reliable communication, and they demand to be acknowledged from the receiver side. Precisely this feature to mark whether the messages need the acknowledgement is what enables QoS differentiation in CoAP, albeit in a limited fashion.

CoAP has an optional feature that can improve the request/response model by allowing clients to continue receiving changes on a requested resource from the server by adding an observe option to a GET request. With this option, the server adds the client to the list of observers for the specific resource, which will allow the client to receive the notifications when resource state changes. Instead of relying on repetitive polling to check for changes in resource state, setting an observe flag in a CoAP client's GET request, allows an interaction which is similar to a publish-subscribe paradigm with the server alerting a client when changes exist. In an attempt to get even closer to publish/subscribe paradigm, IETF has recently released the draft of Publish-Subscribe Broker that extends the capabilities of CoAP for supporting nodes with long interruptions in connectivity and/or up-time, with preliminary performance evaluations showing promising results [8].

As a security mechanism CoAP uses DTLS on top of its UDP transport protocol. It is based on TLS protocol with necessary changes to run over an unreliable connection. The result is a secure CoAPS protocol version. Most of the modifications in comparison to TLS include features that stop connection termination in case of lost or out of order packets. An example is a possibility to retransmit handshake messages. The handshaking process is very similar to the one in TLS, with the exchange of client and server 'hello' messages, but with the additional possibility for a server to send a verification query to make sure that the client was sending its 'hello' message from the authentic source address. This mechanism helps prevent Denial-of-Service attacks. Through these messages, the client and server also exchange supported cipher suits and keys, which will further be used for data exchange protection during the communication. Since DTLS was not originally designed for IoT and constrained devices, new versions optimized for the lightweight devices have emerged recently. Some of the DTLS optimization mechanisms with a goal of making it more lightweight include IPv6 over Low-power Wireless Personal Area Network (6LoWPAN) header compression mechanisms to compress the DTLS header. However, due to its limitations, optimizing DTLS for IoT is still an open issue.

3.1.4 MESSAGE QUEUING TELEMETRY TRANSPORT (MQTT)

MQTT is one of the lightweight messaging protocols that follows the publish-subscribe paradigm, which makes it rather suitable for resource constrained devices and non-ideal network connectivity conditions, such as with low bandwidth and high latency. MQTT was released by IBM, with its latest version MQTT v3.1 adopted for IoT by the OASIS [10]. Because of its simplicity, and a very small message header comparing with other messaging protocols, it is often recommended as the communication solution in IoT. MQTT runs on top of the TCP transport protocol, which ensures its reliability. In comparison with other reliable protocols, i.e., HTTP, MQTT uses lighter headers with are much lower power requirements, making it one of the most prominent protocols solutions in constrained environments.

There are two communication parties in the MQTT architecture that usually take the roles of publishers and subscribers, clients and servers/brokers. Clients are the devices that can publish messages, subscribe to receive messages, or both. The client must know about the broker that it connects to, and for its subscriber role it has to know the subject it is subscribing to. A client subscribes to a specific topic, in order to receive corresponding messages. However, other clients can also subscribe to the same topic and get the updates from the broker with the arrival of new messages. The broker serves as a central component that accepts messages published by clients and with the help of the topic and filtering, delivers them to the subscribed clients. In MQTT, instead of using RESTful HTTP, a publish-subscribe interaction model can be used. The local server has a role of broker, e.g., a PC. For this role, it is necessary to install the MQTT broker library, for example the Mosquitto broker [10], which is one of best-known open source MQTT brokers. It should be noted that there are various other MQTT protocol brokers that are open for use, which differ by way of implementation of the MQTT protocol. A Raspberry Pi could serve as an MQTT client, by installing appropriate MQTT client libraries, such as the Paho Library [11] that is fully compatible with the Mosquitto broker. These clients correspond to IoT abstraction layer, representing devices with sensing and computing capabilities. The broker, on the other hand, corresponds to the higher abstraction layer representing a cloud computing node, characterized by larger computing and storage capacities.

The messages are the string data and they have to be labelled with topics. Topics in MQTT are treated as a hierarchy, with strings separated by slashes that indicate the topic level. One MQTT publisher can publish messages to define a set of topics. This information will be published to the broker which can temporally store it in a local database in case that later another interested subscriber appears. MQTT uses TCP which can be critical for constrained devices. To this end, a solution has been proposed as MQTT for Sensor Networks (MQTT-SN) version that uses UDP and supports topic name indexing. The MQTT-SN was specifically designed for sensor networks and is considered to be an improved version of MQTT. It does not depend on TCP, but instead uses UDP as faster, simpler, and more efficient transport option over a wireless link. The other important improved feature is the reduced size of the payloads. This is done by numbering the data packets with numeric topic id's rather than long topic names. The biggest disadvantage is that at the moment MQTT-SN is only supported by a few platforms, and there is only one free broker implementation known, called Really Small Message Broker.

For QoS, MQTT has the same three QoS levels as AMQP, QoS-0, QoS-1, and QoS-2. The amount of resources necessary to process MQTT packet increases with higher QoS level, so it is important to adjust the QoS choice to specific network conditions. Another important feature MQTT offers is the possibility to store some messages for new subscribers by setting a 'retain' flag in published messages. For example, a temperature sensor publishes new information when the temperature changes. By default, if there is nobody interested in that topic, broker will discard the published messages. In some situations, especially when the state of the followed topic does not change often, it is useful to enable for new subscribers to receive the information on that topic. In this default case new subscribers would have to wait for the state to change in order to receive a message about the temperature. By setting a 'retain' flag to value: true broker is informed that it should store the published message, so it could be delivered to new subscribers, as shown in Figure 9.

Publish	
Topic	"factory/processor"
Payload	{"temperature": 50}
QoS	1
Retain	True

Subscribe	
Topic	"factory/processor"
QoS	1

FIGURE 9. MQTT PUBLISH/SUBSCRIBE INTERACTION MODEL

Since it was designed to be as lightweight as possible, MQTT does not provide encryption, instead data is exchanged as plain-text, which is clearly an issue from the security point of view. Therefore, encryption needs to be implemented as a separate feature, for instance via TLS, which on the other hand increases overhead. Authentication is implemented by many MQTT brokers, through one of the MQTTs control type message packets, called CONNECT. Brokers require from clients, that when sending the CONNECT message, they should define username/password combination before validating the connection or refusing it in case the authentication was unsuccessful.

All in all, while all aforementioned messaging protocols have their own advantages, MQTT and CoAP are more suitable for IoT frameworks like SEMIoTICS as they are more lightweight and can offer a wide range of capabilities in constraint environments. On the other hand, HTTP and AMQP are heavyweight and not recommended. However, since HTTP is widely used in the industry, the SEMIoTICS framework will provide support for all MQTT, CoAP, and HTTP messaging protocols.

3.1.5 OVERVIEW OF NETWORKING PROTOCOLS

All in all, while all aforementioned messaging protocols have their own advantages, MQTT and CoAP are more suitable for IoT frameworks like SEMIoTICS as they are more lightweight and can offer a wide range of capabilities in constraint environments. On the other hand, HTTP and AMQP are heavyweight and not recommended. However, since HTTP is widely used in the industry, the SEMIoTICS framework will provide support for all MQTT, CoAP, and HTTP messaging protocols. Table 3 aggregates the above into a comparison table.

TABLE 3. NETWORKING PROTOCOLS' OVERVIEW

	HTTP	AMQP	COAP	MQTT
Architecture model	client/server	peer-to-peer	client/server	Publish/subscribe
Transport	TCP	TCP	TCP	TCP
Payload	Heavyweight	Heavyweight	Lightweight	Lightweight
QoS Levels	1	3	1	3
Security	High	High	Medium	Medium

3.2 Data formats

The purpose of this section is to provide enough information about most widely used data formats, their viability in IoT systems and their current implementations. Data formats play an integral role in IoT, defining directly the overall system performance in terms of resources and security. Extra overhead data must remain compact and the format is preferred to be simplistic due to the limited capabilities of the sensing devices and vast amount of data processed and stored.

3.2.1 EXTENSIBLE MARKUP LANGUAGE (XML)

XML is a markup language that defines a set of rules for encoding data in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality, and usability across devices. It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, the language is widely used for the representation and interchange of arbitrary data structures such as those used in web services over the internet. Many industry data standards are based on XML and the rich features of the XML schema specification. Disparate systems communicate with each other by exchanging XML messages. In general, XML and its extensions have regularly been criticized for verbosity, complexity and redundancy. Mapping the basic tree model of XML to type systems of programming languages or databases can be difficult, especially when XML is used for exchanging highly structured data between applications, which was not its primary design goal.

JSON is frequently proposed as simpler alternative that focus on representing highly structured data, which may contain both highly structured and relatively unstructured content. However, the standardized XML schema specifications offer a broader range of structured XSD data types compared to simpler serialization formats and offer modularity and reuse through XML namespace.

3.2.2 JAVASCRIPT OBJECT NOTATION (JSON)

JSON is a lightweight data interchange format. It is an open-standard file format that uses human-readable text to transmit data objects. It is easy for humans to read and write. It is easy for machines to parse and generate. It is mainly based on a subset of the JavaScript programming language.

It is a very common data format used for asynchronous browser–server communication, including as a replacement for XML. JSON has become a popular inter-process communication (IPC) data interchange format for a variety of computer languages. It enables structured data to be serialized into a text format. It is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures: A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array. And an ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In contrast, XML is a markup language. JSON on the other hand is a way of representing data objects. Generally, JSON is preferred for IoT applications since it can self-describe and is more programmatic, where XML was initially made for document mark up like HTML. JSON is typically used in IoT protocols that do not provide native support for data structure serialization, due to its simplicity.

3.2.3 GOOGLE PROTOCOL BUFFERS

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data. It works by initially defining the data structure, called message, once and then by using a special generated source code to easily write and read the structured data to and from a variety of data streams and using a variety of languages. The data structure can also be updated without causing problems, because it is well defined and backwards compatible.

Protocol Buffers are a method of serializing structured data. It is useful in developing IoT applications that communicate with each other over a wire or for storing data. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.

Protocol Buffers developed and widely used internally at Google for storing and interchanging all kinds of structured information. The method serves as a basis for a custom remote procedure call (RPC) system that is used for nearly all inter-machine communication at Google.

Canonically, messages are serialized into a binary wire format which is compact, forward and backward compatible, but not self-describing. There is no defined way to include or refer to such an external specification within a Protocol Buffers file. The officially supported implementation includes an ASCII serialization format, but this format—though self-describing—loses the forward- and backward-compatibility behaviour and is thus not a good choice for applications other than debugging.

Though the primary purpose of Protocol Buffers is to facilitate network communication, its simplicity and speed make Protocol Buffers an alternative to data-centric C++ classes and structs, especially where interoperability with other languages or systems might be needed in the future.

Protocol buffers have many advantages over XML for serializing structured data. They are simpler, 3 to 10 times smaller in size and they are 20 to 100 times faster [12]. They also generate data access classes that are easy to be employed by any type of programming language.

Overall, JSON is a better and more versatile option. The simplicity and advanced features of the XML schema specification should also not be ignored. Although Protocol Buffers offer some compelling advantages, JSON and XML are well-established and flexible data formats, widely used in IoT industry and thus will be used in the SEMIoTICS framework.

3.2.4 OVERVIEW OF DATA FORMATS

Considering the above, overall JSON appears as more versatile option that is better suited to the requirements of the project. The simplicity and advanced features of the XML schema specification should also not be ignored. Although Protocol Buffers offer some compelling advantages, JSON and XML are well-established and flexible data formats, widely used in IoT industry and thus will be used in the SEMIoTICS framework. Table 4 presents a comparison among the data formats.

TABLE 4. DATA FORMATS' OVERVIEW

	XML	JSON	PROTOBUF
Creator	W3C	Douglas rockford	Google
Standardized	No	Yes	No
Specification	Specs	Specs	Guide
Binary	Partial	No	Yes
Human-readable	Yes	Yes	Partial
Standard APIs	Yes (DOM, SAX, XQuery, XPath)	Partial, JSON-LD	C++, C#, Java, Python, JavaScript, Go

3.3 Data Modeling - Yet Another Next Generation (YANG)

YANG is a data modelling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF), NETCONF remote procedure calls, and NETCONF notifications. A YANG module defines a hierarchy of data that can be used for NETCONF-based operations, including configuration, state data, Remote Procedure Calls (RPCs), and notifications. This allows a complete description of all data sent between a NETCONF client and server.

Moreover, YANG models the hierarchical organization of data as a tree in which each node has a name, and either a value or a set of child nodes. YANG provides clear and concise descriptions of the nodes, as well as the interaction between those nodes.

In this protocol, the data models are structured into modules and submodules. A module can import data from other external modules, and include data from submodules. The hierarchy can be augmented, allowing one module to add data nodes to the hierarchy defined in another module. This augmentation can be conditional, with new nodes appearing only if certain conditions are met.

Additionally, YANG models can describe constraints to be enforced on the data, restricting the appearance or value of nodes based on the presence or value of other nodes in the hierarchy. These constraints are enforceable by either the client or the server, and valid content MUST abide by them. YANG defines a set of built-in types, and has a type mechanism through which additional types may be defined. Derived types can restrict their base type's set of valid values using mechanisms like range or pattern restrictions that can be enforced by clients or servers. They can also define usage conventions for use of the derived type, such as a string-based type that contains a host name. YANG permits the definition of reusable groupings of nodes. The instantiation of these groupings can refine or augment the nodes, allowing it to tailor the nodes to its particular needs. Derived types and groupings can be defined in one module or submodule and used in either that location or in another module or submodule that imports or includes it. YANG data hierarchy constructs include defining lists where list entries are identified by keys that distinguish them from each other. Such lists may be defined as either sorted by user or automatically sorted by the system. For user-sorted lists, operations are defined for manipulating the order of the list entries.

YANG modules can be translated into an equivalent XML syntax called YANG Independent Notation (YIN), allowing applications using XML parsers and Extensible Stylesheet Language Transformations (XSLT) scripts to operate on the models. The conversion from YANG to YIN is lossless, so content in YIN can be round-tripped back into YANG. YANG strikes a balance between high-level data modelling and low-level bits-on-the-wire encoding. The reader of a YANG module can see the high-level view of the data model while understanding how the data will be encoded in NETCONF operations.

YANG is an extensible language, allowing extension statements to be defined by standards bodies, vendors, and individuals. The statement syntax allows these extensions to coexist with standard YANG statements in a natural way, while extensions in a YANG module stand out sufficiently for the reader to notice them. YANG resists the tendency to solve all possible problems, limiting the problem space to allow expression of NETCONF data models, not arbitrary XML documents or arbitrary data models. The data models described by YANG are designed to be easily operated upon by NETCONF operations. To the extent possible, YANG maintains compatibility with Simple Network Management Protocol's (SNMP's) SMIv2 (Structure of Management Information version 2). SMIv2-based MIB modules can be automatically translated into YANG modules for read-only access. However, YANG is not concerned with reverse translation from YANG to SMIv2.

3.4 IoT Workflow Composition

Key enabling technologies for SEMIoTICS are methods for enabling the definition and execution of IoT workflows that are composed of several services and devices. In the last decades, there has been intense research activity related to the composition of Web services, and more recently also IoT services and devices. In the following, we provide an overview about this field of research based on our previous works [7] and [8].

3.4.1 COMPOSING SERVICES AND DEVICES

Service composition tackles the challenges of discovering services, reserving them, and connecting them to each other. Thereby, we can distinguish two distinct kinds of service composition[9]: *orchestration* and *choreography*. The first case relies on a composed service that controls the interaction with other services, while in the second case the control is distributed, and each Web service describes its part of an interaction. Basis for the composition of services are formal descriptions of their interfaces. This is also the case for the composition of IoT functions or data offered by services, platforms, or devices.

Traditional approaches describe services solely based on syntactical information, as it is for example done using WSDL [10] by specifying service interfaces, their offered operations, and data types. However, if different vendors develop services independently it can easily happen that the same functionalities (or data) are provided by services using different data types. This issue can be addressed by using semantic descriptions or annotations. Instead of relying solely on syntactic data types for the specification of service interfaces, a semantic description for operations and data is used. This description can relate to the semantic enrichment of the service interface and used data formats, as well as the description of constraints for utilizing the service (e.g., quality of service, availability, location, time, or price).

The OWL-S ontology [11] is a W3C recommendation that can be used together with the Web Ontology Language (OWL) [12] to define the semantics of data and operations of Web services. Conceptually similar to OWL-S is the WSMO standard [13]. Building up on the widely-used WSDL standard, SAWSDL [14] has been the first standard for adding semantic annotations to such descriptions of Web services. However, in more recent years, the design of Web services and APIs followed more and more often the REST principles, instead of WSDL and SOAP based Web services. Consequently, approaches for the interface designs for IoT applications and the flows need to be able to capture all relevant information to support this definition and translation of QoS constraints. To fulfil these requirements, we rely on and extend our previous work on *recipes* for IoT application, which we describe in D4.1, Section 3, semantically describing RESTful services came up and examples are hRESTS [15] and RESTdesc [16].

Using semantically-enriched descriptions of services makes their discovery more powerful, as it allows finding of services depending on their semantic descriptions. Through utilizing semantic descriptions, also the composition of services can be advanced by automatically finding semantically matching service instances that can interact. Service compositions utilize service components to provide added functionality. A standard for purely syntactic Web service orchestration is WS-BPEL[28] It is largely supported in industry.

A prominent realization of semantic service composition is the Simple Semantic Web Architecture and Protocol (SSWAP). It originated from the BioMOBY [25] project and comprises over 2.400 resources published in the field of genetic engineering [17]. Other projects that tackle semantic service orchestration include Service Web 3.0 [26] or SOA4All [18].

3.4.2 SUPPORTING THE COMPOSITION OF SERVICES AND DEVICES

Today, when new devices are added to an IoT environment, they need to be connected physically, and the software on the centralized controller needs to be reparametrized and reconfigured. Manually designing composite services is time-consuming, cumbersome, and error prone. If there is a high number of Web services, this is hardly applicable [9].

There have been attempts to fully automate the generation of service compositions based on some user defined request or goal. For example, [16] present a service composition system that enables the goal-driven configuration of smart environments based on semantic meta-data and reasoning. Such fully automated approaches are still facing challenges. The key difficulty lies in the unambiguous semantic description of the goal and states that lead to the goal. This becomes very challenging when dealing with more complex environments, where a lot of devices can interact in a lot of ways and each has a lot of possible states. In such environments semantic models become complex, and reasoners turn to be inefficient when solving goals over them.

Due to these challenges, a *semi*-automated approach that supports users in creating service compositions, rather than to fully automate the process, seems to be the most promising direction. Previous work has been done in this direction, such as [18], where optimal service compositions are automatically computed with support of composition templates, or [19], where a composability model is introduced to ascertain that Web services can be safely combined.

Commercially successful systems such as "If This Then That" (at ifttt.com) use simple composition techniques similar to the recipe context but create and execute centralized orchestrations instead of decentralized choreographies [20] . The IFTTT platform lacks systematic engineering support leading to widely duplicated recipes, as shown by Ur [21]. Node-RED [27] is another tool that follows a similar approach. It provides a browser-based editor that makes it easy to wire hardware devices, APIs and online services, thereby creating application flows. Flows are specified in JSON. Giang et al. [22] focus on application-level distributed choreographies by building on Node-RED as a visual programming tool. However, they do not address the configuration of critical automation systems and their need for failure detection and recovery.

In our previous work [7] the concept of *recipes* has been introduced to represent the design of an IoT service composition separate from its implementation. A semi-automated service composition and instantiation tool is provided, in order to assist the user in creating the composition of placeholders for actual services and devices. Later, these placeholders are replaced with actual services and devices based on suggestions provided by the system using semantic reasoning. While in [7],we generate a simple application script, which is executed by a centralized orchestrator. In [8], we extended this approach by enabling the distributed execution of instantiated recipes as *choreographies*.

"Recipes" define templates for compositions of ingredients and their interactions. Ingredients are placeholders for offerings, i.e., devices and services that process and transform data. Interactions describe the dataflow between these ingredients. The Recipe model (shown in Figure 10) is a light-weight semantic model that describes ingredients and interactions semantically. An ingredient of a recipe specifies the requirements that should be fulfilled by an offering in order to create an application. An Interaction between the ingredients is defined by creating a data-flow between them. That is, by connecting the output data of one ingredient with the input data of another ingredient. In addition to this, an interaction also specifies an operation that defines the operation (e.g., GET, POST, OBSERVE etc.) to be performed on an ingredient to access its data or function.

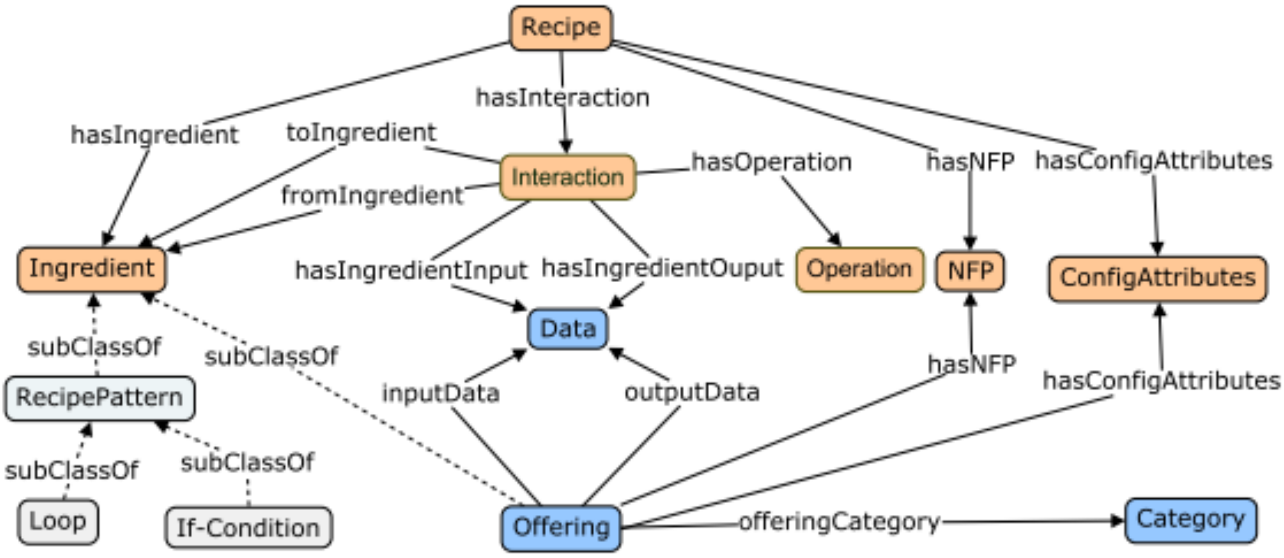


FIGURE 10. RECIPE MODEL [7]

An example recipe, as a template for a lighting control system, is shown in Figure 11. A lighting controller takes input from brightness sensors, calculates the output brightness through an algorithm (averaging, for example) and outputs the calculated value to the connected lights, but only if one of the switches is switched on. Inputs and outputs have both a name and a type. The type is used for matching offerings with ingredients.

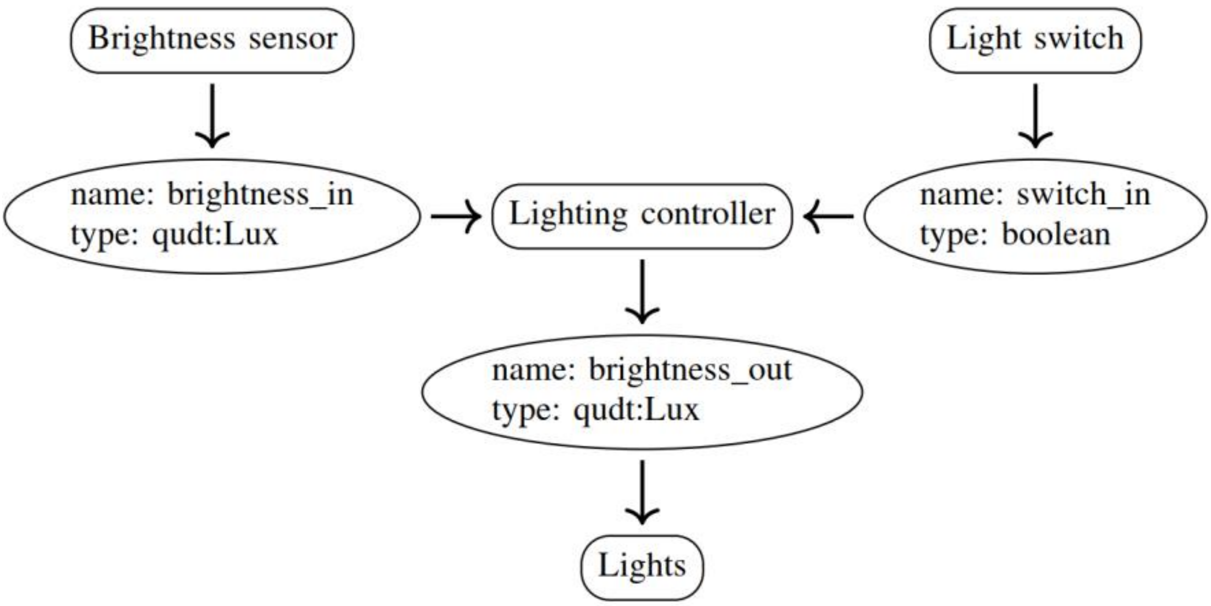


FIGURE 11. EXAMPLE OF A RECIPE [8]

More details on the integration of the above Recipes approach to the pattern-driven SPDI management of the SEMIoTICS deployment (across all layers, including network) can be found in deliverable D4.1.

4 PATTERN-DRIVEN NORTHBOUND INTERFACE

To support the capabilities detailed in Section 1, in addition to the SBI and NBI interfaces and associated technologies detailed in D3.1 and D3.2 a pattern-driven NBI is integrated into the SEMIoTICS SDN controller (SSC). We discuss the corresponding design herein. The design augments the semantically rich networking capabilities with SPDI-driven management of the network layer's operation, as well as its interactions with the south (the field layer) and north (e.g., the SEMIoTICS backend, IoT applications, and external IoT platforms) entities.

4.1 Interface Design

As per SEMIoTICS architecture definition (Figure 12), the majority of interactions at the SSC's exposed NBI are consumed by the overarching Pattern Orchestrator. In the controller YANG is used as a general-purpose modelling language. In order to be compatible with the OpenDaylight controller that already supports YANG, we implement the aforementioned NBIs as REST-based RPCs defined in YANG. In addition, the YANG language, being protocol independent, can be converted into any encoding format, e.g. XML or JSON that the network configuration protocol supports. In order to be flexible in terms of using a variety of network management tools it is considered beneficial to use YANG for modelling.

Based on the above, the Pattern Orchestrator leverages the REST-based northbound interface of the controller to describe the pattern requirements initiated at the higher-layer recipe definition. In the SSC development, the Pattern Schema describing the structure of the networking-related pattern, is intentionally kept open and extensible to support the most diverse types of connectivity-related patterns possible.

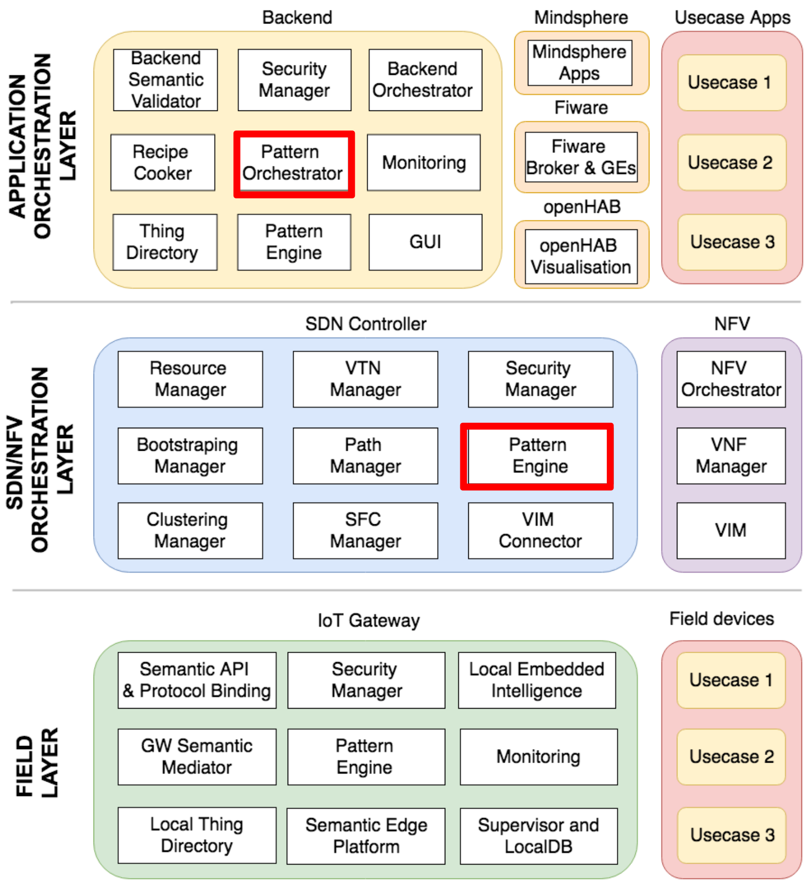


FIGURE 12. PATTERN-DRIVEN NBI ENABLING COMPONENTS IN THE SEMIOTICS ARCHITECTURE

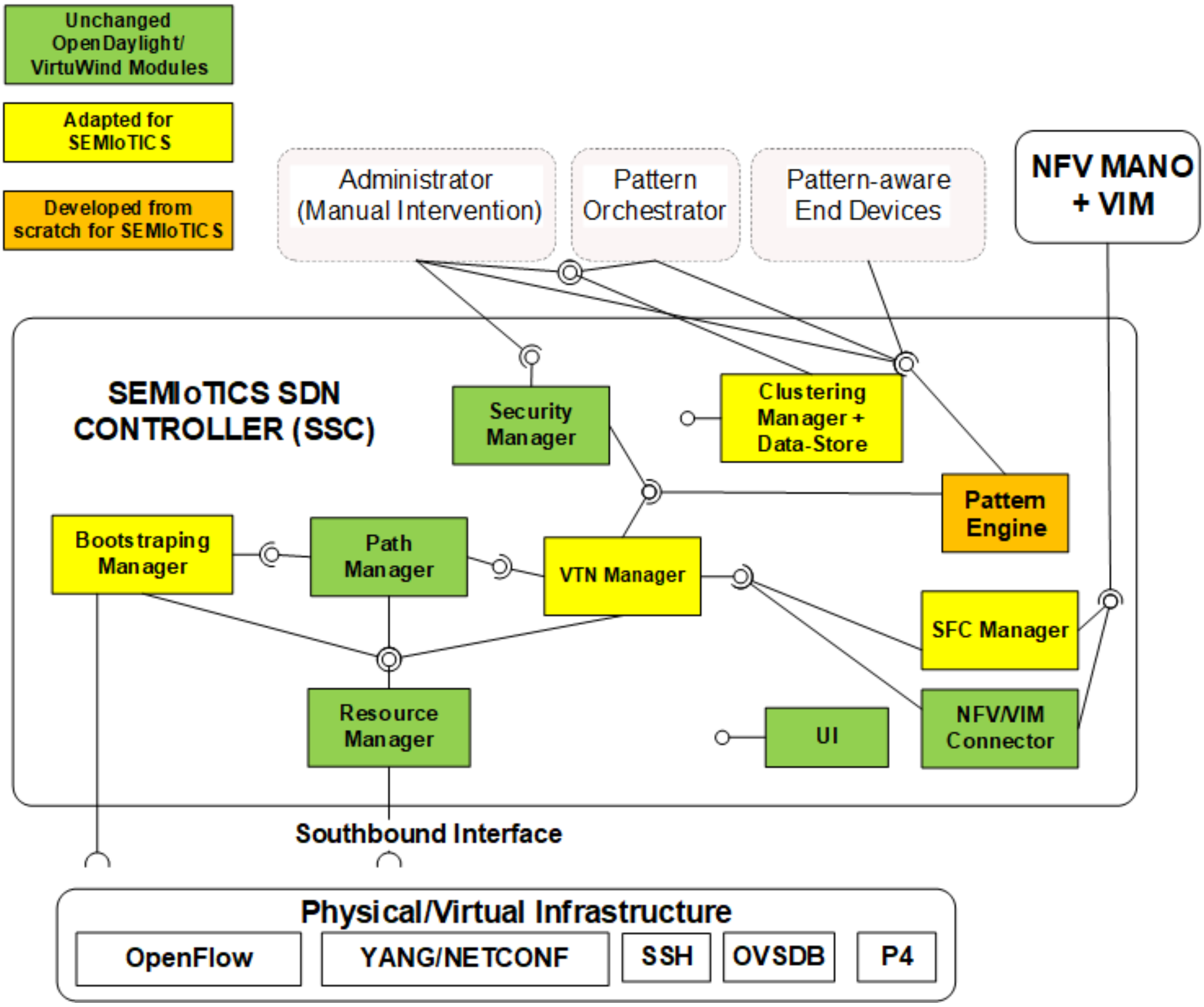


FIGURE 13. THE PATTERN-DRIVEN NBI (IN RED) WITHIN THE SEMIOTICS SDN CONTROLLER'S PATTERN MODULE

As mentioned in D4.1, patterns can be used as an instrument for modifying and verifying the topology of SDN networks, at runtime. At runtime, an existing SDN network design (topology) and the required SPDI properties are provided, and patterns are applied to reconfigure the network according to the specified constraints. The controller is then in charge of analysing the applied pattern and ensuring that the pattern invariants are satisfied. The analysis is based on checking if the state of the network configuration matches totally or partly the specified rule objective with corresponding constraints. When a network that matches and embeds an SDN pattern does not satisfy the required property, the pattern may be used to substitute, add or remove an existing configuration in order to satisfy the property.

Figure 13 depicts the pattern-driven network APIs providing a closer view of the architecture at the SDN/NFV orchestration level. Moreover, the pattern-driven network APIs (refer to D3.1 for SDN Controller architecture details) define and monitor the operation of different SPDI properties of the applications that interact with the API. Doing so it is guaranteed that the said interactions are in line with the SPDI requirements. When that is not the case, appropriate adaptations are triggered. The corresponding information, in this case, is relayed to the Pattern Engine at the SEMIoTICS backend, since this is the entity responsible for SPDI reasoning at the application level.

4.2 Interface Specification

The implementation approach for the pattern-driven network interfacing in order to provide a machine processable form of SPDI properties definition and reasoning is in line with the one defined in Section 4 of D4.1 ("SEMIoTICS SPDI Patterns (first draft)"). Therefore, network-level SPDI properties are expressed as Drools [23] production rules, and the associated rule engine, by applying and extending the Rete algorithm [24].

In more detail, a Drools rule that encodes an SDN pattern includes the inputs of the pattern's components, the type of composition and the required property in Left Hand Side (LHS). When the conditions in the LHS are satisfied, then the rule is fired to execute the actions as described in its Right-Hand Side (RHS). In the RHS, the new requirements of the compositions or atomic components can be inserted, updated or deleted.

In order to specify and express SDN patterns, the semantics of the pattern language should be defined. In Table 5 the most useful preliminary network semantics are presented. In the LHS, the network components which constitute the topology of the pattern are defined. Different network topology facts such as Nodes, Links and Flows are included in the list. Moreover, the Requirement represents the constraints of the topology and the required property. In the RHS, the pattern provides the solution by inserting, modifying, updating or retracting facts from the knowledge base which will also update the inventory list in the controller. Each component is converted through the respective Java class to an understandable format to the SDN controller. Finally, the semantics of Drools language give the potentiality to represent more complex patterns by adding more variables and pattern properties.

Key elements in the pattern rule definition include:

- **Pattern Requirement:** The pattern constraints are defined as requirements which represent the property that the pattern guarantee such as E2E path establishment or fault tolerance. Depending on the type of requirement property, the SDN controller may utilize various routing algorithms to deploy the path, e.g., Dijkstra algorithm which adapts breadth-first algorithm to find single source shortest path in the simplest case, or more complex path finding for cases where additional QoS properties are specified.
- **Pattern Action:** When the pattern identifies the fitting path between source and destination, the actions of the pattern includes the installation of suitable flow rules in the OpenFlow-enabled switches.

TABLE 5. NETWORK PATTERN RULE CONSTRUCTS

Type	Syntax	Description
rule	rule "name"	name of the rule
Left Hand Side (LSH)		
	Network Pattern Elements (Facts)	
when	Node (address, ports, txPackets, rxPackets)	match network nodes such as switches and hosts
	Link (srcId, srcPort, destId, destPort)	match links between source and destination nodes
	Path (srcId, destId)	match paths between source node intermediate links and destination node
	Flow (switchId, inPort, outPort, priority)	match flow rules between nodes

	Requirement (src, dest, category, satisfied)	match requirements of pattern such as source, destination, property category and satisfied
	Conditional Elements	
	==	match conditions
	contains	contains object (logical)
	not	not match (logical)
	!=	not match (arithmetic)
Right Hand Side (RSH)		
then	**Actions**	
	modify (\$fact)\{pro=pro'\}	modify knowledge base fact
	retract (\$fact)	retract knowledge base fact
	insert (new Fact ())	insert knowledge base fact
	update (\$fact)	update knowledge base fact
	Java commands	other Java language syntax

We extend the above presented LSH Network Pattern Elements from [33] with additional matching requirements for more-detailed specification of the QoS-encompassing patterns specific to SEMIoTICS use cases:

– **Application structure:** structure that contains application-related information and consists of the following fields
  - **Application Identifier:** A unique identified for the specific application
  - **Application Tenant**: identifier for different tenant contexts of each application
– **Service structure:** Structure that contains information about a running or a requested service

  Service: substructure that groups connectivity, QoS and time requirements for a requested service
  - **Service Identifier:** A unique service identifier.
  - **List of Flows**: the application defines the flows that requests to be established and the QoS requirement for each flow. The default connectivity type that this design enables is unidirectional point-to-point. This is considered as a single flow that can have specific QoS requirements. However, the design is also made in a way that allows establishing bidirectional flows by also providing the reverse flow information as well as point-to-multipoint connectivity by defining a number of flows which share the same source identifier. Note that this scheme allows each requested flow in a bidirectional or a point-to-multipoint scenario (or even in a scenario that combines them) to have different QoS requirements. In more detail, each industrial flow entry consists of, i) Flow Requirement Structure and ii) Flow QoS structure.
    - **Flow Requirement structure**: Structure that defines the information of the end hosts for each required E2E connection request
      - Endpoint structure (src, dst): structure that specifies different options for expressing end host information:
        - Host: host identifier information (i.e., a generic node ID/name)
        - Host MAC: MAC address and VLAN identifier information
        - Host IP: IP address information
        - Host IP+port number: IP address and port information
    - **Flow QoS structure**: Structure that groups QoS requirements for each end-to-end connection
      - **Bandwidth**: measured in kbit/s, default value 0 – no bandwidth guarantees.

- **Burst**: maximum burst size of a flow, measured in Bytes, default value 0 – no burst size guarantees.
- **Delay**: measured in milliseconds, default value 0 – no delay guarantees. O
- **Resilience**: integer identifier of a resilience class – values:
  - **0** (default value) – no protection
  - **1** – OpenFlow standard protection (using OpenFlow Fast-Failover Groups)
  - **2** – rapid path protection (using a data-plane-based end-to-end custom protection mechanism that addresses the limitations of the OpenFlow Fast-Failover Groups – to be developed in a later phase of the project)

4.3 Implementation Details

The development of SEMIoTICS' Pattern-driven Network Services API, as well as the preliminary testing results is presented in this section. The technologies that are used include Java, Maven, and YANG.

The pattern-driven NBI follows the same logic regarding the interaction with the Pattern Orchestrator as the other Pattern Engines in the other layers (i.e., Backend and Field) as described in D5.2 section 3.4.2. The main purpose of this interaction is to dispatch the created Drools facts and rules. These interactions take place via the common API exposed by the Pattern Engines; the design decision to have a common API in all Pattern Engines was taken to enhance homogeneity and facilitate integration, as applications can interact with all Pattern Engines, regardless of their layer, through the same functions.

Moreover, said API is used by the pattern-driven NBI (as well as the Pattern Engine residing to the field layer) to send at runtime fact updates to the Backend Pattern Engine, allowing the latter to have an up-to-date view of the SPDI state of SDN layer and the corresponding components.

The main web services exposed from the pattern-driven NBI are shown in Figure 14.

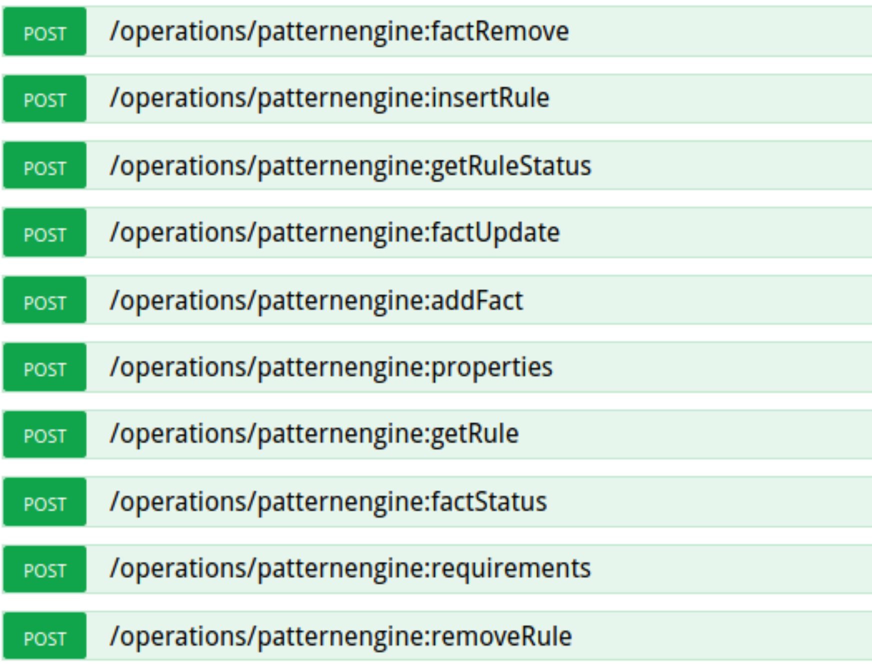


FIGURE 14. PATTERN-DRIVEN NBI API

The above correspond to the creation, retrieval, deletion of facts and creation and deletion of rules. In more detail, the addFact REST service is used by the Pattern Orchestrator for the communication of new Drools facts of a new IoT Service orchestration.

Moreover, the factRemove is used in order for a fact to be deleted from the Drools Memory of the SDN Pattern Engine. The factUpdate is used again by the Pattern Orchestrator in case some changes need to be applied to a Drools Fact. The factStatus REST service returns the current status of a special type of Drools facts, the instances of Property class. These instances are used to describe SPDI and QoS properties for the components of an IoT Service orchestrator. The requirements REST service can be used for the visualization of the SPDI properties of an orchestration. Finally, the insertRule REST service is used only by the Pattern Orchestrator to communicate Drools Rules to the pattern-driven NBI for the reasoning of the SPDI and QoS properties.

4.3.1 TESTING

In order to test and demonstrate the functionality of the pattern-driven NBI, we assume a simple setup which features a virtual network topology comprising an SDN controller, one OpenFlow switch (s1) and two hosts (which are part of a testing orchestration, interacting with each other); see Figure 15, and refer to D5.3, section 3.4.2.3 for a more detailed description.

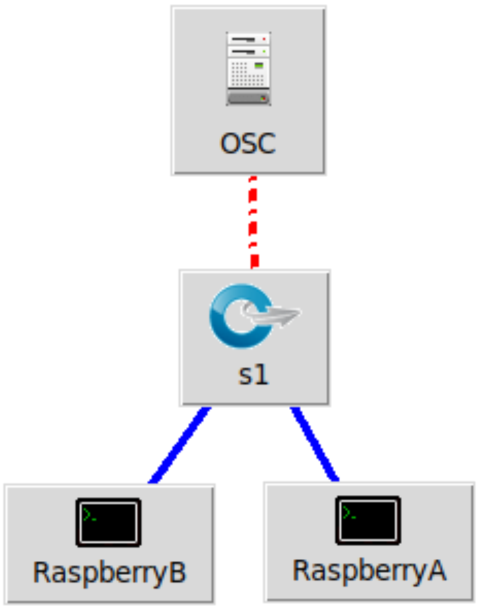
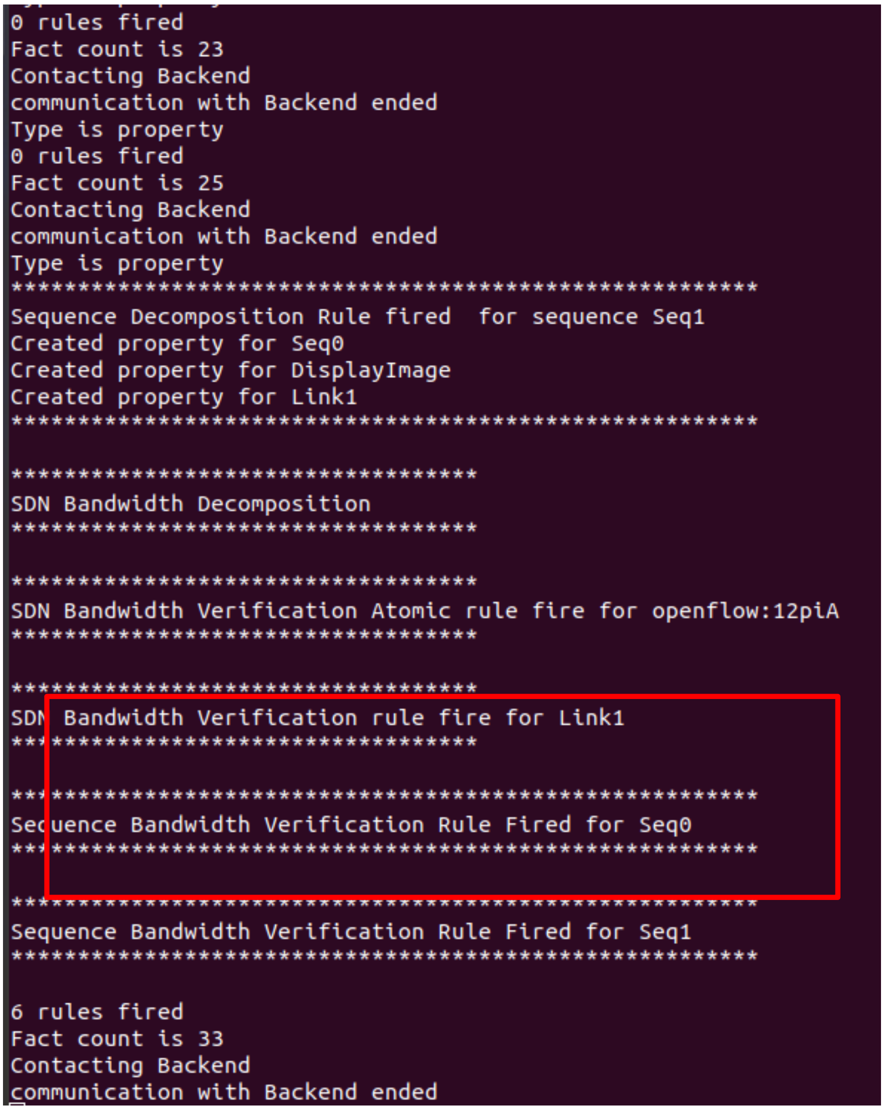


FIGURE 15. TEST SETUP NETWORK TOPOLOGY

The first step of our testing methodology is to send a request to Pattern Orchestrator, which is expected to receive instantiated orchestrations of IoT services (i.e. Recipes). The Pattern Orchestrator receives the incoming Recipe and creates instances of the corresponding Java classes (Host, Softwarecomponent, Link, Sequence, Property), which correspond to Drools facts in the Pattern Engine. These Java instances are then sent to the pattern-driven NBI through one of its REST APIs, called addFact. The Pattern Engine receives all the Java instances sent by the Pattern Orchestrator. Each of the received instances are inserted into the working memory of the Drools Rule Engine, as Drools facts. Once there, they can trigger Drools Rules (patterns; see Figure 16).



```
0 rules fired
Fact count is 23
Contacting Backend
communication with Backend ended
Type is property
0 rules fired
Fact count is 25
Contacting Backend
communication with Backend ended
Type is property
*********************************************************
Sequence Decomposition Rule fired  for sequence Seq1
Created property for Seq0
Created property for DisplayImage
Created property for Link1
*********************************************************

***********************************
SDN Bandwidth Decomposition
***********************************

***********************************
SDN Bandwidth Verification Atomic rule fire for openflow:12piA
***********************************

***********************************
SDN Bandwidth Verification rule fire for Link1
***********************************

*********************************************************
Sequence Bandwidth Verification Rule Fired for Seq0
*********************************************************

*********************************************************
Sequence Bandwidth Verification Rule Fired for Seq1
*********************************************************

6 rules fired
Fact count is 33
Contacting Backend
communication with Backend ended
```

FIGURE 16. TESTING PATTERN-DRIVEN NBI OPERATION AND RULE FIRING

The output of the network drools rules that run for the verification of properties in question, is included in the red rectangle. The description of the test setup, evaluation scenario, network rules and assessment results for the pattern-driven interface and its interactions with the Pattern Orchestrator have been presented in detailed in section 3.4.2.3 of D5.3; for the sake of brevity we do not repeat them here, and refer the reader to said deliverable.

Due to the decision to use of standardized, REST-based interfaces for all pattern engines, the development of a dedicated client was not needed. Instead, the use of any typical REST client is adequate to test the implementation and compatibility of third party modules in terms of their interaction with the Pattern Engines. In our testing Postman was used (a well-established tool in this regard) to send these requests to the exposed REST APIs Postman allows us to create REST requests with the needed headers and the body we are interested in. To that end, specific scripts were created that allowed us to send well-formulated requests and test all operations supported by the Pattern Engines' interfaces (including the pattern-driven NBIs). These scripts are available at the project's GitLab.

It should also be noted that, since fundamental functionality is in place and tested, and basic interfacing is achieved, efforts towards the final stages of the project will focus on the interconnection of the Pattern

Orchestrator component (see D4.1) with the NFV Management and Orchestration components via the Os-Ma-Nfvo endpoint, and consequently with the SEMIoTICS SDN Controller (SSC) via its Pattern-driven NBI presented herein. This will not only enable the E2E deployment of semantically-rich IoT Orchestrations (as described in subsection 3.4), but also the triggering of adaptations (e.g., spawning a new VNF when needed to guarantee a specific SPDI or QoS property). Nevertheless, these developments are outside of the scope of Task 3.4, and thus are not documented herein.

4.4 Interface Security Considerations

The Pattern Related components are responsible not only for validating SPDI/QoS properties but enforcing them when necessary as well. Therefore, it is imperative to take into consideration the security aspects of the exposed interface.

The SDN Pattern Engine adopts the security mechanisms available in the ODL controller, and by extension the SSC, which features basic authentication capabilities (via username and password). Through this feature, all modules used by the SDN controller are subject to this authentication. Case in point, the Pattern Orchestrator is forced to provide credentials in order to be able to communicate with the SDN Pattern Engine.

Additionally, security is hardened on a per-case basis, considering the intrinsic requirements (e.g., complexity of interactions) foreseen in each scenario. Said intrinsic requirements are addressed by adding encryption to the communication (E2E, where needed) by using SSL/TLS in the REST endpoints.

Moreover, advanced Authentication, Authorisation and Accounting (AAA) features are implemented with the help of the Security Manager who is responsible for providing tokens that enforce the said features (see Task 4.5, focused on E2E security and privacy). Leveraging these mechanisms, the communication between Pattern Orchestrator and pattern-driven NBI is hardened by the use of an authentication token. In all interactions, the Pattern Orchestrator will first request a token from the Security Manager that will later use to contact the pattern-driven NBI. If the token is verified to be able to grant access to the pattern-driven NBI then the communication proceeds successfully. This process prevents unauthorized use of the pattern-driven NBI.

5 PATTERNS FOR NETWORK-LEVEL SEMANTIC INTEROPERABILITY

To fully exploit the above pattern-driven networking features and provide an E2E provision for interoperability throughout the SEMIoTICS framework, a set of Interoperability-focused patterns have to be defined. A first set of such patterns pertaining to the definition and monitoring of SPDI properties across the SEMIoTICS layers, also including Interoperability -related aspects, were defined in Section 5 of D4.1 ("SEMIoTICS SPDI Patterns (first draft)").

The full set of SPDI patterns will be documented in D4.8 ("SEMIoTICS SPDI Patterns (final)") which is currently under finalisation, whereby the network-related ones are extracted from said deliverable and presented herein. An overview of these patterns and their coverage in terms of type, data state and platform connectivity are presented in Table 6.

TABLE 6. SUMMARY OF INTEROPERABILITY PATTERNS AND THEIR COVERAGE

Pattern		Interoperability Type				Data State Coverage			Platform Connectivity	
#	Name	Technical	Syntactic	Semantic	Organisational	In Transit	At Rest	In Processing	Within	Across
1	Technical	✓				✓			✓	
2	Syntactic		✓			✓		✓	✓	
3	Semantic			✓				✓	✓	
4	Organisational				✓	✓		✓		✓
5	E2E Within	✓	✓	✓		✓		✓	✓	
6	E2E Across	✓	✓	✓	✓	✓		✓		✓

Elaborating on Table 6, and as discussed in deliverable D4.1 (see Section 2), four levels of interoperability are considered in SEMIoTICS: technical, syntactic, semantic and organizational interoperability. In more detail, from bottom up, the following types of interoperability can be distinguished and will be covered by SEMIoTICS:

- **Technical interoperability** – enables seamless operation and cooperation of heterogeneous devices that utilize different communication protocols on the transmission layer
- **Syntactic interoperability** – establishes clearly defined formats for data, interfaces and encoding
- **Semantic interoperability** – settles commonly agreed information models and ontologies for the used terms that are processed by the interfaces or are included in exchanged data
- **Organizational interoperability** – cross-domain and cross-platform service integration and orchestration, through common semantic and programming interfaces

The above correspond to the "Interoperability Type" column of Table 6, while the different patterns necessary to ensure the above are presented as lines in said table, with their definitions being presented in the subsection that follow. Moreover, these patterns are classified in terms of their coverage of the different data states and cases of platform connectivity, as is the norm with all patterns defined within SEMIoTICS.

It is important to note that the higher levels of interoperability assume the existence of the lower ones, otherwise they cannot be achieved, e.g., to have syntactic interoperability, you need to have established technical first (see Figure 17).

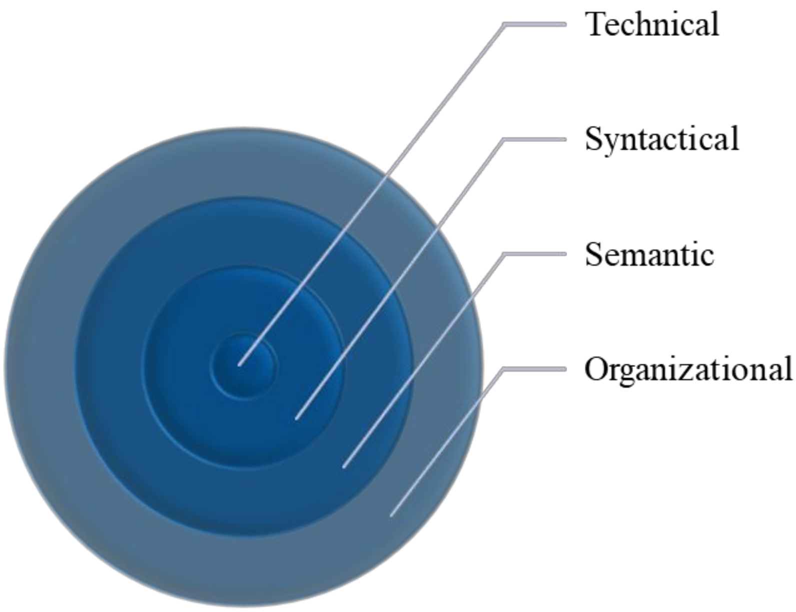


FIGURE 17. DIFFERENT LEVELS OF INTEROPERABILITY AND THEIR DEPENDENCY RELATIONSHIPS

Moreover, even though the boundaries of each level are not strict, we consider in our methodology that technical, syntactic, and semantic interoperability enable E2E interoperability between the involved technologies within a platform and/or specific IoT deployment, while operation across verticals and platforms is accomplished through organizational interoperability.

Regarding data states, it should be noted that interoperability cannot be defined for data at rest since, by definition, data as rest is data that is not being used, access or processed upon and, thus, no interoperability challenges arise. When an entity accesses said data (to read a value, perform analytics etc.), it becomes data in transit and in processing, depending on the scenario. Therefore, as shown in Table 6, the Interoperability of data at rest is not covered within the defined patterns.

The definition of the interoperability patterns of SEMIoTICS is detailed in the subsections that follow.

5.1 Technical Interoperability

Technical Interoperability is about enabling the communication between systems and platforms at a protocol level and the infrastructure needed for those protocols to operate [34]. Within the context of the work presented herein, the associated pattern rule aims to cover and address the technological issues that may arise from the interaction among heterogeneous devices, with different technical specifications and supported communication means on the transmission layer (e.g., wireless motes communicating via ZigBee, other motes via 802.15.4, and more powerful infrastructure devices communicating over WiFi or Ethernet), as is often the case in IoT environments.

5.1.1 PATTERN DEFINITION

Let us consider:

- C := the set of all instantiated components
- TA := A set of technical attributes
- $C_1, C_2 \subseteq C$, where $C_1 \neq C_2$
- $C_{i\_TA} \subseteq TA$:= technical attributes of C_i
- TMD := Technical Mediator (mediator which connects to components with various technical attributes; e.g., a sensor gateway that acts as a bridge between 802.15.4 radio and wired network infrastructures using 6LBR [35])

Then, we can define the following:

Lemma 1: *If C_1, C_2 are at the same domain and $C_{1\_TA} \cap C_{2\_TA} \neq \emptyset$ then C_1 and C_2 are directly technically interoperable.*

Lemma 2: *If C_1, C_2 are on different domain but are both directly technically interoperable with TMD (Figure 18) then C_1, C_2 are indirectly technically interoperable.*

Lemma 3: *If C_1, C_2 are directly or indirectly technical interoperable, then C_1, C_2 are technically interoperable.*

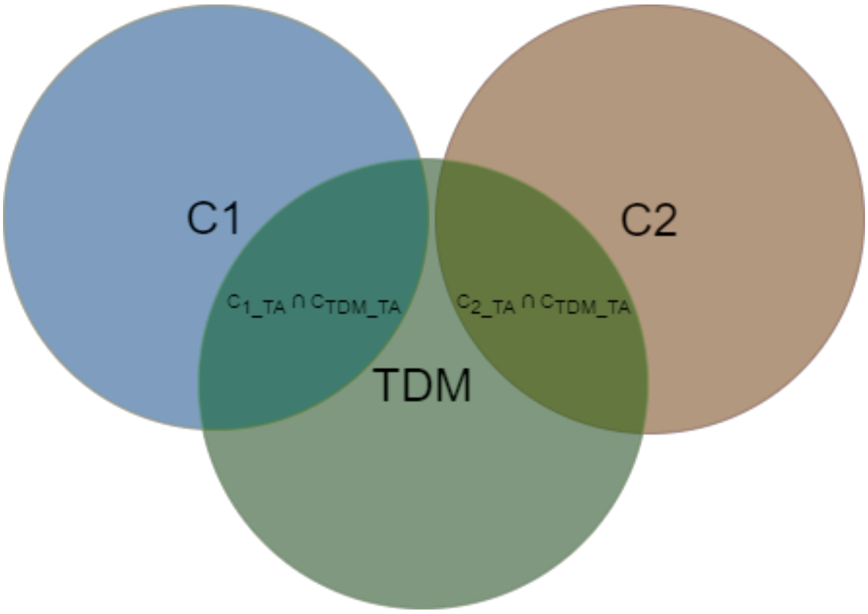


FIGURE 18. INDIRECT TECHNICAL INTEROPERABILITY VIA TECHNICAL MEDIATOR

5.1.2 PATTERN SPECIFICATION RULE

Using the above detailed pattern, we can derive the following workflow-based definition of technical interoperability for the fundamental scenario of two IoT components communicating with each other:

1. <u>**WF “technical-interoperability”**</u>
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (TMD, (PlaceholderActivity, ”technical mediator”))
5. Link (L1, A1, A2)
6. Link (L2, A1, TMD)
7. Link (L3, A2, TMD)
8. Property (conn1, L1, required, (pattern-based, pattern),”_technical-interoperability”, in_transit)
9. Property (conn2, L2, required, (pattern-based, pattern),”_technical-interoperability”, in_transit)
10. Property (conn3, L3, required, (pattern-based, pattern),”_technical-interoperability”, in_transit)

11. Property (conn4, "_technical-interoperability", required, (pattern-based, PR1), "_technical-interoperability", end_to_end)
12. **Pattern rule: (PR1: conn1 || (conn2, conn3) → conn4)**

For details on this workflow-based approach followed in SEMIoTICS pattern definition, please refer to deliverable D4.1, and its follow-up D4.8. In said deliverables the process of transforming these workflows and the pertinent requirements into a machine-processable format using Drools rules and facts is also documented. The output of this process is shown in Listing 1.

LISTING 1. TECHNICAL INTEROPERABILITY VERIFICATION DROOL RULE

```
rule "Sequence Technical Interoperability Verification"
    when
        Placeholder($pA:=placeholderid)
        Property ($pA:=subject, category=="technical", $prvaluein1:=input_value,
$prvalueout1:=output_value, satisfied==true)
        Placeholder($pB:=placeholderid)
        Property ($pB:=subject, category=="technical", $prvaluein2:=input_value,
$prvalueout2:=out_value, satisfied==true)
        Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
        $PR: Property ($sId:=subject, category=="technical", $prvalueout1==$prvaluein2,
satisfied==false)
    then
        modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

Whenever the *"Sequence Technical Interoperability Verification"* rule, is fired, the *Property* with category *technical* of a Sequence is verified. According to the LSH part of the rule, if:

i) the two Placeholders of a Sequence have a Property of category *technical*; and
ii) the output_value of the first Placeholder is equal to the input_value of the second Placeholder ($prvalueout1==$prvaluein2),

then the RHS part of the rule verifies the corresponding Property of the Sequence in question and sets its input_value and output_value to $prvalueout1 and $prvaluein2 respectively.

A property of category *technical* with input_value $prvaluein1 and output_value $prvaluout1, denotes that the corresponding component uses a specific input communication protocol whose description is given by $prvaluein1 and a specific output communication protocol whose description is given by $prvalueout1. For example, if we have a component (PlaceholderA) which has an endpoint that uses the Wi-Fi 802.11a protocol, then this would be translated to a Property with category *technical* and input_value "Wi-Fi 802.11a". Additionally, if the said component uses a different protocol for forwarding information to other components such as Ethernet 802.3, then the output_value of the same Property will be "Ethernet 802.3".

Figure 19 depicts a sequence (Sequence1) of two Placeholders. PlaceholderA has a property of category *technical* with input_value "Wi-Fi 802.11a" and output_value "Ethernet 802.3". Similarly, PlaceholderB has a property of category *technical* with input_value "Ethernet 802.3" and output_value "Wi-Fi 802.11g". As step 1 shows, the corresponding property of the Sequence1 is false and input_value and output_value are not set. The aforementioned protocol properties trigger the *"Sequence Technical Interoperability Verification"* rule, verifying (satisfied=true) the property of category *technical* of Sequence1, and assigning the appropriate values to input_value and output_value of the property (step 2).

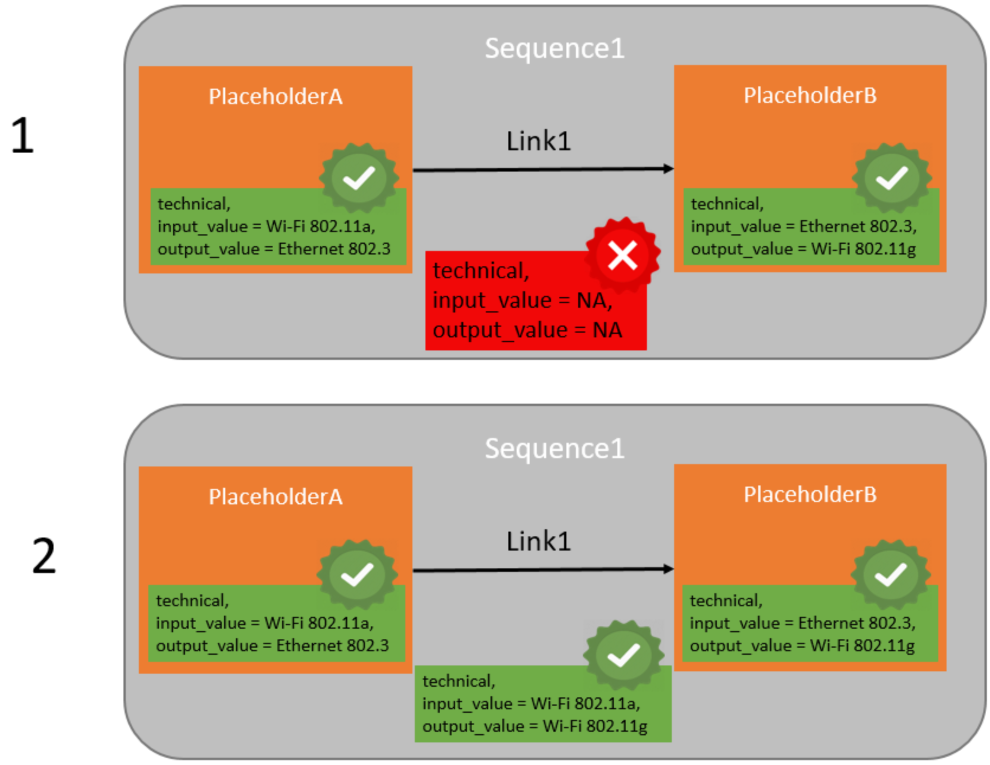


FIGURE 19. EXECUTION STEPS OF TECHNICAL INTEROPERABILITY PATTERN

5.2 Syntactic Interoperability

Syntactic Interoperability refers to the challenge of enabling interactions between different devices which may often be communicating using different messaging protocols, while also usually associated with data formats [34]. This is especially challenging in the IoT domain where, while manufacturers typically try to adopt standardised messaging protocols, the plethora of such established protocols with different intrinsic characteristics (e.g., RESTful HTTP, CoAP, XMP, MQTT, DPWS) and a variety of data formats (e.g., XML, JSON), which leads to a fragmented landscape.

5.2.1 PATTERN DEFINITION

Let us consider:

- $C :=$ the set of all instantiated ingredients/activities in an IoT orchestration
- $PR :=$ A set of protocols
- $C_1, C_2 \subseteq C$, where $C_1 \neq C_2$
- $C_{i\_PR} \subseteq PR :=$ protocols supported by Ci
- $SyMD :=$ Syntactic Mediator (component which connects to components with various protocols, and translates between them, such as SeMIBIoT [36])

Then, we can define the following:

Lemma 1: *If C_1, C_2 are technically interoperable and $C_{1\_PR} \cap C_{2\_PR} \neq \varnothing$ then C_1 and C_2 are directly syntactically interoperable.*

Lemma 2: *If C_1, C_2 are technically interoperable and are both directly syntactical interoperable with SyMD (Figure 20) then C_1, C_2 are indirectly syntactically interoperable*

Lemma 3: *If C_1, C_2 are directly or indirectly syntactically interoperable, then C_1, C_2 are syntactically interoperable.*

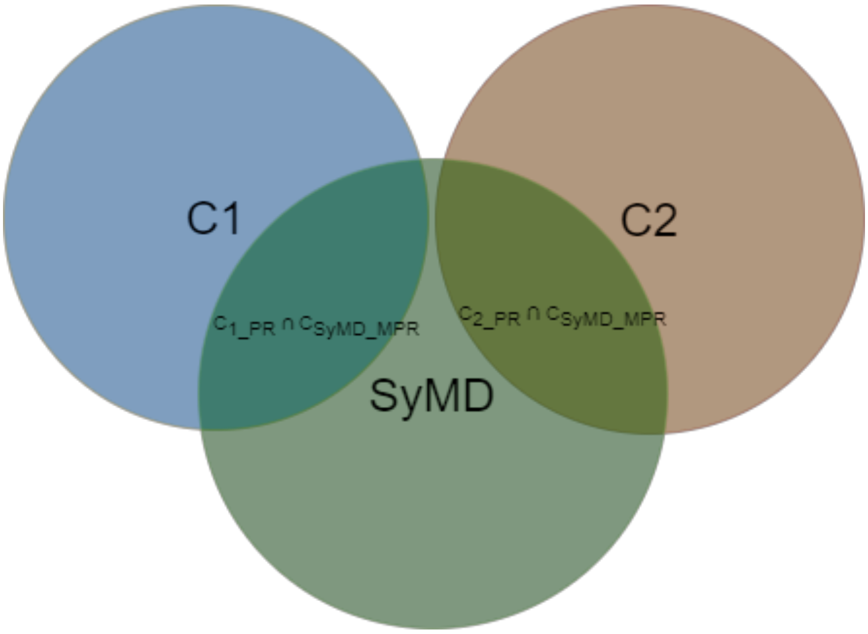


FIGURE 20. INDIRECT SYNTACTIC INTEROPERABILITY VIA SYNTACTIC MEDIATOR

5.2.2 PATTERN SPECIFICATION RULE

Considering the above, we can define the following workflow-based definition for Syntactic Interoperability between two IoT activities A1, A2 interacting with each other.

1. **<u>WF "syntactic-interoperability"</u>**
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (SyMD, (PlaceholderActivity,"syntactic mediator"))
5. Link (L1, A1, A2)
6. Link (L2, A1, SyMD)
7. Link (L3, A2, SyMD)
8. Property (conn01, L1, required, (pattern-based, pattern),"_technical-interoperability", in_transit)
9. Property (conn02, L2, required, (pattern-based, pattern),"_technical-interoperability", in_transit)
10. Property (conn03, L3, required, (pattern-based, pattern),"_technical-interoperability", in_transit)
11. Property (conn1, L1, required, (pattern-based, pattern),"_syntactic-interoperability", in_transit_ ∨ in_processing)
12. Property (conn2, L2, required, (pattern-based, pattern),"_syntactic-interoperability", in_transit_ ∨ in_processing)
13. Property (conn3, L3, required, (pattern-based, pattern),"_syntactic-interoperability", in_transit_ ∨ in_processing)
14. Property (conn4, "_syntactic-interoperability", required, (pattern-based, PR1), "_syntactic-interoperability", end_to_end)
15. **Pattern rule: (PR1: (conn01,conn1) || (conn02,conn2,con03,conn3) ➔ conn4)**

Furthermore, the above property can be encoded in machine-processable Drool-based form, as shown in Listing 2.

LISTING 2. SYNTACTIC INTEROPERABILITY VERIFICATION DROOL RULE

```
rule "Sequence Syntactic Interoperability Verification"
    when
        Placeholder($pA:=placeholderid)
        Property ($pA:=subject, category=="dataFormat", $prvaluein1:=input_value,
$prvalueout1:=output_value, satisfied==true)
        Placeholder($pB:=placeholderid)
        Property ($pB:=subject, category==" dataFormat ", $prvaluein2:=input_value,
$prvalueout2:=out_value, satisfied==true)
        Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
        $PR: Property ($sId:=subject, category==" dataFormat ", $prvalueout1==$prvaluein2,
satisfied==false)
    then
        modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

Similar to the process described for the Technical Interoperability Verification rule, each time the *"Sequence Syntactic Interoperability Verification"* rule, is fired, the *dataFormat Property* of a Sequence is verified. According to the LSH part of the rule, if:

i) the two Placeholders of a Sequence have a Property of category *dataFormat*; and
ii) the output_value of the first Placeholder is equal to the input_value of the second Placeholder ($prvalueout1==$prvaluein2)

then the RHS part of the rule verifies the corresponding Property of the Sequence in question and sets its input_value and output_value to $prvalueout1 and $prvaluein2 respectively.

A property of category *dataFormat* with input_value $prvaluein1 and output_value $prvaluout1, denotes that the corresponding component uses a specific syntax for the incoming data whose description is given by $prvaluein1 and a specific output protocol whose description is given by $prvalueout1. For example, if we have a component (PlaceholderA) which receives data in XML format, then this would be translated to a Property with category "dataFormat" and input_value "XML". Moreover, if the said component uses a different syntax for forwarding information to other components such as JSON, then the output_value of the same Property will be "JSON".

Figure 21 depicts a sequence (Sequence1) of two Placeholders. PlaceholderA has a dataFormat property with input_value "XML" and output_value "JSON". Similarly, PlaceholderB has a dataFormat property with input_value "JSON" and output_value "HTML". As step 1 shows, the dataFormat property of the Sequence1 is false, and input_value and output_value are not set. The aforementioned dataFormat properties trigger the *"Sequence Syntactic Interoperability Verification"* rule, verifying(satisfied=true) the dataFormat property of Sequence1 and assigning the appropriate values to input_value and output_value of the property (step 2).

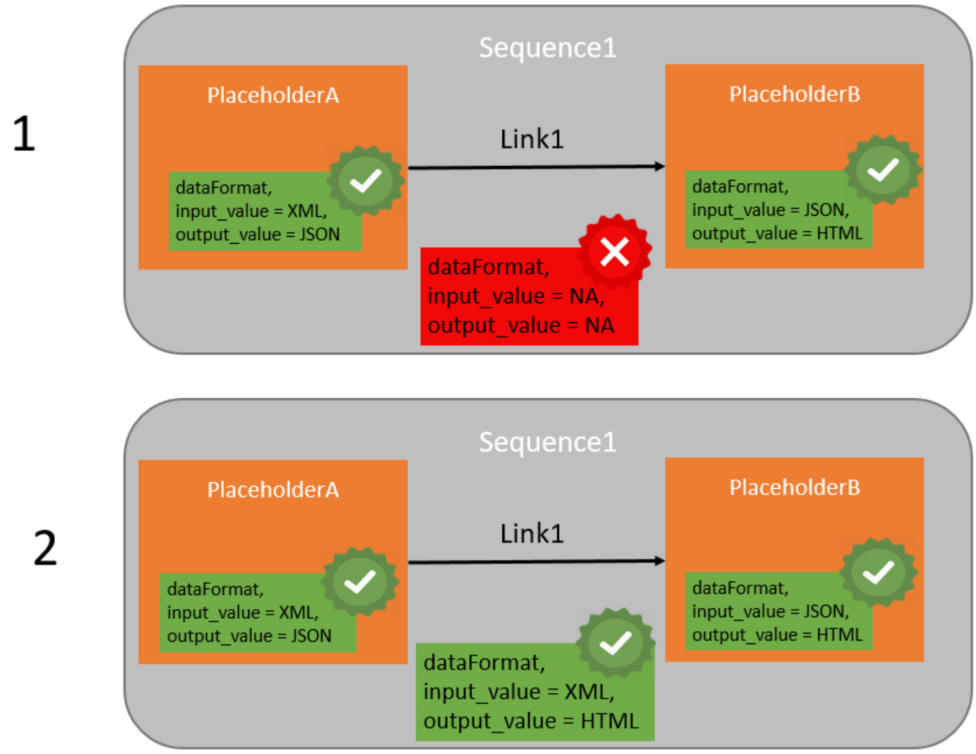


FIGURE 21. EXECUTION STEPS OF SYNTACTIC INTEROPERABILITY PATTERN

5.3 Semantic Interoperability

Interoperability on the Semantic level means that there is a common understanding between the involved systems of the meaning of the content (information) being exchanged. This means that the exchanged data have an unambiguous, shared meaning. For example, temperature units can be Fahrenheit, Celsius or Kelvin, but they express the same information which can be obtained after proper instance transformation.

5.3.1 PATTERN DEFINITION

Let us consider:

– C := the set of all instantiated components
– MDL := A set of semantic models
– $\mathrm{C_1,C_2} \subseteq \mathrm{C}$, where $\mathrm{C_1} \neq \mathrm{C_2}$
– $\mathrm{C_{i\_MDL}} \subseteq \mathrm{MDL}$:= semantic models used by Ci
– SeMD := Semantic Mediator; e.g., a Semantic Mediator as defined in Hatzivasilis et al [37], or a Semantic Information Broker [38].

Then, we can define the following:

Lemma 1: *If C_1, C_2 are syntactically interoperable and $C_{1\_MDL} \cap C_{2\_MDL} \neq \varnothing$ then C_1 and C_2 are directly semantically interoperable*

Lemma 2: *If C_1, C_2 are syntactically interoperable and are both directly semantically interoperable with SeMD (Figure 22), then C_1, C_2 are indirectly semantically interoperable*

Lemma 3: *If C_1, C_2 are directly or indirectly semantically interoperable, then C_1, C_2 are semantically interoperable.*

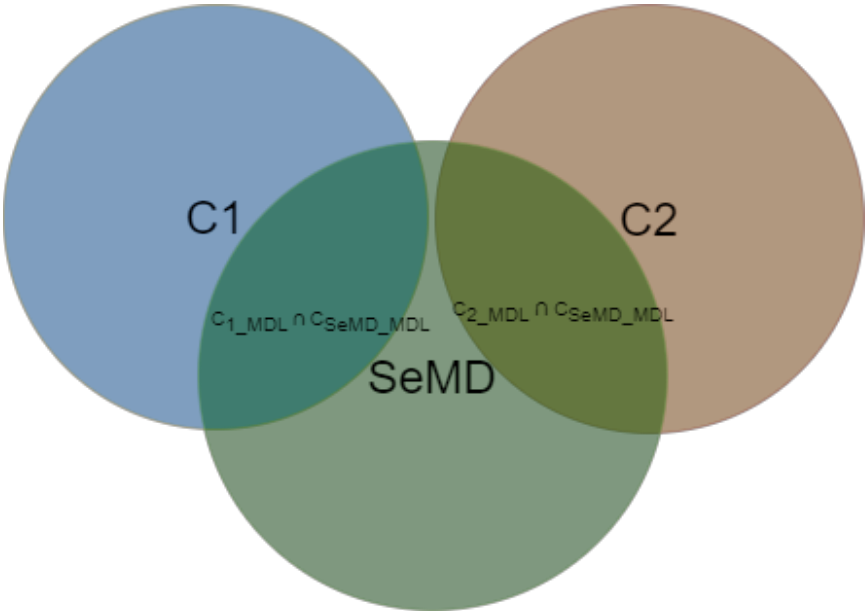


FIGURE 22. INDIRECT SEMANTIC INTEROPERABILITY VIA SEMANTIC MEDIATOR

5.3.2 PATTERN SPECIFICATION RULE

Based on the above, the workflow-based definition of semantic interoperability in the fundamental scenario of two IoT activities A1 and A2 interacting with each other, is as follows:

1. **WF "semantic-interoperability"**
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (SeMD, (PlaceholderActivity,"Semantic Broker"))
5. Link (L1, A1, A2)
6. Link (L2, A1, SeMD)
7. Link (L3, A2, SeMD)
8. Property (conn01, L1, required, (pattern-based, pattern),"_syntactic-interoperability" , in_transit_ ∨ in_processing)
9. Property (conn02, L2, required, (pattern-based, pattern),"_syntactic-interoperability" , in_transit_ ∨ in_processing)
10. Property (conn03, L3, required, (pattern-based, pattern),"_syntactic-interoperability" , in_transit_ ∨ in_processing)
11. Property (conn1, L1, required, (pattern-based, pattern),"_semantic-interoperability" , in_processing)
12. Property (conn2, L2, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
13. Property (conn3, L3, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
14. Property (conn4, "_semantic-interoperability", required, (pattern-based, PR1)," "_semantic-interoperability", end_to_end)
15. **Pattern rule: (PR1: (conn01,conn1) || (conn02,conn2,conn03,conn3) ➔ conn4)**

Moreover, we can define the semantic interoperability rule in a machine-processable Drool rule format, as show in Listing 3.

LISTING 3. SEMANTIC INTEROPERABILITY VERIFICATION DROOL RULE

```
rule "Sequence Semantic Interoperability Verification"
    when
        Placeholder($pA:=placeholderid)
        Property ($pA:=subject, category=="semantic", $prvaluein1:=input_value,
$prvalueout1:=output_value, satisfied==true)
        Placeholder($pB:=placeholderid)
        Property ($pB:=subject, category=="semantic", $prvaluein2:=input_value,
$prvalueout2:=out_value, satisfied==true)
        Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
        $PR: Property ($sId:=subject, category=="semantic", $prvalueout1==$prvaluein2,
satisfied==false)
    then
        modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

The “Sequence Semantic Interoperability Verification” rule verifies the semantic Property of a Sequence every time it is triggered. According to the LSH part of the rule, if:

i) the two Placeholders of a Sequence have a Property of category semantic, and
ii) the output_value of the first Placeholder is equal to the input_value of the second Placeholder ($prvalueout1==$prvaluein2),

then the RHS part of the rule verifies the corresponding Property of the Sequence in question and sets its input_value and output_value to $prvalueout1 and $prvaluein2 respectively.

A property of category semantic with input_value $prvaluein1 and output_value $prvaluout1, denotes that the corresponding component has a specific understanding regarding the content of the incoming data whose description is given by $prvaluein1 and a specific understanding regarding the content of the output data whose description is given by $prvalueout1. For example, if we have a component (PlaceholderA) which understands temperature in Kelvin scale, then this would be translated to a Property with category “semantic” and input_value “Kelvin”. Moreover, if the said component uses a different understanding for forwarding temperature to other components such as Celsius, then the output_value of the same Property will be “Celsius”.

Figure 23 depicts a sequence (Sequence1) of two Placeholders. PlaceholderA has a semantic property with input_value “Kelvin” and output_value “Celsius”. Similarly, PlaceholderB has a semantic property with input_value “Celsius” and output_value “Fahrenheit”. As step 1 shows, the semantic property of the Sequence1 is false and input_value and output_value are not set. The aforementioned semantic properties trigger the “Sequence Semantic Interoperability Verification” rule, verifying(satisfied=true) the semantic property of Sequence1 and assigning the appropriate values to input_value and output_value of the property (step 2).

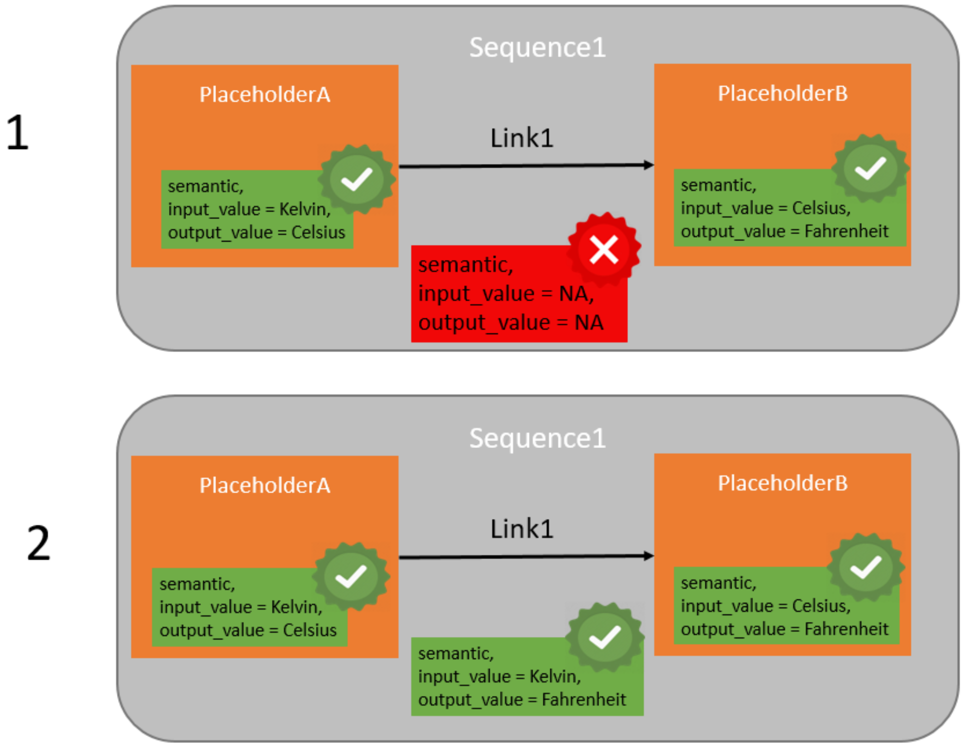


FIGURE 23. EXECUTION STEPS OF SEMANTIC INTEROPERABILITY PATTERN

5.4 Organisational Interoperability

Organisational Interoperability refers to the ability of different organisations to effectively exchange (meaningful) data, even though they may be using a variety of different information systems over widely different infrastructures. In the context of IoT and IIoT deployments, which are the focus of SEMIoTICS, organisational interoperability is mapped to cross-domain and cross-platform service integration and orchestration.

5.4.1 PATTERN DEFINITION

Let us consider:

- C := the set of all instantiated IoT platform deployments
- $CSPI$:= A set of common semantic and programming interfaces
- $C_1, C_2 \subseteq C$, where $C_1 \neq C_2$
- $C_{i\_CSPI} \subseteq CSPI$:= common semantic and programming interfaces supported by Ci
- IP := Integration Proxy; i.e., a proxy, broker or middleware, such as a platform integration gateway, management or proxy service [39] or an IoT Broker [40].

Then, we can define the following:

Lemma 1: *If C_1, C_2 are semantically interoperable and $C_{1\_CSPI} \cap C_{2\_CSPI} \neq \varnothing$ then C_1 and C_2 are directly organisationally interoperable*

Lemma 2: *If C_1, C_2 are semantically interoperable and are both directly organisationally interoperable with IP (Figure 24), then C_1, C_2 are indirectly organisationally interoperable*

***Lemma 3:** If C_1, C_2 are directly or indirectly organisationally interoperable, then C_1, C_2 are organisationally interoperable.*

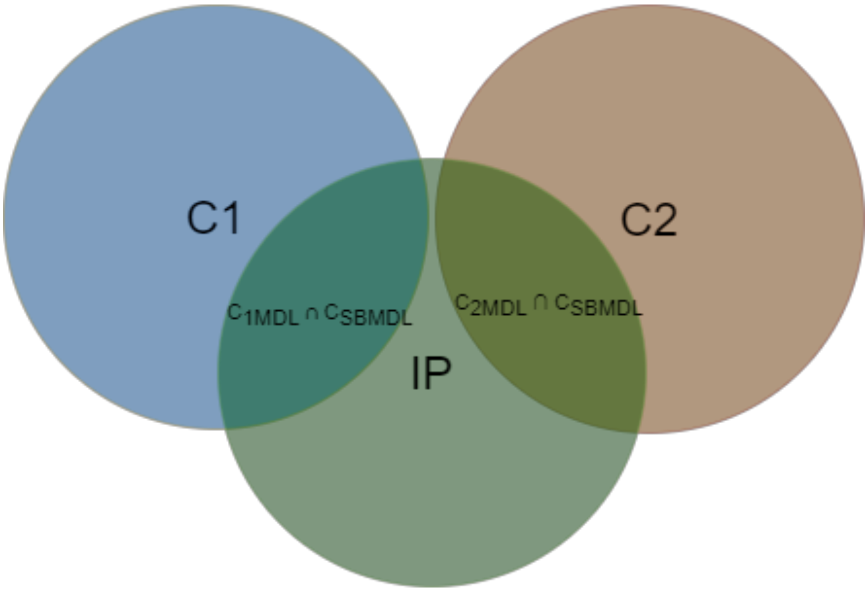


FIGURE 24. INDIRECT ORGANISATIONAL INTEROPERABILITY VIA INTEGRATION PROXY

5.4.2 PATTERN SPECIFICATION RULE

Based on the above, we can derive the following workflow-based definition of organisational interoperability for the fundamental case of two IoT platform deployments, A1 & A2, interacting with each other:

1. **WF "organisational-interoperability"**
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (IP, (PlaceholderActivity,"Integration Proxy"))
5. Link (L1, A1, A2)
6. Link (L2, A1, IP)
7. Link (L3, A2, IP)
8. Property (conn01, L1, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
9. Property (conn02, L2, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
10. Property (conn03, L3, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
11. Property (conn1, L1, required, (pattern-based, pattern),"_organisational-interoperability" , in_transit_ ∨ in_processing)
12. Property (conn2, L2, required, (pattern-based, pattern),"_organisational-interoperability" , in_transit_ ∨ in_processing)
13. Property (conn3, L3, required, (pattern-based, pattern),"_organisational -interoperability" , in_transit_ ∨ in_processing)
14. Property (conn4, "_organisational-interoperability", required, (pattern-based, PR1)," semantic-interoperability", end_to_end)
15. **Pattern rule: (PR1: (conn01,conn1) || (conn02,conn2,conn03,conn3) ➔ conn4)**

Moreover, we can specify organisational interoperability via the following machine-processable Drools rule, as shown in Listing 4.

LISTING 4. ORGANIZATIONAL INTEROPERABILITY VERIFICATION DROOL RULE

```
rule "Sequence Organizational Interoperability Verification"
    when
        Placeholder($pA:=placeholderid)
        Property ($pA:=subject, category=="organizational", $prvaluein1:=input_value,
$prvalueout1:=output_value, satisfied==true)
        Placeholder($pB:=placeholderid)
        Property ($pB:=subject, category=="organizational", $prvaluein2:=input_value,
$prvalueout2:=out_value, satisfied==true)
        Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
        $PR: Property ($sId:=subject, category=="organizational", $prvalueout1==$prvaluein2,
satisfied==false)
    then
        modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

The “Sequence Organizational Interoperability Verification” rule verifies the organizational Property of a Sequence every time it is triggered. According to the LSH part of the rule, if:

i) the two Placeholders of a Sequence have a Property of category organizational, and
ii) the output_value of the first Placeholder is equal to the input_value of the second Placeholder ($prvalueout1==$prvaluein2),

then the RHS part of the rule verifies the corresponding Property of the Sequence in question and sets its input_value and output_value to $prvalueout1 and $prvaluein2 respectively. It is mentioned that Placeholders in this case refer to processes/workflows of different IoT platforms and/or organisations (e.g., interactions between a SEMIoTICS and FI-WARE deployment, or even between two SEMIoTICS deployments belonging to different organisations).

As mentioned above, organizational interoperability requires the existence of all other more basic forms of interoperability (semantic, syntactic, technological). Then, the verification focuses on the compatibility of organizational processes and workflows. A property of category organizational with input_value $prvaluein1 and output_value $prvaluout1, denotes that the corresponding process has a specific understanding regarding the content of the incoming data whose description is given by $prvaluein1 and a specific understanding regarding the content of the output data whose description is given by $prvalueout1.

Nevertheless, this is not limited to the semantics, as in section 5.3 above, but also from a workflow and organisational/business process context. For example, if we have a process (PlaceholderA) which expects inputs from a specific organisational activity (e.g., new IoT asset registration request), then this would be translated to a Property with category “organizational” and input_value “Activity”. Moreover, if this process uses different means for its output (e.g., needs to interact with an external service residing in another organisation/IoT platform to retrieve assets available there), then the output_value of the same Property will be “ExtService”.

To visualise this, Figure 25 depicts a sequence (Sequence1) of two Placeholders. PlaceholderA, residing in “IoT Platform A” has an organizational property with input_value “Activity” and output_value “ExtService”. Similarly, PlaceholderB has an organizational property with input_value “ServiceReq” (to denote that process PlaceholderB residing in IoT Platform B expects requests from external parties) and output_value “Workflow” (to denote this external request is then mapped to an internal workflow; e.g., for local asset discover). As step 1 shows, the organizational property of the Sequence1 is false, and input_value and output_value are not set. The aforementioned organizational properties trigger the “Sequence Organizational Interoperability Verification” rule, verifying (“satisfied=true”) the organizational property of Sequence1 and assigning the appropriate values to input_value and output_value of the property (step 2).

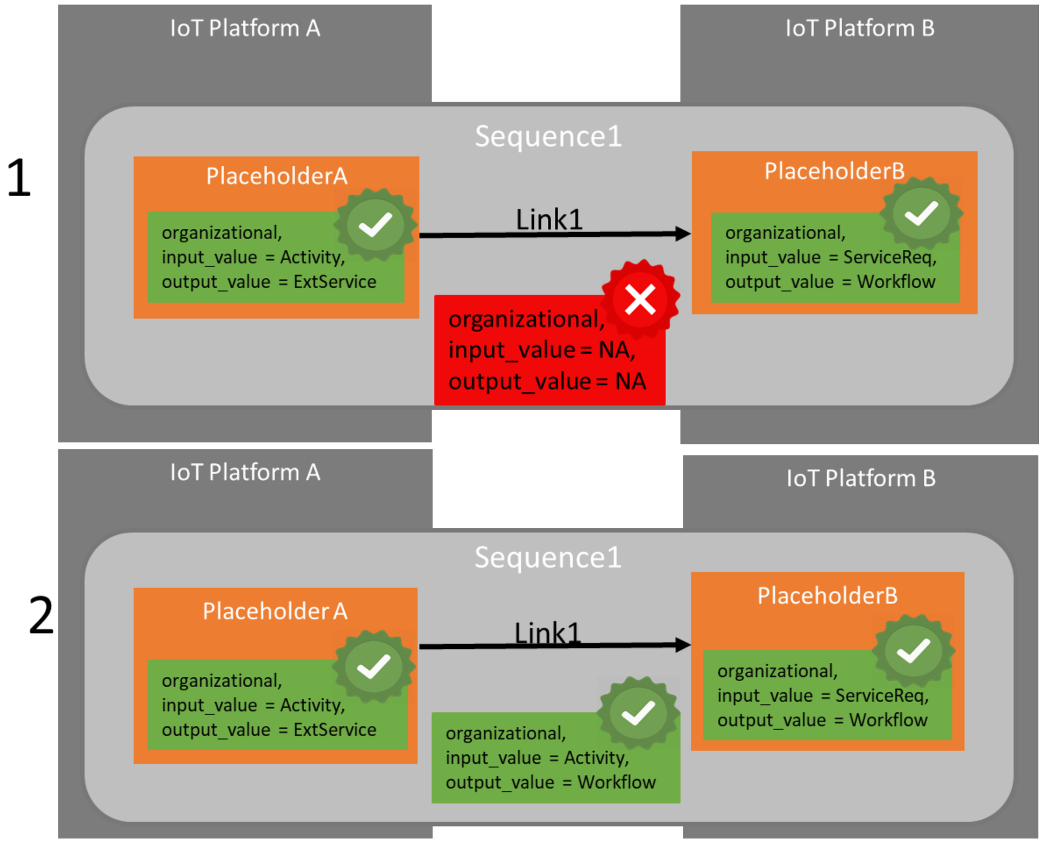


FIGURE 25. EXECUTION STEPS OF ORGANISATIONAL INTEROPERABILITY PATTERN

5.5 E2E Interoperability Within the SEMIoTICS platform

Until now, we have described how the individual interoperability types (i.e., Technical, Syntactic, Semantic and Organisational) are defined and verified for the fundamental case of a sequence of two components/activities. Nevertheless, using these rules the E2E properties of these simple as well as more complex orchestrations can be verified. More specifically, using the process defined within D4.1 (i.e., leveraging the SEMIoTICS system model and the associated activity composition and decomposition rules defined in said deliverable), more complex cases can also be verified using these fundamental property rules.

Moreover, and specifically for interoperability, if between an orchestration of two of more components/activities within a given IoT platform (e.g., a deployed instance of SEMIoTICS) we can verify that the three properties of Technical, Syntactic and Semantic Interoperability all hold, then we can claim that E2E Interoperability also holds for the given orchestration within the given IoT platform. In other words, the Syntactic Interoperability property, as defined in section 5.2, is the fundamental case of E2E Interoperability within an IoT platform, since if it evaluates as true, the verifications of the other two underlying interoperability types (Technical and Syntactic) also evaluate as true, and thus there is full interoperability between the interacting entities. Consequently, in the more complex orchestrations, and using the process already described in D4.1 (see section 2.4 and 4.4), if all of these three interoperability properties are verified for all parts of a given orchestration, then the E2E Interoperability within the platform (referred to as the *“E2E_WP_Interoperability”* Property) is also verified. This is what the rule in Listing 5 depicts; Technical, Syntactic and Semantic properties are the prerequisites (LSH part of the rule) for the overall Interoperability Property to hold (RSH part of the rule).

LISTING 5. END-TO-END INTEROPERABILITY WITHIN PLATFORM VERIFICATION DROOL RULE

```
rule "Sequence Interoperability"
    when
        Sequence($sId:=placeholderid)
        $PR1: Property ($sId:=subject, category=="technical", satisfied==true)
        $PR2: Property ($sId:=subject, category=="dataFormat", satisfied==true)
        $PR3: Property ($sId:=subject, category=="semantic", satisfied==true)
        $PR4: Property ($sId:=subject, category=="E2E_WP_Interoperability", satisfied==false)
    then
        modify($PR4){satisfied=true};
end
```

Elaborating on the above, Figure 26 depicts a sequence (Sequence1) whereby the technical, dataFormat (i.e. syntactic) and semantic properties have already been verified (step 1). The aforementioned interoperability properties trigger the "Sequence Interoperability Verification" rule, verifying (satisfied=true) the E2E interoperability property of Sequence1 as instantiated within the bounds of a given IoT platform (step 2).

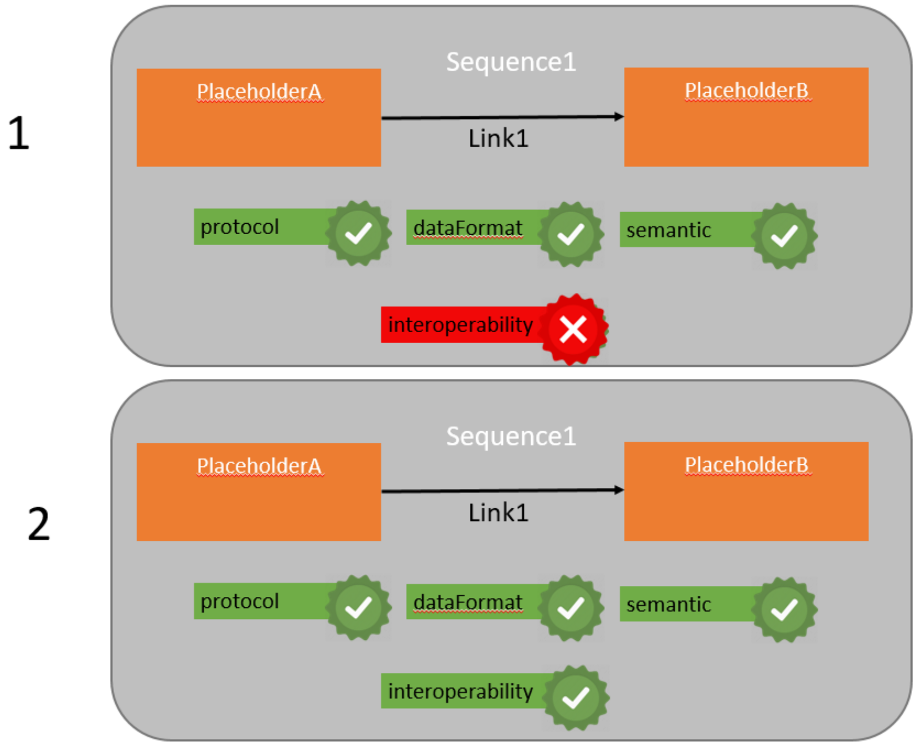


FIGURE 26. EXECUTION STEPS OF E2E INTEROPERABILITY WITHIN PLATFORM PATTERN

5.6 E2E Interoperability Across IoT Platforms

Similar to the reasoning detailed in section 5.5 above regarding the Semantic Interoperability and E2E Interoperability verification within the SEMIoTICS platform, the fundamental case of E2E Interoperability across platforms (e.g., across a SEMIoTICS and a FI-WARE deployment) is covered by the definition of the Organisational Interoperability, as presented in section 5.4.The additional requirement in this case is the satisfaction of the Organisational Interoperability property between the two interacting entities across different IoT platforms, which in turn requires the verification of the Semantic, Syntactic and Technical Interoperability properties in order to be possible. This is shown in

Listing 6, with the property being referred to "*E2E_AP_Interoperability*".

LISTING 6. END-TO-END INTEROPERABILITY ACROSS PLATFORM VERIFICAITON DROOL RULE

```
rule "Sequence Interoperability"
    when
        Sequence($sId:=placeholderid)
        $PR1: Property ($sId:=subject, category=="protocol", satisfied==true)
        $PR2: Property ($sId:=subject, category=="dataFormat", satisfied==true)
        $PR3: Property ($sId:=subject, category=="semantic", satisfied==true)
        $PR4: Property ($sId:=subject, category=="organisational", satisfied==true)
        $PR5: Property ($sId:=subject, category=="E2E_AP_Interoperability", satisfied==false)
    then
        modify($PR5){satisfied=true};
end
```

Figure 27 depicts a sequence (Sequence1) of two Placeholders. In this case, both Placeholders refer to IoT platforms. As step 1 shows, E2E interoperability across IoT platforms property of the Sequence1 is false, and input_value and output_value are not set. However, the technical, syntactic, semantic and organizational properties hold for Sequence1. The aforementioned properties trigger the "Sequence Interoperability Across IoT Platforms" rule, verifying(satisfied=true) the semantic property of Sequence1 (step 2).

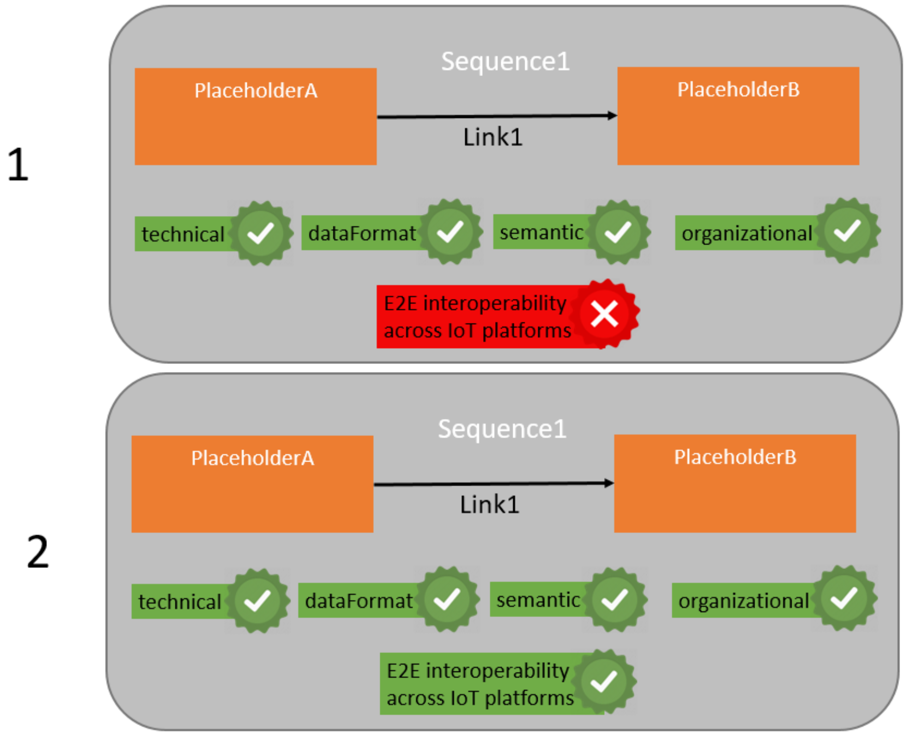


FIGURE 27. EXECUTION STEPS OF E2E INTEROPERABILITY ACROSS PLATFORMS PATTERN

Considering more complex orchestrations, in the context of potential SEMIoTICS applications, and as detailed in section 2 herein and section 2.4 of deliverable D4.1, end-to-end interoperability should cover heterogeneous cases of cross-platform and scale connectivity. More specifically, based on previous work [41] carried out in the BIG IoT project, the cases of interoperability across IoT platforms, as sketched in Figure 28, include:

I. **Cross platform** – applications or services access resources from multiple platforms though common interfaces.
II. **Platform-scale independence** – integrates the resources from platforms at different scale in the way that application can uniformly aggregate information for different scale platforms (cloud-, fog-, device-level).
III. **Platform independence** – refers to distinct platforms that implement the same functionality in the way that ensures that a single driver application can interoperate with both platforms in a uniform manner without requiring any changes.
IV. **Cross application domain** – refers to uniform access to information from platforms that process data from different domains.
V. **Higher-level service facades** – services can also interact themselves through common API. Therefore, a single application can interact with two platforms to create value-added operations.

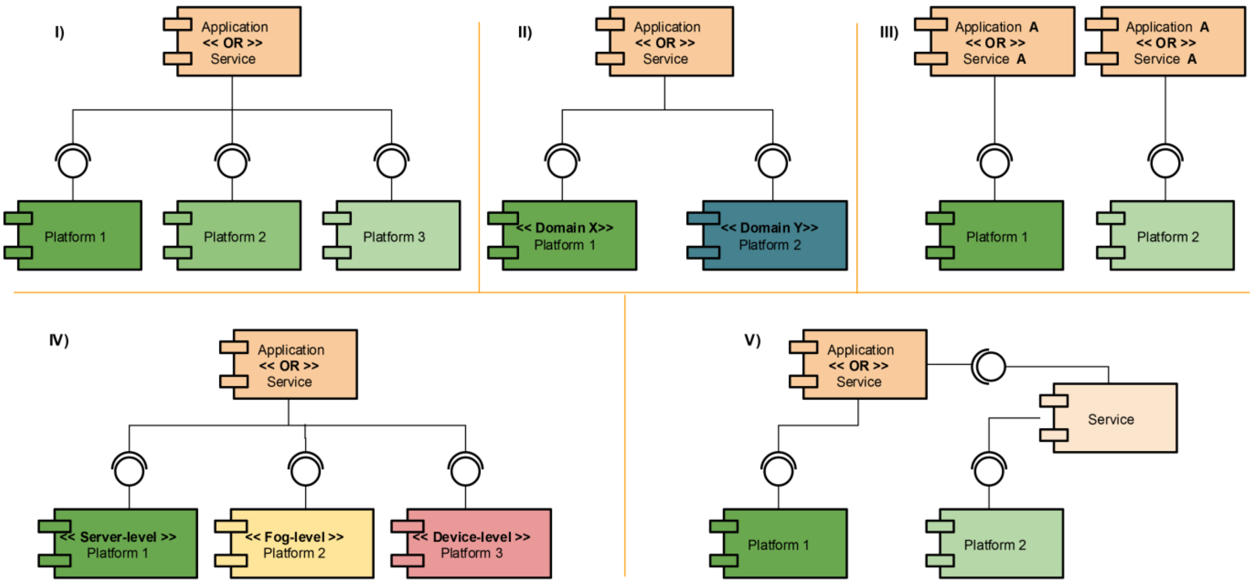


FIGURE 28. FIVE CASES OF INTEROPERABILITY ACROSS PLATFORMS [41]

Each of the five cases described above and depicted in Figure 28 is supported through the SEMIoTICS end-to-end semantic interoperability mechanisms and the network mechanisms detailed herein.

More specifically, and as mentioned in the case of the Interoperability with platforms (section 5.5), all of these cases can be covered by the fundamental *E2E_AP_Interoperability* property verification rule definition, through decomposition of said complex scenarios to one or more verifications of instances of these fundamental property.

This process of composition and decomposition of complex workflows and the verification of their properties has been defined in the context of Task 4.1 ("Architectural SPDI Patterns") and detailed in the associated outputs of the Task (D4.1, with the final version appearing in D4.8).

6 PATTERN-DRIVEN NBI IN USE – AN ENABLER OF KEY SEMIOTICS FEATURES

6.1 Enabling IoT Orchestrations with End-to-End Semantic Interoperability, SPDI and QoS Guarantees

The Pattern-driven NBI and associated mechanisms developed in the context of T3.4 and presented in the previous sections are essential enablers in the implementation of a number of key features in SEMIoTICS. These include the provision of E2E semantic interoperability, while also allowing the specification and runtime verification of SPDI properties required of IoT applications and their orchestrations, and the use of the SEMIoTICS network features from external entities (e.g., other IoT platforms). These are detailed in the subsections that follow.

The SDN NBI presented herein is an important enabler for the SEMIoTICS End-to-End semantic interoperability capabilities, ensuring interoperability (i.e., the "I" in "SPDI") from the application definition all the way through to the execution at runtime, while also spanning all layers of the SEMIoTICS deployment. This relies on four levels of abstraction and accordingly three steps of transformation between them, as shown in Figure 29 and detailed below:

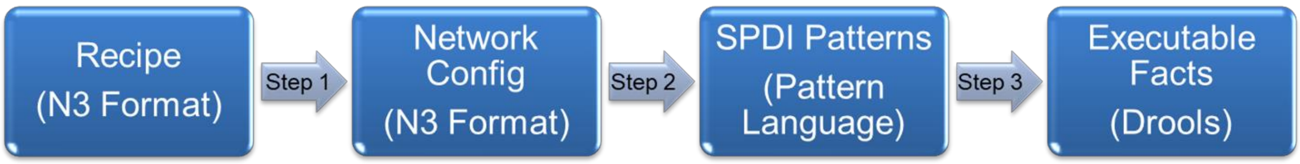


FIGURE 29. TRANSLATIONS FROM RECIPES TO EXECUTABLE FACTS

– **Step 1: From the Recipe language to a semantically-rich network model** (see Figure 30). As mentioned in 3.4.2, SEMIoTICS adopts a user-friendly and semantically rich approach to IoT workflow composition. In this first translation step, workflows created using this Recipe approach (left side of Figure 30) are transformed to the SDN model (right side of Figure 30), which is inspired by the data structures used by the northbound interfaces of SDN controllers, such as the ones defined by [43]. Through this scheme for expressing application-level SPDI and QoS constraints as a collection of semantic rules, and by including these rules in the triple store together with the semantic models, the application level constraints are automatically translated by the semantic reasoner of the triple store into instances of the lower-level SDN model. These instances can then be submitted as configurations to the SDN Pattern Engine, through its pattern-driven NBI. More details on this process are presented in D4.4 ("Semantic Interoperability Mechanisms for IoT (first draft)", as part of the Task 4.4 ("End-to-End Semantic Interoperability) efforts.

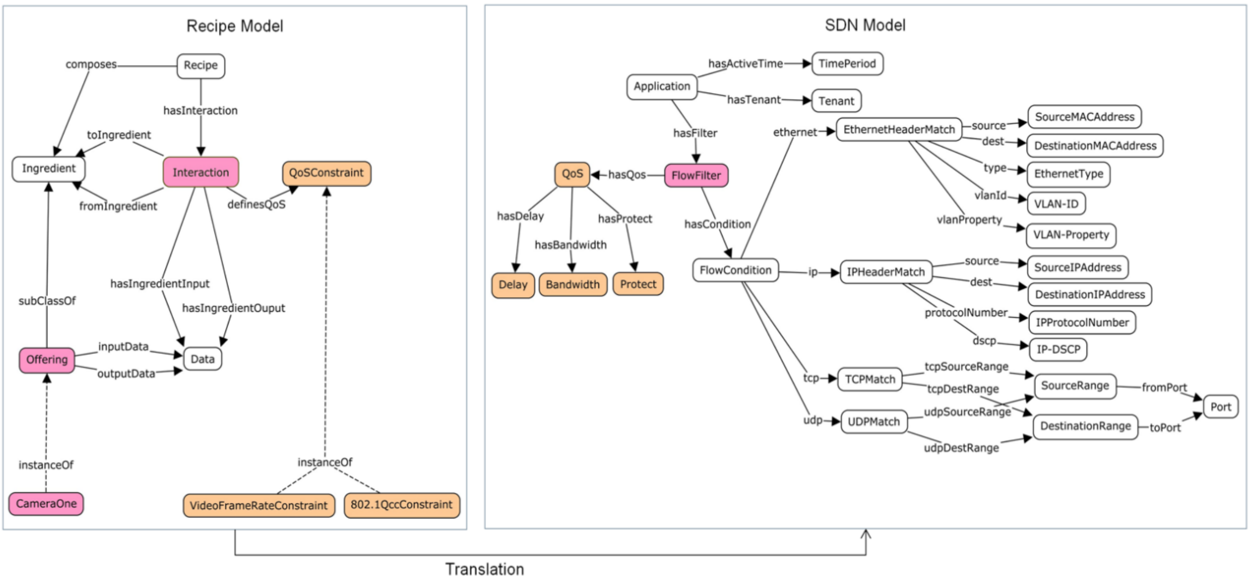


FIGURE 30. TRANSORMATION FROM RECIPE TO NETWORK CONFIGURATION [42]

– **Step 2**: **From the semantically-rich network model to workflow-based definitions leveraging the SEMIoTICS Pattern Language**. The IoT workflows defined and the results of the translation to the network model are then translated to follow the workflow-based IoT Orchestration System Model, leveraging the associated Pattern Language, using the process detailed and constructs specified in D4.1.
– **Step 3**: **From the SEMIoTICS pattern language to Drools executable rules and facts.** This last step in the process involves translation from the workflows and their properties defined using the pattern language into Drools facts and rules. Again, as detailed in D4.1, this allows the definition in a machine-processable format, enabling reasoning engines (referred to as Pattern Engines) deployed at all layers of the SEMIoTICS, to independently verify the desired SPDI and QoS properties hold, and trigger adaptations if needed (thus acting as key enablers in the multi-layered embedded intelligence and semi-autonomous adaptation aspects of SEMIoTICS).

The key components involved on implementing on this End-to-End Interoperability concept are depicted in Figure 31 (semantic components in red, pattern ones in green), while more details will be presented in D4.11 (“Semantic Interoperability Mechanisms for IoT (final)”, as part of the Task 4.4 (“End-to-End Semantic Interoperability) efforts.

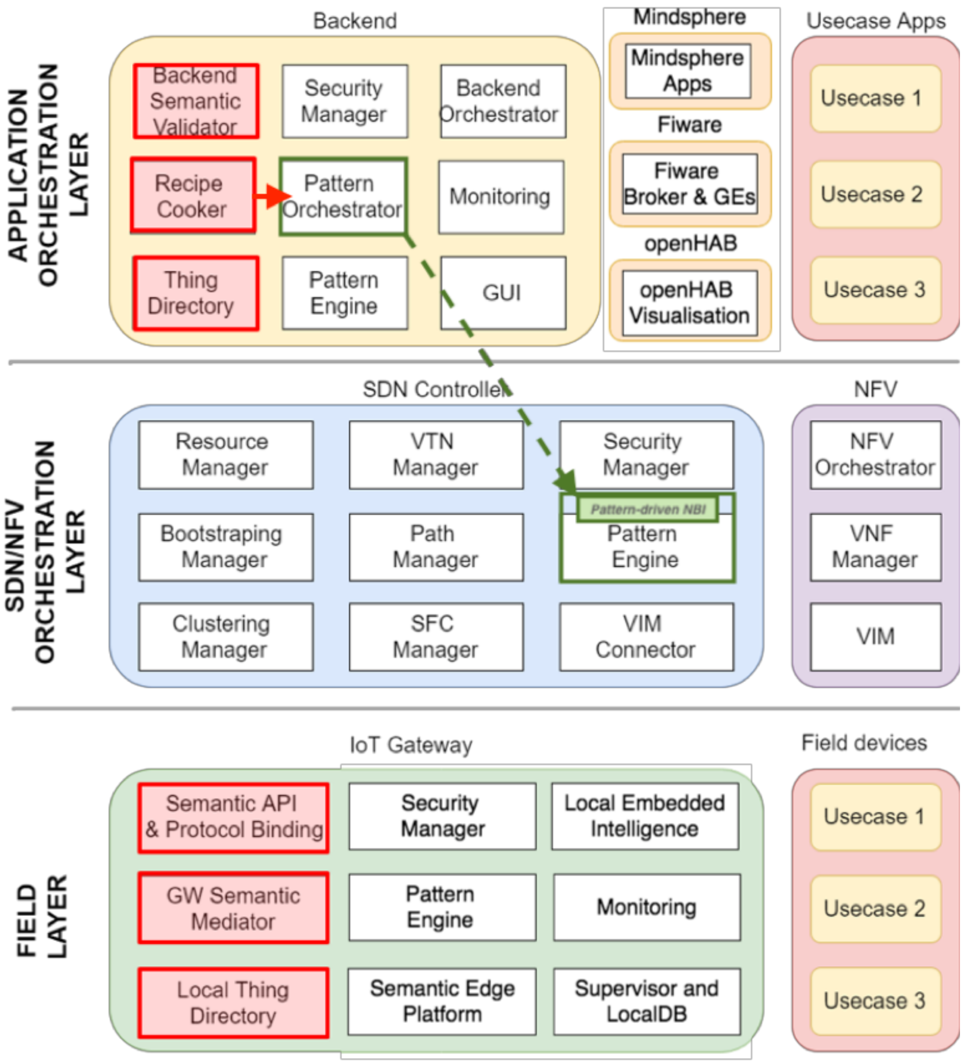


FIGURE 31. PATTERN-DRIVEN NBI AS AN ENABLER OF END-TO-END SEMANTIC INTEROPERABILITY

Through the above process and components, and leveraging the pattern-driven orchestration definition, property verification and associated adaptations (via the interoperability-related patterns rules, as presented in section 5) the end-to-end interoperability is ensured at design-time (at the Recipe specification phase), at deployment-type (when the orchestrations are instantiated), as well as at runtime (via the interoperability properties' verification through pattern reasoning).

From an IoT Orchestration definition perspective, as it is shown in Figure 32 below, the user defines the recipe (i.e., the application flow) and specifies the expected capabilities of ingredients, such as input and output data types. The Recipe Cooker tool is utilized for this specification. After this step the instantiation of the recipe takes place. "Instantiation" refers to the replacement of abstract components with concrete available components. The recipe is then deployed. The recipe deployment triggers the transmission of the recipe instance to the Pattern Translation Middleware, which is used for the translation of the network configuration and details into SPDI patterns. It converts the network configuration defined in N3 into the Extended Backus-Naur Form (EBNF) grammar defined in the ANTLR (https://www.antlr.org/) format. What follows is the description of the recipe instance in terms of the pattern language. Translation from the JSON format into the pattern language is realized through a series of graph transformation steps, where nodes from the recipe are collapsed into an orchestration of the pattern language (Sequence, Merge, etc.), until the graph has only a single node left.

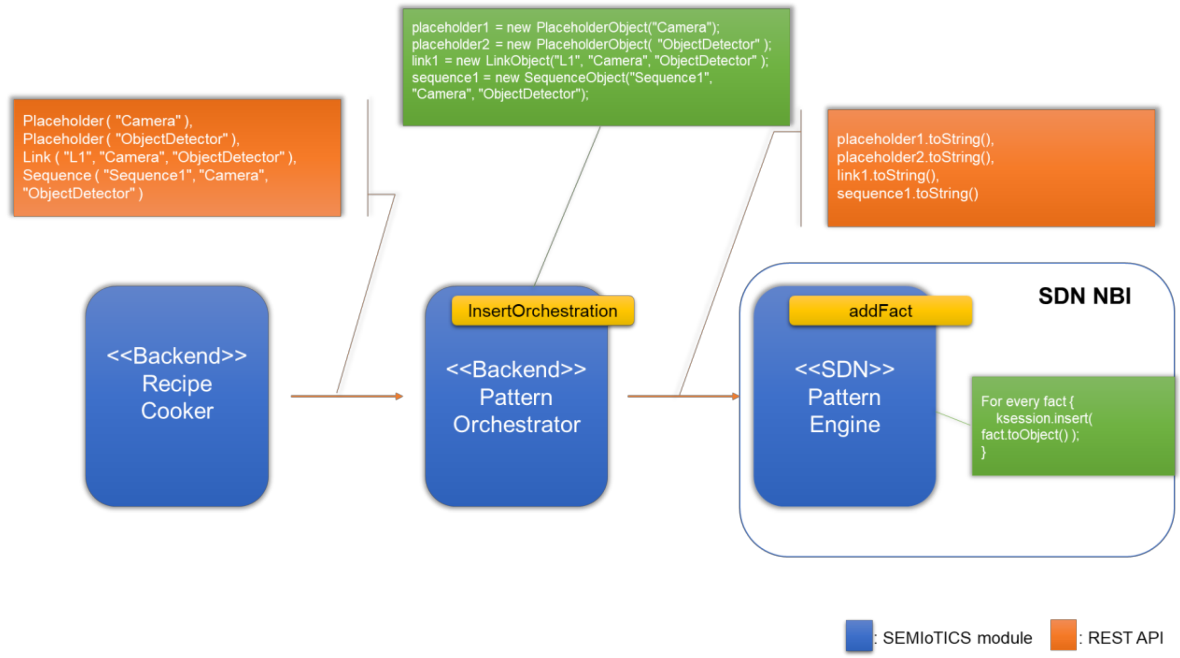


FIGURE 32. PATTERN-DRIVEN IOT ORCHESTRATIONS – KEY INTERFACES AND COMPONENT INTERACTIONS

In sequence, the recipe expressed as the pattern is transmitted to Pattern Orchestrator. For that purpose, a POST service request has been developed named insertRecipe. Pattern Orchestrator receives a request from Recipe Cooker, which includes a recipe description in JSON format. Such a request is depicted in Figure 33. Under “recipeID” a unique string that acts as an identifier is provided, while under “recipe” label lays the recipe description itself. The recipe instance depicted in Figure 33 is very simple and consists of two software components that are placed in sequence, which means that the output of the former is consumed as input by the latter.

```
{
    "recipeID": "Demo2WF1",
    "recipe": "Softwarecomponent(\"f5a474bd4cd818\", \"0\", \"piB\"), Softwarecomponent(\"f86e905a107f1\", \"0\", \"piB\"),
        Sequence(\"Seq0\", \"f86e905a107f1\", \"f5a474bd4cd818\", \"Link0\")"
}
```

FIGURE 33. INSERT RECIPE REQUEST

An indicative scenario that highlights the above procedure and underlines the importance of the flexibility offered by the pattern-driven NBI is presented in D5.3 - Section 4.1 “Use Case 1 demonstrator”, considering the scenario of oil leakage detection IIoT application in a wind park environment (see Figure 34). In the defined use case application, and aiming to focusing on the network aspects, while maintaining the high-level abstractions needed for user-friendliness, a “Network Link” node enables definition of SPDI and QoS constraints (e.g., encryption, minimum bandwidth, latency) and the whole orchestration specification (a “Recipe”). In the specific example, QoS constraints are translated into the SEMIoTICS pattern language and sent to Pattern Orchestrator. From the latter, the information is relayed to the pattern-driven NBI. A full analysis of this scenario is outside the scope of this deliverable, since it only focuses on the pattern-driven interface exposed at the network layer; a detailed description of the scenario will be provided in D4.8 ("SEMIoTICS SPDI Patterns (final)").

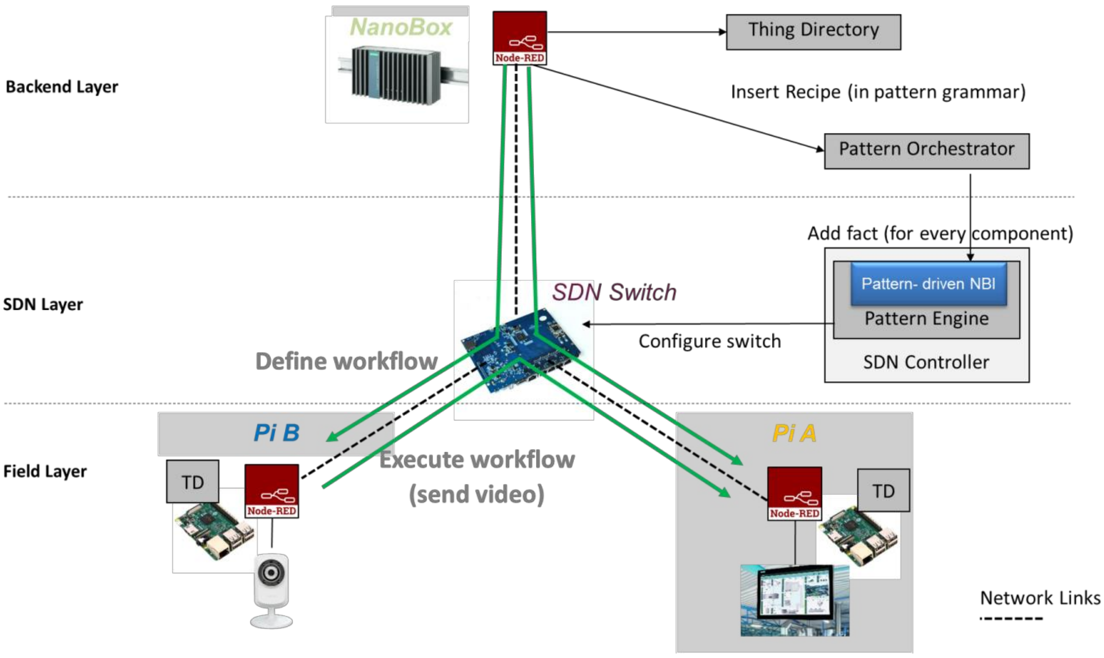


FIGURE 34. OIL LEAKAGE DETECTION APPLICATION SCENARIO (UC1)

6.2 Interfacing with External IoT Platforms

As already described in D5.3 section 3.3.5, there are two directions regarding interoperability with external IoT platforms: i) interactions from SEMIoTICS towards external platforms (e.g., a SEMIoTICS service accessing a components of a FI-WARE deployment), and; ii) interactions from external IoT platforms towards SEMIoTICS (e.g., a service running on a FI-WARE deployment accessing SEMIoTICS assets).

The direction of focus in the context of this deliverable is the one whereby information is given from the SEMIoTICS framework to an external platform. A fundamental concept in this process is providing exposed interfaces of key selected components of the SEMIoTICS framework that can be consumed by external parties, thus being able to leverage the SEMIoTICS infrastructure (e.g., specific components at the various layers and/or services) and integrate them into their workflows.

A key component in this regard is the Pattern Orchestrator residing at the SEMIoTICS backend. This component, as described in Section 4 (and presented in more detail in deliverable D4.1, along with all pattern-related components), exposes interfaces that allow the definition of IoT Orchestrations and the required SPDI (and QoS, where needed) properties for said orchestrations. Then, through its interaction with and management of the pattern engines residing at the various layers of the SEMIoTICS deployment, the Pattern Orchestrator can offer a real-time view of the status of the orchestrations in terms of the SPDI properties, as well as inform about potential adaptation that had to be made to the orchestrations to retain these desired properties.

The use of these exposed interfaces of the Pattern Orchestrator and the features they enable are demonstrated by its integration with the Recipes approach for user-friendly IoT Orchestration definition (see Sections 3.4.2 and 6.1 above, as well as the detailed description of this integration in D4.1). As the Recipe Cooker interfaces with (and leverages the features of) the Pattern Orchestrator to deploy its Recipes and monitor the desired SPDI and QoS properties, any external IoT platform may utilize the same mechanisms in order to integrate

SEMIoTICS into its workflows, while also making use of the SPDI and QoS guarantees that the pattern-driven approach provides.

The Pattern Orchestrator is the preferred entry point for interaction with external platforms, instead e.g., of exposing directly the pattern driven NBI residing at the network layer, as the Pattern Orchestrator is in the unique position of communicating with all three layers of the SEMIoTICS architecture, and maintaining a global, real-time view of the SPDI and QoS properties' status and the associated pattern components at all layers (see Figure 35). Moreover, it offers a level of abstraction that allows for controlled interactions with the underlying SEMIoTICS components, which is important both from a security perspective (e.g., avoiding malicious use of the SEMIoTICS network or other assets) but also from a dependability one (e.g., avoiding unpredicted interactions with the SEMIoTICS components from a third party that might affect the operation of other applications relying on that).

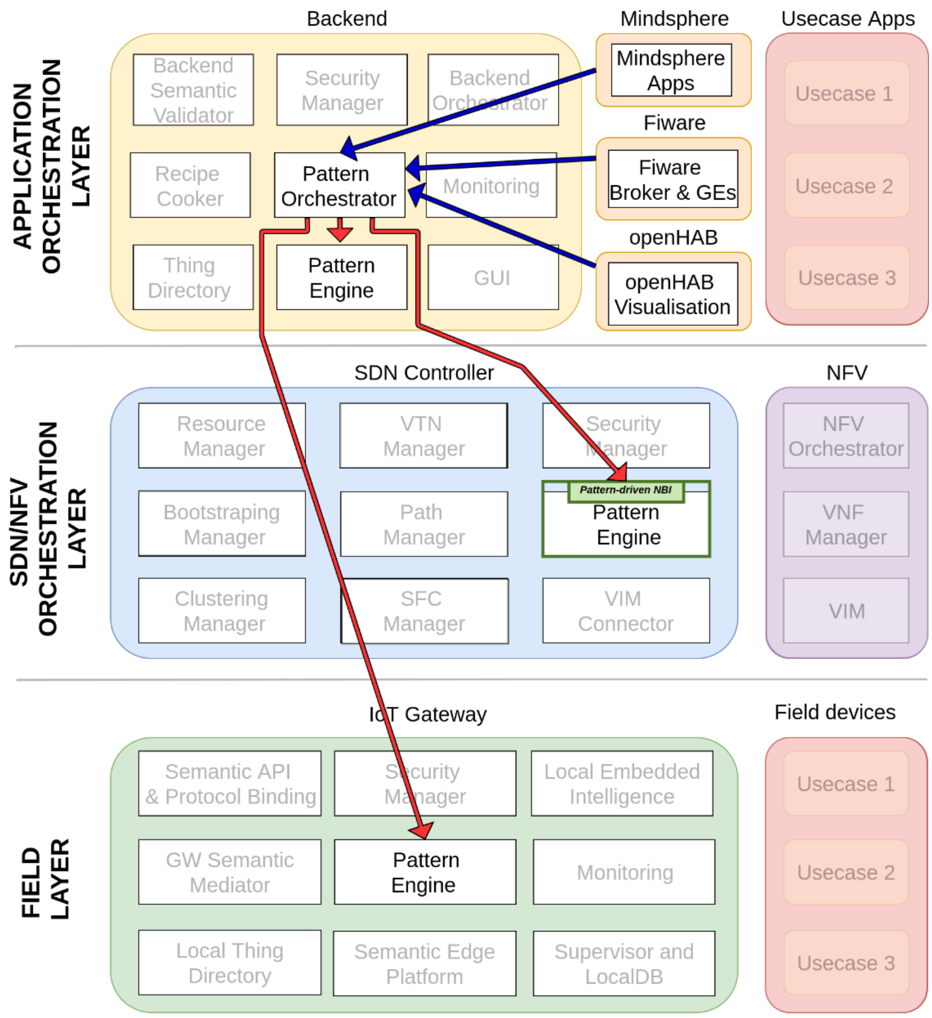


FIGURE 35. INTERFACING WITH EXTERNAL IOT PLATFORMS

7 CONCLUSION

This deliverable presented the design approach followed for the development of the SEMIoTICS network programming interfaces exposed to enable the deployment of network services from all the framework's layers and its seamless interaction with IoT applications, towards addressing the corresponding objective of WP3 dictating the need to *"...develop and offer adaptable and dynamic networking services to client IoT applications"*.

The considerations and requirements presented in Section 1 are accomplished through the adoption of semantically rich network interfacing capabilities throughout the SEMIoTICS framework, including the SBI-focused aspects covered in Task 3.1 ("Software defined Aggregation, Orchestration and cloud networks"), as documented in D3.1 ("Software defined programmability for IoT devices (first draft)"), the NBI-focused work carried out in the context of Task 3.2 ("IIoT Network Function Virtualization"), as documented in D3.2 ("Network Functions Virtualization for IoT (1st draft)"), and the pattern-driven NBI which is designed in the context of Task 3.4 ("Task 3.4 – Network-level semantic Interoperability") and is detailed within this deliverable.

SEMIoTICS' network-level semantic interoperability are enabled via the adoption of a set of key enabling technologies and implemented interfaces (as documented in sections 3 and 4 above, respectively), along with a full set of interoperability patterns (section 5) and the semantic descriptions developed in the context of Task 3.3 ("Semantics-based bootstrapping & interfacing") and documented in D3.3 ("Bootstrapping and interfacing SEMIoTICS field level devices (1st draft)"). The semantic descriptions in specific, in tandem with the architectural SPDI patterns (see Task 4.1 "Architectural SPDI Patterns", and D4.1 "SEMIoTICS SPDI Patterns (first draft)"), form the core of the SEMIoTICS Network-level semantic interoperability and SPDI-driven monitoring and adaptation capabilities.

The SPDI properties also drive the pattern-driven NBI present on the SEMIoTICS' SDN controller, the design of which is presented herein. Via this interface, SPDI patterns, through translation from Recipes which describe the high level IoT service orchestrations and their requirements and desired properties (see D4.1), are used to define the operation of the SEMIoTICS network layer and its interactions with the IoT applications that may run on top (i.e. at the backend or external IoT applications). Therefore, the interoperability and the SPDI properties of the IoT/IIoT deployments can be defined at design time, as well as verified at runtime, triggering adaptations, if needed, at the network layer.

Moreover, through well-defined and open interfaces at the SEMIoTICS backend (and more specifically, the Pattern Orchestrator component), external IoT platforms can also leverage these features and integrate SEMIoTICS assets and services into their workflows, with the SPDI and QoS guarantees that the framework can provide.

While Task 3.4 concludes with the delivery of D3.10 (i.e., at M26 of the project), work will continue on refining the pattern-driven network and semantic interoperability capabilities described herein in terms of their integration with other SEMIoTICS components (in the context of Tasks 3.5 ("Implementation of Field-level middleware & networking toolbox"), 4.6 ("Implementation of SEMIoTICS backend API") and 5.2 ("Software system integration"), as well as the demonstration of the integrated solution on the context of the project's use cases Tasks 5.4-5.7.

REFERENCES

[1] ETSI (2013) 'Network Functions Virtualisation (NFV); Virtualisation Requirements', *Etsi Gs Nfv 004 V1.1.1*, 1(10), pp. 1–17. doi: DGS/NFV-0011.

[2] Hatzivasilis, G., I. Askoxylakis, G. Alexandris, D. Anicic, A. Broring, V. Kulkarni, K. Fysarakis, and G. Spanoudakis. (2018) 'The Interoperability of Things: Interoperable solutions as an enabler for IoT and Web 3.0', IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, CAMAD*, 2018– (September). doi: 10.1109/CAMAD.2018.8514952.

[3] 'Virtual Network Functions Architecture' (2014), 1, pp. 1–93.

[4] ETSI (2016) *Open Source MANO (OSM) RO Northbound Interface*. Available at: https://osm.etsi.org/wikipub/index.php/RO_Northbound_Interface.

[5] W3C (2017) 'Web of Things (WoT) Architecture'. Available at: https://www.w3.org/TR/2017/WD-wot-architecture-20170914/.

[6] W3C (2017) 'Web of Things (WoT) Things Description'. Available at: https://w3c.github.io/wot-thing-description/.

[7] Thuluva, A.S., A. Bröring, G.P. Medagoda Hettige Don, D. Anicic & J. Seeger (2017): Recipes for IoT Applications. Proceedings of the 7th International Conference on the Internet of Things (IoT 2017), 22.-25.

[8] Seeger, J., R.A. Deshmukh & A. Bröring (2018): Running Distributed and Dynamic IoT Choreographies. Global Internet of Things Summit (GIoTS 2018), 4.-7. June 2018, Bilbao, Spain. IEEE.

[9] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, et al. Web services composition: A decade's overview. Information Sciences 280 (2014), 218–238

[10] Christensen, E., Curbera F., Meredith G., Weerawarana S. *et al.* (2001) *Web Services Description Languagfe (WSDL) 1.1*, *W3C*.

[11] Martin, D.*, Burstein M., Mcdermott D., Mcilraith S., Paolucci M. et al.* (2007) 'Bringing semantics to web services with OWL-S', *World Wide Web*, 10(3), pp. 243–277. doi: 10.1007/s11280-007-0033-x.

[12] Allemang, D. and Hendler, J. (2008) *Semantic Web for the Working Ontologist: Modeling in RDF, RDFS and OWL*. Amsterdam.

[13] Ruben, L., Polleres, A. and Lausen, H. (2004) 'A conceptual comparison of WSMO and OWL-S', *WSMO Working Group working draft*, pp. 1–27. doi: 10.1007/b100919.

[14] Kopechy, J., Vitvar T., Bournez C., Farrell J. *et al.* (2007) 'Sawsdl: Semantic annotations for WSDL and XML schema'. Available at: https://www.w3.org/TR/sawsdl/.

[15] Kopecky, J., Gomadam, K. and Vitvar, T. (2008) 'hRESTS : An HTML microformat for describing REST-ful web services Conference Item', 619. doi: 10.1109/WIIAT.2008.379.

[16] Mayer, S., Verborgh, R. and Kovatsch, M. (2016) et al. 'Smart Configuration of Smart Environments', *IEEE Transactions on Automation Science and Engineering 13*, pp. 1–8.

[17] Gessler, D. D., Schiltz, G. S. and May, G. D. et al. (2009) 'SSWAP: A Simple Semantic Web Architecture and Protocol for semantic web services', *BMC Bioinformatics*, 10(1), pp. 1–21. doi: 10.1186/1471-2105-10-309.

[18] Lécué, F., Gorronogoitia, Y. and Gonzalez, R. (2010) 'SOA4ALL: An innovative integrated approach to services composition', in *ICWS 2010 - 2010 IEEE 8th International Conference on Web Services*, pp. 58–67. doi: 10.1109/ICWS.2010.68.

[19] Medjahed, B. and Bouguettaya, A. (2005) 'A multilevel composability model for semantic web services', *IEEE Transactions on Knowledge and Data Engineering*, 17(7), pp. 954–966. doi: 10.1109/TKDE.2005.101.

[20] Ovadia, S. (2014) 'Automate the Internet With "If This Then That" (IFTTT)', Behavioral and Social Sciences Librarian, 33(4), pp. 208–211. doi: 10.1080/01639269.2014.964593.

[21] Ur, B., Pak, M. and Ho, Y. (2016) 'Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes', *Conference on Human Factors in Computing Systems*, pp. 3227–3231. doi: 10.1145/2858036.2858556.

[22]Villarroel, G. F., Crosta, D. E. and Romero, C. (2017) 'Integration of Analytical Tools to Obtain Reliable Production Forecasts for Quick Decision-making', *79th EAGE Conference and Exhibition 2017 - SPE EUROPEC*. doi: 10.3997/2214-4609.201701624.

[23]*Drools Business Rules Management System (BRMS)* (2019). Available at: https://www.drools.org.

[24]Science, C. (1982) 'Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem *', *Artificial Intelligence*, 19(3597), pp. 17–37. doi: 10.1016/0004-3702(82)90020-0.

[25]'Biomoby' (2019). Available at: http://biomoby.org.

[26]'Serviceweb30' (2019). Available at: http://www.serviceweb30.eu.

[27]'NodeRed' (2019). Available at: https://flows.nodered.org.

[28]Weerawarana, S. *et al.* (no date) 'Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more'.

[29]Shin, S. and Gu, G. (2013) 'Attacking Software-Defined Networks : A First Feasibility Study', *InProceedings of ACM SIGCOMMWorkshop on Hot Topics in Software Defined Networking (HotSDN'13)*, pp. 165–166.

[30]Shin, S., Vinod, Y. and Porras, P. (2013) 'AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks'.

[31]Kotani, D. and Okabe, Y. (2016) 'A Packet-In message filtering mechanism for protection of control plane in OpenFlow switches', *IEICE Transactions on Information and Systems*, E99D(3), pp. 695–707. doi: 10.1587/transinf.2015EDP7256.

[32]Yoon, C., Lee, S. and Kang, H., et al. (2017) 'Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks', *IEEE/ACM Transactions on Networking*, (25). doi: 10.1109/TNET.2017.2748159.

[33]Petroulakis, N. E., Spanoudakis, G. and Askoxylakis, I. (2017) 'Fault Tolerance Using an SDN Pattern Framework', *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*.

[34]H. van der Veer, A. Wiles, Achieving Technical Interoperability – the ETSI Approach, April 2008 ETSI White Paper No. 3, [Online]. Available: http://www.etsi.org/images/files/ETSIWhitePapers/IOP%20whitepaper%20Edition%203%20final.pdf

[35]CETIC 6LBR, 6LoWPAN/RPL Border Router solution. [Online]. Available: https://github.com/cetic/6lbr/wiki

[36]SeMIBIoT: Secure Multi-protocol Integration Bridge for the IoT, E. Palavras, K. Fysarakis, I. Papaefstathiou, and I. Askoxylakis, IEEE International Conference on Communications (IEEE ICC 2018), Communications QoS, Reliability, and Modeling Symposium (CQRM), Kansas City, MO, USA, May 20-24, 2018.

[37]G. Hatzivasilis, O. Soultatos, E. Lakka, S. Ioannidis, D. Anicic, A. Bröring, K. Fysarakis, G. Spanoudakis, M. Falchettom, and L. Ciechomski, "Secure Semantic Interoperability for IoT Applications with Linked Data", 2019 IEEE Global Communications Conference (GLOBECOM 2019), Waikoloa, HI, USA, Dec. 9-13, 2019.

[38]Kiljander J. e. a., "Semantic interoperability architecture for pervasive computing and Internet of Things," IEEE Access, vol. 2, pp. 856-873, 2014.

[39]Schmid, S., Bröring, A., Kramer, D., Käbisch, S., Zappa, A., Lorenz, M., ... & Gioppo, L. (2016, November). An architecture for interoperable IoT ecosystems. In International Workshop on Interoperability and Open-Source Solutions (pp. 39-55). Springer, Cham.

[40]FIWARE Open Specification IoT Broker. [Online]. Available: https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.IoT.Backend.IoTBroker

[41]Bröring, A., Schmid, S., Schindhelm, C. K., Khelil, A., Käbisch, S., Kramer, D., ... & Teniente, E. (2017). Enabling IoT ecosystems through platform interoperability. IEEE Software, 34(1), 54-61.

[42]Seeger, J., Bröring A., Pahl M.-O., Sakic. E. 2019. Rule-Based Translation of Application-Level QoS Constraints into SDN Configurations for the IoT. EuCNC 2019, 18.-21. June, Valencia, Spain. IEEE.

[43]E. Sakic, Kulkarni V., Theodorou V., Matsiuk A., Kuenzer S., Petroulakis N. E., Fysarakis K. 2018. Virtuwind–an SDN-and NFV-based architecture for softwarized industrial networks, in International Conference on Measurement, Modelling and Evaluation of Computing Systems. Springer, 2018, pp. 251–261.