Ref. Ares(2020)1280324 - 29/02/2020

SEMIoTICS

Deliverable D3.9
Bootstrapping and Interfacing SEMIoTICS Field Level Devices (final)

Deliverable release date	29.02.2020
Authors	1. Darko Anicic, (SAG) 2. Mirko Falchetto (ST), 3. Konstantinos Fysarakis (STS), 4. Korbinian Spielvogel, Felix Klement, Henrich C. Pöhls (UP) 5. Philip Wright, Domenico Presenza (ENG)
Responsible person	Darko Anicic (Siemens AG)
Reviewed by	Kostas Ramantas (IQU), Othonas Soultatos (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	Meaning
IIoT	Industrial Internet of Things
CoAP	Constrained Application Protocol
DTM	Device Type Manager
EDD	Electronic Device Description
EDDL	EDD Language
FDT	Field Device Tool
GSD	General Station Description (ger. Gerätestammdaten)
HART	Highway Addressable Remote Transducer
HTTP	HyperText Transfer Protocol
ICT	Information and Communication Technology
IHES	Intelligent Heterogeneous Embedded System
IODD	Input Output Device Description
IoT	Internet of Things
JSON	JavaScript Object Notation
MCU	Micro Controller Unit
MQTT	Message Queuing Telemetry Transport
OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
PROFIBUS	Process Field Bus
PROFINET	Portmanteau for Process Field Net
QoS	Quality of Service
RPi	Raspberry PI
SDN	Software-Defined Networking
SEMIoTICS	Smart End-to-end Massive IoT Interoperability, Connectivity and Security
SPDI	Security, Privacy, Dependability, and Interoperability
TD	W3C Web of Things Thing Description
UC	Use Case
W3C	World Wide Web Consortium
WoT	Web of Things

EXECUTIVE SUMMARY

SEMIoTICS architecture consists of three levels: Field level, Network level, and Backend/Cloud level. This deliverable provides design and the first draft of the semantic integration of the Field level into the SEMIoTICS architecture. In particular, this includes semantics that aims to make brownfield devices from existing automation systems interoperable with newly bootstrapped devices. The new devices are Internet of Things (IoT) devices. They interact in a way, which is different to standards-conform brownfield devices. IoT devices can also not be engineered with existing industrial tools. On the other hand, they are cheaper and can make automation systems more flexible. For example, it should be possible to easily plug a new IoT device and create a new application that process data from existing brownfield devices, as well as data from the new device. This possibility would significantly decrease costs of upgrading automation systems and development of new applications. Moreover, it would be possible to create a new class of IoT applications, which have not been envisioned at the time of creation of an existing automation system. In order to enable this, we have to integrate the brownfield system with the IoT Field level. This assumes realization of a common communication access, as well as an integrated data access. As a prerequisite for this, we have to describe capabilities of both brownfield and IoT devices with a harmonized semantic model. This will enable application developers to easily understand underlying infrastructure when developing new applications and will enable tool support when discovering and engineering devices. Therefore, in this deliverable we are developing an IoT gateway that fulfils these requirements at the Field level. The integrated device semantics at this level is a key enabler for bootstrapping and easier integration of devices in an IoT system, as well as a facilitator for creation of new applications. Apart from this, this work will be the central basis for the connectivity network and the "glue" between the Backend/Cloud on one hand side, and Field level devices on the other side.

The first version of this work contains a concept of an IoT gateway and technology building blocks for its realization. This work extends the first version by providing the implementation of the concept and reviewing gateway's requirements at the Field level. In particular, we have provided the gateway implementation in Section 4 (see Section 4.1 Section 4.2 Section 4.3 Section 4.4 Section 4.5, and Appendix 8.1 Appendix 8.2, Appendix 8.3). The entire document has been reviewed and updated.

1 INTRODUCTION

1.1 Semantic Interoperability in SEMIoTICS

The Internet of Things (IoT) is the network of things that are connected together. They interact and exchange data over Internet. Things can represent physical or virtual objects. Thus, manipulating IoT things it is possible to impact the physical world represented by those things. Before IoT emerged as a paradigm shift, the networks of connected devices had already existed. Hence the question is what is new by IoT.

IoT promises a new class of applications based on things that interact over Internet. The game changer is not the fact that we connect many things to Internet. Instead, the difference that IoT aims to make is the *interoperability of things*, that is, the ability of things to interact in a meaningful way. How can we enable things to interact, knowing that there are so many diverse things and even more possible ways of their interaction, different communication protocols, different serialization- or data- formats they exchange, and different purposes of Things? A new class of IoT applications assumes even interactions of things that never before have been envisioned to interact together. Thus, the greatest challenge in IoT is to make things interoperable. One way to achieve this is to describe things, their capabilities, and data they produce or consume in a machine understandable form. Such a description could be then used to discover things relevant for an application. It can also serve to figure out how these things could interact. The description should be formalized, with a clear semantic meaning, so that both humans and machines can interpret it. In this way we would not have just Internet of mere things. Instead, IoT would be the Internet of *semantically-described things*. Semantics for IoT is the key enabler of applications that operate on physical world objects. It is a prerequisite for achieving the interoperability of things, and thus for realization of a new class of IoT applications. Figure 1: Enabling real-world Applications with IoT semantics depicts this vision.

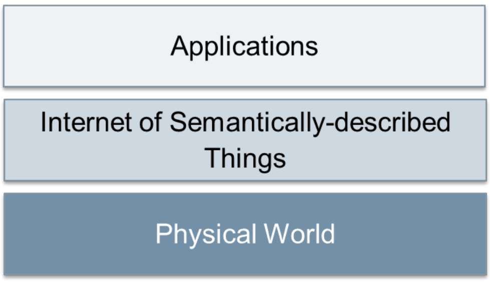


Figure 1: Enabling real-world Applications with IoT semantics

The Industrial Internet of Things (IIoT) refers to the IoT, where things are industrial devices and applications are bound to various industrial operations. The interoperability in IIoT plays as important role as in IoT. Field devices in automation systems originate from different manufacturers and have to be integrated in such a way that a standard access to their data is possible. This aims to reduce the effort for device- engineering, configuration, management, operation, and versioning, as well as to enable industrial applications to operate on integrated data. A distinguished difference in IIoT is that semantics for describing things must be standardized, and a good portion of it already exists. Thus, in the context of IIoT application, before enabling the things interoperability we have to tackle a challenge of *integrating semantics of brownfield industrial devices* with new IoT things.

In this work we use semantic models to provide the meaning to data that is exchanged between things, and further to describe capabilities of things in a machine interpretable format. Our gateway will serve as a *semantic mediator* in the task of integrating semantics of brownfield industrial devices and new IoT things. As the input, the gateway accepts data from diverse field devices. As the output, it provides an API to access semantically-described data along with descriptions of capabilities of connected devices. The API is based on W3C Web of Things (WoT) standard, and things are specified in the WoT Thing Description (TD) format. TD is semantically annotated with iotschema.org. In our approach we will strive to use existing standards to describe things, as only the standard semantics provides the necessary base for the interoperability. Thus, we will extend iotschema.org with standard semantics that is required for SEMIoTICS use cases. These semantic models will be a cornerstone for different tasks such as discovery of services and devices; bootstrapping and interfacing of IIoT field devices to enable plug-and-play functionality; creation of IIoT applications with low effort, and others.

1.2 Focus and Relations to Other Project Tasks

Figure 2 shows three main levels of concern in the SEMIoTICS project, i.e., Field level, Network level, and Backend/Cloud level.

In the first level, field devices become things in the context of IoT or IIoT. Both brownfield devices and new devices are brought to a common accessibility layer by Gateway. The accessibility layer assumes a common communication interface and semantic description of capabilities of field devices, provided in a standard and harmonized way, see Figure 2. Field-level data and devices, described and enriched with IoT semantic models, are ready to be used for Edge analytics. Such localized analytics (Embedded Intelligence) deploys machine learning algorithms to extract the most important features of the data locally. It also transfers valuable results to the cloud for further, global, processing and updates of the learning model.

The second layer is the networking layer. IIoT applications typically need to satisfy a range of quality of service (QoS) parameters related to the networking. In order to accomplish this, IIoT applications will need to be resource and network aware. Only then they will take full advantage of agile networks and underlying network programmability as provided by Software Defined Networking (SDN). SDN allows network programmability, which can be used to decouple network control from the forwarding network (aka data) plane and to make the latter directly programmable by the former. Integrating IIoT and SDN will increase network efficiency, as it will make it possible for a network to respond to changes or events detected at the IIoT application layer through network reconfiguration.

The Backend/Cloud layer gathers data from different sources and provides higher-level services (apps). Semantic meta-data of the gathered data is passed from the Field level to this level, where it can be further enriched with Cloud-level semantic models (Knowledge Graphs). Semantically described assets in the Cloud can be used in further processing by apps. In SEMIoTICS apps will be realized as application templates (Recipes) that are instantiated with particular assets. Recipes are semantically annotated with the same semantic models as assets. Hence the discovery of relevant Recipes, as well as the matching between Recipes and applicable assets, will be based on semantic processing.

The focus area and tasks related to the role of semantics in the SEMIoTICS project are positioned by the oval in Figure 2. In particular, the task T3.3 has the goal to use the standardized semantic models in order to enable the interaction between IIoT field level on one hand side and the SDN Controller Southbound Interface on the other one. The task provides the accessibility layer as a common communication interface and semantic description of capabilities of field devices. Further on, this work will make all field-level resources discoverable on a Gateway-level thing directory. The same mechanism will be applied for Cloud-level discoverability. Finally, field-level semantics will enable bootstrapping of filed devices in a SEMIoTICS system, thereby allowing easier integration of brownfield and new IIoT devices in an integrated ecosystem where rapid development of IIoT application is possible.

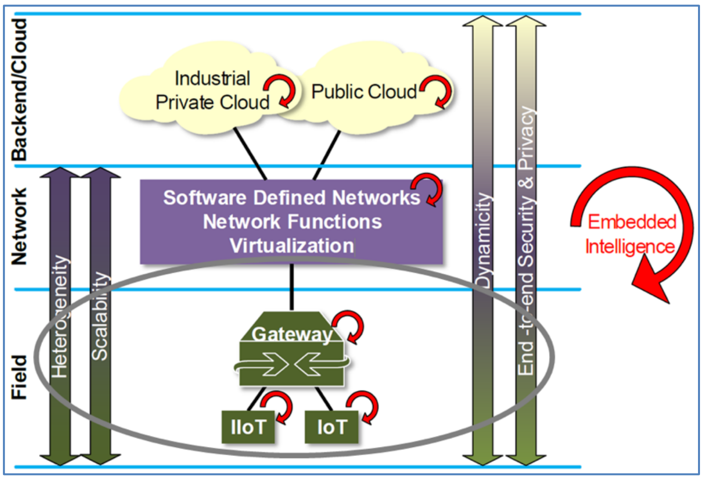
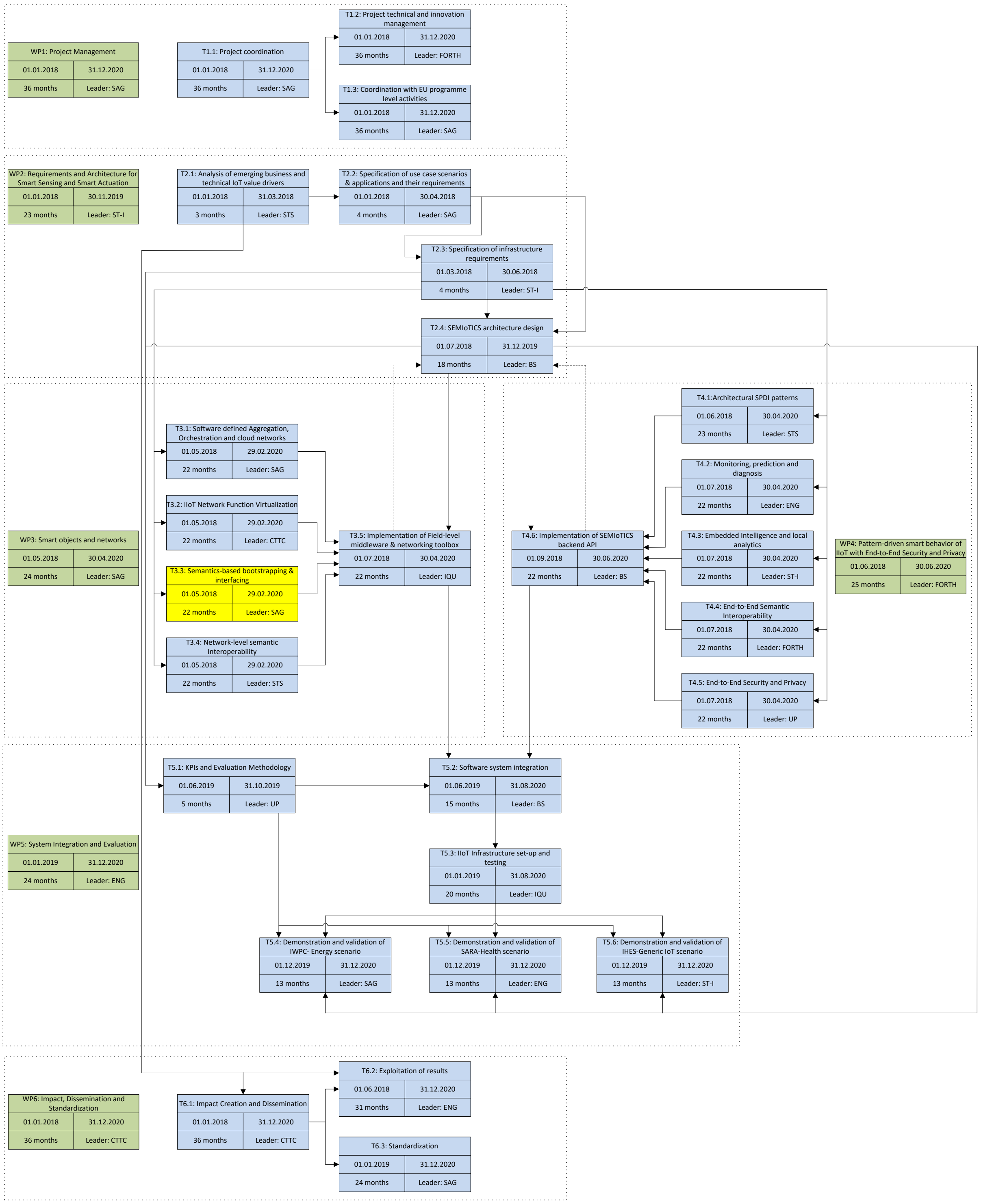


Figure 2: KEY IOT CHALLENGES DRIVING SEMIoTICS

1.3 PERT chart of SEMIoTICS


WP1: Project Management
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.1: Project coordination
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.2: Project technical and innovation management
01.01.2018
31.12.2020
36 months
Leader: FORTH
T1.3: Coordination with EU programme level activities
01.01.2018
31.12.2020
36 months
Leader: SAG
WP2: Requirements and Architecture for Smart Sensing and Smart Actuation
01.01.2018
30.11.2019
23 months
Leader: ST-I
T2.1: Analysis of emerging business and technical IoT value drivers
01.01.2018
31.03.2018
3 months
Leader: STS
T2.2: Specification of use case scenarios & applications and their requirements
01.01.2018
30.04.2018
4 months
Leader: SAG
T2.3: Specification of infrastructure requirements
01.03.2018
30.06.2018
4 months
Leader: ST-I
T2.4: SEMIoTICS architecture design
01.07.2018
31.12.2019
18 months
Leader: BS
WP3: Smart objects and networks
01.05.2018
30.04.2020
24 months
Leader: SAG
T3.1: Software defined Aggregation, Orchestration and cloud networks
01.05.2018
29.02.2020
22 months
Leader: SAG
T3.2: IIoT Network Function Virtualization
01.05.2018
29.02.2020
22 months
Leader: CTTC
T3.3: Semantics-based bootstrapping & interfacing
01.05.2018
29.02.2020
22 months
Leader: SAG
T3.4: Network-level semantic Interoperability
01.05.2018
29.02.2020
22 months
Leader: STS
T3.5: Implementation of Field-level middleware & networking toolbox
01.07.2018
30.04.2020
22 months
Leader: IQU
T4.6: Implementation of SEMIoTICS backend API
01.09.2018
30.06.2020
22 months
Leader: BS
T4.1:Architectural SPDI patterns
01.06.2018
30.04.2020
23 months
Leader: STS
T4.2: Monitoring, prediction and diagnosis
01.07.2018
30.04.2020
22 months
Leader: ENG
T4.3: Embedded Intelligence and local analytics
01.07.2018
30.04.2020
22 months
Leader: ST-I
T4.4: End-to-End Semantic Interoperability
01.07.2018
30.04.2020
22 months
Leader: FORTH
T4.5: End-to-End Security and Privacy
01.07.2018
30.04.2020
22 months
Leader: UP
WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy
01.06.2018
30.06.2020
25 months
Leader: FORTH
WP5: System Integration and Evaluation
01.01.2019
31.12.2020
24 months
Leader: ENG
T5.1: KPIs and Evaluation Methodology
01.06.2019
31.10.2019
5 months
Leader: UP
T5.2: Software system integration
01.06.2019
31.08.2020
15 months
Leader: BS
T5.3: IIoT Infrastructure set-up and testing
01.01.2019
31.08.2020
20 months
Leader: IQU
T5.4: Demonstration and validation of IWPC- Energy scenario
01.12.2019
31.12.2020
13 months
Leader: SAG
T5.5: Demonstration and validation of SARA-Health scenario
01.12.2019
31.12.2020
13 months
Leader: ENG
T5.6: Demonstration and validation of IHES-Generic IoT scenario
01.12.2019
31.12.2020
13 months
Leader: ST-I
WP6: Impact, Dissemination and Standardization
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.1: Impact Creation and Dissemination
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.2: Exploitation of results
01.06.2018
31.12.2020
31 months
Leader: ENG
T6.3: Standardization
01.01.2019
31.12.2020
24 months
Leader: SAG

Please note that the PERT chart is kept on task level for better readability.

1.4 Specific Project Requirements Related to This Project Task

This section contains project requirements that are derived from deliverable D2.3 and are specific to this task. The requirements are identified by request identifiers (Req-IDs), see tables below. These identifiers will be used throughout the deliverable to denote parts thereof, which address specific requirements.

Table 1: Specific requirements for this task from the General Platform Requirements

Req-ID	Functional	Description	Req. level	Status Referenced In
R.GP.1	Yes	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	MUST	Section 3.1

Table 2: Specific Field Layer requirements

Req-ID	Functional	Description	Req. level	Status Referenced In
R.FD.5	Yes	Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components	SHOULD	Section 3.1
R.FD.6	Yes	Field devices MUST interoperate using a standard communication protocol like REST APIs, COAP, MQTT.	MUST	Section 3.1
R.FD.7	Yes	Field devices MUST use standardize interoperable message format (e.g. JSON, etc.).	MUST	Section 3.1
R.FD.8	Yes	Field devices MUST support secure bootstrapping / registration protocol.	MUST	Section 3.3
R .FD.12	Yes	Greenfield device is expected to expose its capability over a W3C Thing Description, which semantically describes field resources, and to be computationally powerful enough to run a node-wot servient (that exposes the TD).	MUST	Section 3.1
R.FD.13	Yes	Brownfield device is assumed to consist of a sensor/actuator and a controller (PLC). The controller is expected to expose capability of its sensor/actuator over a native brownfield protocol (without the need for IIoT Gateway to interact directly with them).	MUST	Section 4

Table 3: Specific requirements for use case 1

Req-ID	Functional	Description	Req. level	Status Referenced In
R.UC1.1	Yes	Automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders	MUST	Section 4
R.UC1.8	Yes	Semantic and robust bootstrapping/registration of IIoT sensors and actuators with IIoT gateway MUST be supported.	MUST	Section 4
R.UC1.9	Yes	Semantic interaction between use-case specific application on IIoT Gateway and	MUST	Section 4

		legacy turbine control system MUST be supported.		
R.UC1.10	Yes	Sufficient compute environment MUST be supported on the IIoT Gateway to run use-case specific applications.	MUST	Section 4
R.UC1.11	No	Device composition and application creation SHALL be supported through template approach.	SHALL	
R.UC1.12	No	Standardized semantic models for semantic-based engineering and IIoT applications MUST be utilized.	MUST	Section 3, Section 4
R.UC1.13	Yes	Middleware functionality MUST be supported on IIoT gateway, to deal with termination of IIoT sensors, signal processing and termination of interfaces to legacy systems to provide prioritization and QoS for IIoT applications.	MUST	Section 4

Table 4: Specific requirements for use case 2

Req-ID	Functional	Description	Req. level	Status Referenced In
R.UC2.5	Yes	The SEMIoTICS platform should allow the SARA solution to discover the IoT devices that are registered in the system. IoT devices deployed by the SARA solution are expected to register themselves into the system using various standard protocols (e.g. LwM2M, MQTT, Bluetooth LE, ZigBee, etc.).	SHOULD	Section 4
R.UC2.6	Yes	The SEMIoTICS platform SHOULD allow the SARA solution to retrieve the resources exposed by registered devices via their object model (i.e. a data structure wherein each element represents a resource, or a group of resources, belonging to a device). The SEMIoTICS platform SHOULD support at least the OMA LWM2M object model.	SHOULD	Section 3, Section 4

Table 5: Specific requirements for use case 3

Req-ID	Functional	Description	Req. level	Status Referenced In
R.UC3.2	Yes	IHES Sensing unit shall be able to interface and register to the IHES Sensing gateway with a standard IP based (i.e. TCP transport) 1 to many M2M communication protocol to properly handle node registration and capabilities negotiation.	MUST	Section 4
R.UC3.9	Yes	IHES Sensing gateway shall support 1 to many standard IP based (i.e. TCP transport) M2M communication protocol	MUST	Section 4

		to interface a number N of connecting IHES Sensing units (e.g. broadcast type).		
R.UC3.12	Yes	IHES Sensing gateway shall be capable to run Linux (e.g. Ubuntu OS) and standard graphics and browser libraries.	MUST	Section 4
R.UC3.13	Yes	IHES Sensing gateway should be able to support http and standard protocols for cloud interfacing (e.g. to make IHES LocalDB data available).	SHOULD	Section 4
R.UC3.14	Yes	The specific M2M protocol adopted on UC3 is based on MQTT. A MQTT broker service will be available to dispatch messages between the IHES Sensing gateway and its associated Sensing units.	MUST	Section 4
R.UC3.15	Yes	A use case specific serialized message protocol is required to coordinate the gateway and its associated units and exchange data / events / anomalies between them. JSON will be the preferred serialization format adopted.	SHOULD	Section 4
R.UC3.16	Yes	Each registered IHES sensing unit should send to the sensing gateway a keep alive signal on a specified period (e.g. few seconds) to notify the gateway it is correctly working. The sensing gateway should detect by this mean any non-working sensing unit and reconfigure the system accordingly.	SHOULD	Section 4
R.UC3.17	Yes	IHES Sensing units and IHES sensing gateway should share a common clock (i.e. global reference time), precise up to milliseconds, to properly classify events and data acquired during the processing. This global reference time will be negotiated when a sensing unit node will join a given gateway. Internally the system will work scheduling activities according to this global reference time.	SHOULD	Section 4

2 SEMIOTICS USE CASES & REQUIREMENTS – SEMANTICS PERSPECTIVE

The Internet of Things, among other benefits, promises extensible, flexible, and dynamic applications. For example, an existing automation system could be equipped with additional sensors and actuators in order to provide a new feature. Data produced and consumed by new devices can be used for new applications. In certain cases, owners of automation systems are incentivized to share this data with application providers in order to offer new added-value services or to decrease costs of their systems. This motivates us to work on industrial automation systems that are easily extensible with new IoT devices. The IoT promises this feature in a form of *plug-and-play* functionality. In this section we review SEMIoTICS use cases from a semantic perspective, i.e., we provide requirements related to device bootstrapping, engineering and networking, where the use of semantics bring benefits.

2.1 Use Case 1: Wind Energy

Automation systems are fully integrated vertical systems. The full integration brings them efficiency. At the same time, it bears inflexibility too. Once these systems have been engineered and operational, they cannot be changed easily. For example, it is not straightforward to plug a new device into a running system and expect the device to be functional with respect to an already engineered system (as specified by requirement R.UC1.8 in deliverable D2.3). Or it is not effortless to develop an added-value service for an existing automation system (as specified by requirement R.UC1.11 in deliverable D2.3). Automation systems are complex, diverse, and engineered for a specific purpose. The change in a running system must not impact the system itself. Second, the change, if needed, needs to be integrated in the rest of the system so that the new system is operational for existing and new applications. Today these tasks are typically performed by engineers. They have required know-how. The main reason why an automated procedure for adding a new device, in sense of plug-and-play is not possible lies in the fact that the expert's know-how is not explicitly represented in a *machine-interpretable* form. In order to enable creation of new IIoT applications in a dynamic environment we need to *explicitly represent this knowledge*, thereby expressing capabilities of field devices in machine-interpretable form. Moreover, device capabilities must be realized with standardized semantics as required by R.UC1.12 in deliverable D2.3. Only then it will be possible to extensively use reasoning machines to certain automate engineering tasks.

The following use case describes problems found in the current vertically integrated automation systems and sketches the role of semantics in IIoT in order to amend these problems. Figure 3 depicts three parts: an existing control system in a wind turbine; a new IoT device; and an industrial network, which connects all components. The existing control system runs and is expected to continue its functionality also after adding a new IoT device. For example, a Siemens SIMATIC S7 controller controls sensors and actuators, which are needed for a normal operation of a wind turbine. Values from these sensors and actuators are exposed over a SIMATIC S7 controller or an OPC-UA server. Our goal is to realize a new application, which requires an additional temperature sensor. This is often a case, for example, when the position of an existing temperature sensor is not appropriate for measurements needed for the new application. Therefore, we need to add a new sensor. Suppose for our application, the new temperature sensor can be an inexpensive IoT device. The question which arises here is how to integrate an IoT device with an existing automation system (see also requirement R.UC1.9 from deliverable D2.3). First, the IoT device cannot communicate over standard industrial protocols. Second, the IoT device cannot be added to the system over existing engineering tools (e.g., Siemens TIA Portal). Third, it is not effortless to develop a new added value application since the know-how for an existing and new systems are contained by different experts. Fourth, the application may impose additional requirements, e.g., quality of service (QoS) constraints related to the underlying network. These requirements are expressed in Figure 3 as a network constraint rule (NCR) and assumed by requirement R.UC1.1 in deliverable D2.3.

Based on this example application we will explain the role of semantics for interfacing SEMIoTICS field level devices. In order to enable an application to process data from brown field automation system and new IoT devices, we first need to enable a common application protocol. Second, we need to provide a common data model. Third, we need to provide a common semantic model, which will describe interaction patterns and capabilities of devices. The semantic model will also involve contextual information and expert's know-how, explicitly represented in a machine-interpretable format. Forth, QoS network-related criteria can be semantically described and interpreted by Software Defined Network (SDN) controllers prior to the deployment of the application in order to check whether the communication infrastructure can meet the requirements of an application. Thus, it is also a goal of our work in SEMIoTICS to provide a semantic model for describing QoS network-related parameters and SDN/NFV infrastructure so that an automated evaluation of both is possible. Only then, it will be possible for an application developer to efficiently discover field devices (based on capabilities they provide), and to put them into semantically-correct interactions, also when they demand different functional and non-functional requirements to be fulfilled. Overall, we see that the plug-and-play functionality is not easily achievable. Nevertheless, our goal is to enable realization of new IoT applications that have not been envisioned at the time of engineering an existing automation system.

Further on, semantic models and tools that we will provide as a part of this task can support other use cases too. For example, *semantic validation* of produced and consumed device data can be enabled in an automated manner. The same will be possible for an *automatic matchmaking* of the device's capabilities with the requirements of an application, or *replacement* of a malfunctioning automation device with a new IoT device, and so forth.

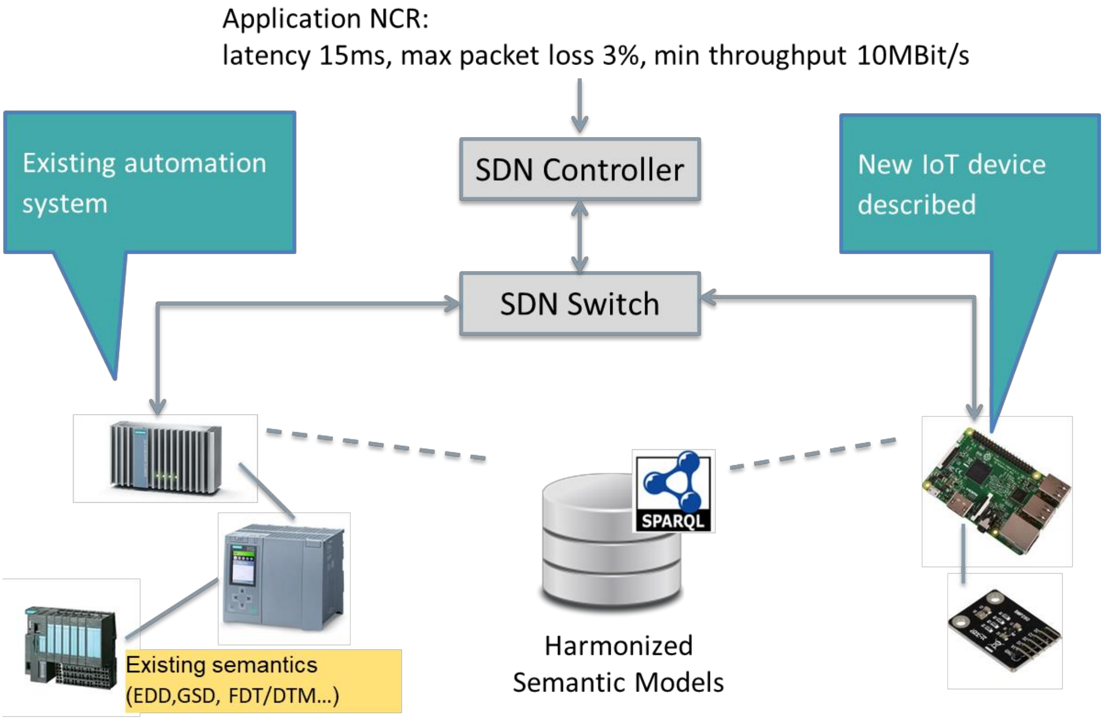

Application NCR:
latency 15ms, max packet loss 3%, min throughput 10MBit/s
Existing automation system
SDN Controller
New IoT device described
SDN Switch
SPARQL
Existing semantics (EDD,GSD, FDT/DTM...)
Harmonized Semantic Models

Figure 3: Semantic-based Engineering & Networking

In the following we summarize the current state of the practice in existing automation systems and specify goals to be achieved with our semantic-based approach.

2.2 Use Case 2: SARA-Health

The aim of the SARA case study (UC2) is to evaluate how the technologies and methodologies developed in the SEMIoTICS research project could improve the development of an Information and Communication Technology (ICT) solution aimed at sustained independence and preserved quality of life for elders with Mild Cognitive Impairment or mild Alzheimer's disease, with the overall goal of delaying institutionalization: supporting both 'aging in place' (individuals remain in the home of choice as long as possible) and 'community care' (long-term care for people who are mentally ill, elderly, or disabled provided within the community rather than in hospitals or institutions).

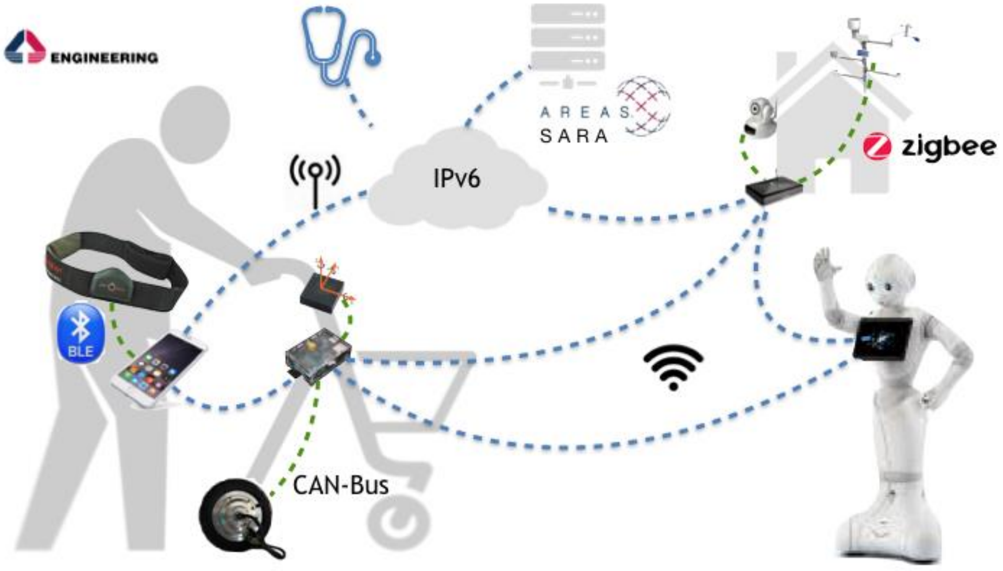


Figure 4: SARA Key components and protocols

This envisaged solution relies on a network of sensors and actuators. The communications within the network may be either wired or wireless. The nodes in the network are highly heterogeneous ranging from single accelerometers, presence sensors (e.g. camera) to sophisticated robotic components (e.g. a rollator frames improved with sensors and actuators, Pepper - a humanoid robot). The sensors and actuators network also need to communicate with backend cloud services not only to store data (e.g. measures that are used as monitoring data by doctors to improve treatment and provide assistance) and run computationally intensive tasks entailed by the assistive tasks provided by the solution to their users (e.g. patients, caregivers), see requirement R.GP.1 in deliverable D2.3. The field devices part of the solution communicates with cloud back-end services either via wired network or cellular connectivity.

The development of the software component of such a solution presents a number of challenges:

- the development of software aimed to control physical processes (i.e. falls).
- the integration of heterogeneous application protocols brought in the system by the use of off-the-shelf components (e.g. the humanoid robot).
- the customization of the software with respect to the introduction of new sensors/actuators needed to address patient-specific care requirements as formulated by doctors/carers.
- the provisioning of self-adaptation mechanisms to enable opportunistic networking with IoT elements deployed by third parties (e.g. sensors and actuators part of the smart environment).
- At the technological level the solution integrates a wide range of components, including commercial products, third-party components and research prototypes. Though most of the components of the solution are readily available, their integration results in a clear technological innovation.
- At the service level the solution leads to added-value personalized solutions and services. This will allow care services providers (e.g., health organizations, housing organizations, insurance companies) to include innovative added-value solutions to their services portfolio.

- The implementation of the SARA solution requires to address interoperability issues at various stages of the development:

- Design time: due to the heterogeneous application protocols (e.g. Bluetooth, ZigBee) brought in the system by the use of off-the-shelf devices. As an example, consider the situation faced by the developer of the Weight Balancer controller of the Robotic Rollator. The Weight Balancer is one of the software modules running on the single board computer on board of the Robotic Rollator. Its function is to control the motorized hub wheels trying to counteract the forces that might result in a fall of the user of the rollator. The Weight Balancer takes its decisions based on the measures taken by a series of sensors (e.g. IMUs, proximity sensors) residing either on the rollator or worn by the user (i.e. belonging to the BAN - Body Area Network - subsystem). The Weight Balancer, hence, during its initialization phase have to discover which are the sensors available and suitable for its purposes. However, as shown by Figure 4, the communication protocol of the sensors on board of the rollator and that of the sensors within the BAN varies (e.g. CAN Bus vs Bluetooth). As a consequence, the code initializing the Weight Balancer has to be conceived to deal with two different discovery protocols and object models. The situation is not satisfactory since, apart from the low manageability of the resulting code, the solution would be fragile with respect to possible changes in the protocols adopted by the rollator or the BAN. The use of a semantic layer has a positive impact on the development of a Weight Balancer relying on it since it would isolate the code of the Weight Balancer from the differences brought by the different protocols.

- Deploy time: although all the instances of the SARA solution will share a common set of functions, it is expected that additional features may have to be deployed in specific instances to address patient-specific care requirements as formulated by doctors/carers (e.g. the need of an oxygen concentrator for patients affected by respiratory diseases). The deployment of this feature occurs when technologies (e.g. robotic rollator, BAN) are configured for a specific patient. However, the set of the possible additional features has to be left open given the high variability of the conditions and needs we may encounter across different patients. Taking again the rollator as an example, we want to minimize the effort required to implement a new functionality very specific to a patient: the cost of developing a patient specific functionality should be a fraction of the cost of the design of the entire rollator. As an example, let us take the situation where on a specific rollator has to be extended with an additional functionality, say a Gait Analysis module. A Gait Analysis module is a software module that takes low level measures from the sensors on board of the rollator, aggregates them into higher level measures and forward them to the SARA backend cloud services (e.g. for storage or furthermore complex analysis). Part of the effort needed to develop such module for an already engineered rollator would be devoted to discover, which sensors are available for that purpose, and more importantly, the settings of their operational parameters (e.g. sampling rate). This part of the effort would be reduced by the availability of a semantic layer enabling the development of module with reflective capabilities and, hence, having the capacity to self-configure.

- Run time: the activation of some of the functionalities envisaged for the SARA solution requires the availability of services/devices external to the solution. As an example, the support to navigation offered by the Robotic Rollator relies on the availability of a service able to provide a map of the environment. This service could be offered, for example, by a Building Information Management (BIM) system external to SARA. The Guidance module is activated only if a map service is available from the surrounding environment. A desired behavior for the Robotic Rollator is to automatically activate the Guidance functionality whenever required (implicitly or explicitly) by the user and a map service is available. Also, in this case, the availability of a semantic layer eases the development of a service discovery function by hiding the differences between the protocols used by the different map services to advertise their capabilities.

- On course a development based on the availability of the semantic layer introduced to address the above-mentioned interoperability issues should not prevent the achievement of other key general requirements for the solutions:

- The SARA solution should process sensors data in a real-time fashion to be able to control physical processes (i.e. falls).

- The solution should support a highly heterogeneous range of sensors/actuators: from single accelerometers, presence sensors (e.g. camera) to sophisticated rollator frames improved with sensors and actuators.

2.3 Use Case 3: IHES Generic-IoT

In the following we specify requirements for the case study (UC3): The Intelligent Heterogeneous Embedded System.

- IoT (SEMIoTICS) system should be based on standardized IoT technologies, e.g., JSON, MQTT and W3C WoT (see requirement R.UC1.12 in deliverable 2.3).
- IHES generic IoT UC will implement / deploy within SEMIoTICS a distributed local intelligent unsupervised learning pattern: several intelligent sensing devices are coordinated by a local IHES Sensing Supervisor in order to enable all system functionalities. In this respect interoperability and clear semantic pattern messages are a key-focus (see requirement R1.10 in deliverable 2.3).
- IHES system should provide bootstrapping capabilities to IHES sensing nodes to join/detach from a local computing intelligent cluster managed by IHES supervisor. A semantic pattern to register a new IHES sensor node to the local IHES supervisor should be defined (see requirement R.UC1.8 in deliverable 2.3).
- Each IHES sensing node will analyze locally sensed data and will detect at single node level any anomaly (according to a self-learned model) and will be reported to joined IHES supervisor node. Raw sensing data are transmitted only during node bootstrap and will not be propagate outside local computing cluster managed by a single supervisor instance running on IoT gateway. IoT gateway deployed IHES supervisor will be responsible to propagate to upper level SEMIoTICS components all relevant events collected by underlying connected sensing nodes.
- All raw sensing data and the events triggered by the local analytics deployed in each IHES Sensing node, will be stored in the IHES Local DB component and should be made available under through REST API.
- IHES UC Local analytics will be implemented by means of unsupervised learning algorithms further detailed as part of D4.10.
- IHES supervisor is implemented as a service component into IIoT SEMIoTICS Gateway. End-to-end semantic interoperability will be ensured at this level of the architecture by the Gateway Semantic Mediator component (see D2.4 for further details)
- Goal 1: implement an unsupervised learning distributed system at edge level devices by implementing intelligent self-sensing/self-learning algorithms at STM32 Micro-controller units (MCUs).
- Goal2: allow massive system scalability by spreading system overall complexity at several levels of the architecture (MCUs, RPi like gateway devices, Backend/Cloud), allow for lower power consumption and network congestion by processing sensor raw data directly on the sensing unit and/or local supervisor cluster. These distributed systems are more resilient to the environment and devices disconnection compared to IoT centric ones in which any kind of analytics is deployed at cloud level.
- Goal 3: Plug & Play a new IHES sensing unit in the IoT IHES Supervisor and make its resources at disposal for other SEMIoTICS components through a subset of the IoT gateway components (e.g. the GW Semantic Mediator or the Semantic Edge Platform Components).

2.3.1 Semantic Mapping using JSON interchange format

The IHES Generic IoT UC main goals could be summarized in two main objectives:

- Provide a reference generic framework for local distributed intelligence and analytics as opposed to the cloud-centric traditional approach (this aspect will be analyzed and better discussed in T4.3). In short, these ones are the specific IHES UC (local analytics) components that will be presented in D2.5 referred in Figure 27 (i.e. "Model Validation", "AI Online learning" and "Event Detection"): they will be all implemented at MCU level as a binary firmware on STMicroelectronics STM32 prototype boards.

- Provide a new communication pattern where intelligent nodes cooperates together to form a local / lightweight computation cluster resilient on one side to the sensed environment and to poor connection conditions in the other side (i.e. vs the backend / cloud level). For this a new semantic will be needed and has been presented.

As part of WP2 activities IHES UC has been detailed and analyzed and specific requirements has been defined by considering mainstream enabling technologies and innovative approaches inspired by the emerging IoT edge computing paradigm. From the beginning it was evident that in such distributed systems communication plays relevant roles: in particular communication between "intelligent" devices. Thus, more complex and heterogeneous semantic patterns have to be defined. At the very beginning we considered the communication aspect and the associated semantic as different aspects of the whole problem, even if it is common sense to say that a communication pattern implies a specific semantic to be considered.
Hence, we focused first on defining the right communication pattern for the IHES use case. Three of them are available in literature: 1-to1 communication patterns (e.g. a request/response client/server communication is a typical example of this), 1-to-N communication patterns (i.e. message broadcasting like UDP datagrams), N to N asynchronous communication patterns (e.g. the publish/subscribe pattern implemented in MQTT). The first pattern is not suitable for the IHES demonstrator because it implies a 1-to-1 synchronous communication: not the ideal one when more connected devices are involved. Thus, we had the option to adopt a 1-to-N pattern or a more generic N-to-N one. Even if 1-to-N will in theory fits the requirements of the demo (in theory a supervisor is connected with N registered nodes), we opted instead for the N-to-N publish/subscribe pattern over TCP networks. This is the more general one and will allow us to have more flexibility in order to integrate the solution into SEMIoTICS or other vertical apps or services. In particular among all the possible technologies implementing a public/subscribe pattern we opted for the MQTT infrastructure, widely used in IoT connected systems: this infrastructure relies on asynchronous messages published to a centralized broker service, over a specific topic (i.e. dashboard), via optionally secured TCP transport stream). These messages could be received from a recipient that subscribe to the specific topic on a specific broker. Eventually as part of the QoS capabilities a broker could be asked to retain a message for a given timeframe. This way it is possible to realize an event-driven, asynchronous reliable transmission to N parties. In a subsequent step, once defines the transfer technology we focused on the specific semantic (i.e., data formats) conveyed by the message sent over MQTT transport stream, and the naming convention for the topics. We considered several options there as well: from more compact binary serialized data (usually non-human readable), such as the google protocol buffers or flat buffers, to more common textual based (human readable) such has XML or JSON formats. Both have pro and cons: the former usually have a more compact representation of a given message, but they are not very interoperable and requires an additional file named ".proto" shared between communicating partners in order to know how to parse the binary message. Latter ones are less compact (each message is a readable string composed by Ascii or UTF8 characters), but offers better interoperability, since the syntax is already embedded in the message and there is no need to share an additional file to parse them. So, for the IHES use case we opted for using JSON format. An example of some message / event shared between an IHES sensing node and its supervisor is shown in Figure 5, where a "Bootstrap" and "Change" messages are shown. All messages have a "type" field discriminating the type of message. They also have a "seqn" (i.e. growing sequence number) to uniquely identify the message sent by an IHES Sensing Unit in the local cluster and a timestamp "ts" field useful to coherently plot on a consistent shared timeline the messages received. This timestamp could be for example any kind of monotonic increasing clock. For the IHES demonstrator we opted to use as timestamp the number of milliseconds from each node boot time. According to the type of message other additional (extensible) attributes are possible in order to implement all the requirements for the communication defined in D2.3.

```
{
        "type": "BOOTSTRAP",
        "ts": 11650,
        "seqn": 0,
        "payload": {
                "id": "00-80-e1-00-00-99",
                "interval": 20000,
                "name": "Node-000099",
                "URI": "http://192.168.200.30:80",
                "s_caps": [
                        {
                                "name": “ACCELEROMETER”,
                                "id": 3,
                                 "unit": "mG",
                                 "sample_count": 150,
                                 "sample_period": [200.000000, 200.000000]
                                "ds_caps": [
                                        {
                                                "name": “ACC_X”,
                                                "id": 3,

                                        },
                                {
                                                "name": “ACC_Y”,
                                                "id": 4,

                                        },
                                {
                                                "name": “ACC_Z”,
                                                "id": 5,

                                        }

                                ]

                        },
                ]
        }
}
```

Figure 5: Example of JSON MQTT Protocol on IHES Generic IoT

3 DEVICE BOOTSTRAPPING AND SEMANTIC INTEGRATION IN SEMIOTICS

3.1 Building Blocks for Realization of Semantic Integration in SEMIoTICS

In the following we provide main building blocks, which will be used to address requirements imposed by previous section.

3.1.1 W3C Web of Things

The Web of Things (WoT) is a standardization activity by the World Wide Web Consortium[1] (W3C). WoT seeks to counter the fragmentation of the IoT through standard complementing building blocks (e.g., metadata and APIs) that enable easy integration across IoT platforms and application domains [1]. Figure 6 shows the three levels where the WoT building blocks can be applied: the device level, the gateway level (or Edge level), and the cloud level [2]. There exist interactions between different Things at each of these levels, including Web browser interactions too. The problem targeted by W3C WoT is a seamless integration of Things at different levels so that these interactions can be accomplished easier than today (without W3C WoT).

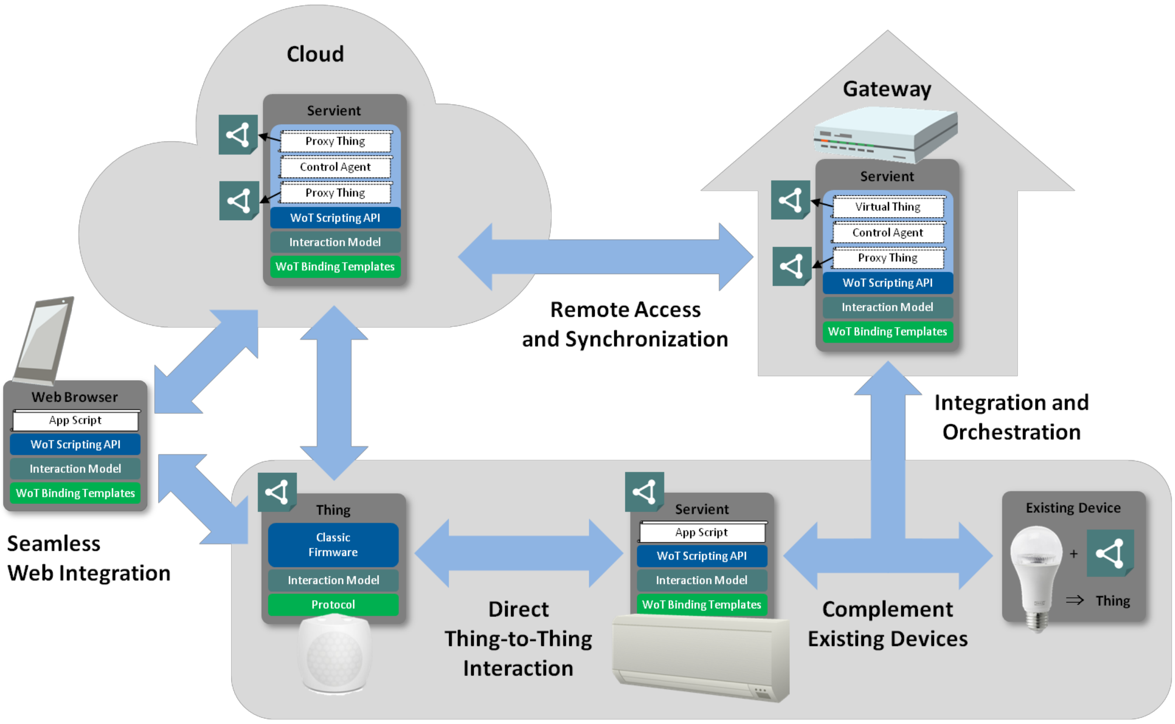


Figure 6: Abstract Architecture of W3C WoT

Figure 7: Conceptional Architecture of the WoT Building Blocks shows WoT building blocks [2]. A Thing is the abstraction of a physical or virtual entity that needs to be represented in IoT applications. This entity can be a device, a logical component of a device, a local hardware component, or even a logical entity such as a location (e.g., room or building) [2]. Thing is represented by a Thing Description.

[1] https://www.w3.org/

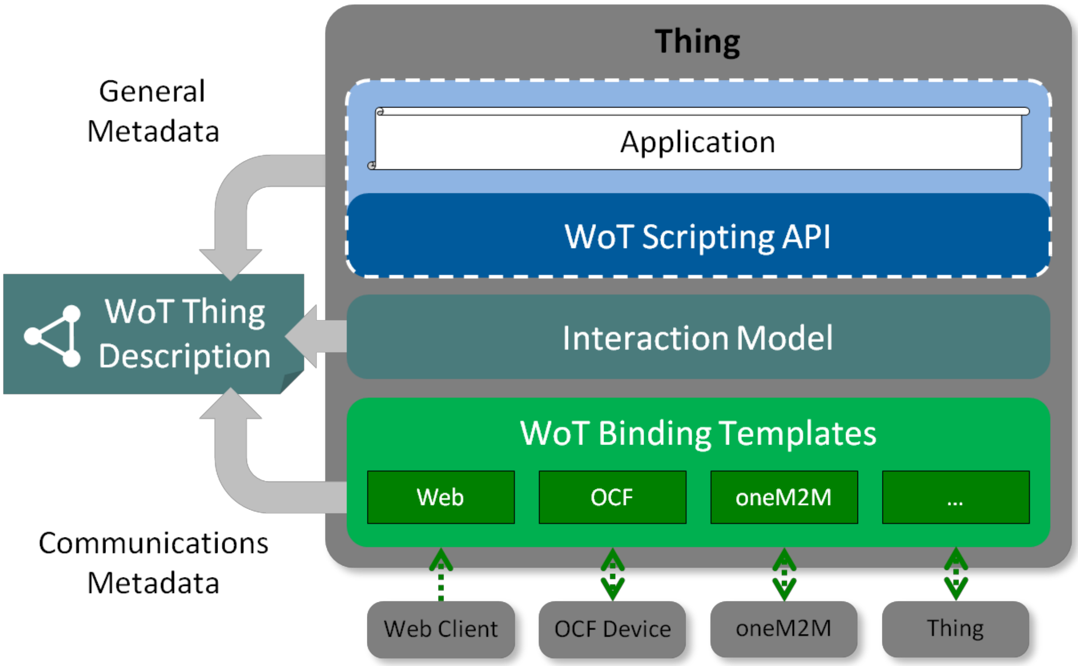


Figure 7: Conceptional Architecture of the WoT Building Blocks

3.1.1.1 Thing Description

W3C Thing Description (TD) is a building block in the WoT architecture, see Figure 7: Conceptional Architecture of the WoT Building Blocks. It is a machine-readable description of a Thing. A TD provides general metadata of a Thing as well as metadata about the Interactions, data model, communication, and security mechanisms of a Thing [2]. Thing's Interactions are specified in a so-called Interaction Model. The model defines three types of Interactions: Property, Action, and Event.

Properties expose the internal state of a Thing (its data points) that can be directly retrieved via GET method of the HTTP protocol or optionally modified via HTTP's POST method. For example, a GET method at the URI "https://mysensor.example.com/status" will return a string status value for that sensor. Properties can be observable which means pushing the new state after a change occurs (not an event).

Actions are functions that may manipulate the internal state of the thing in a way that is not possible through setting Properties. For example, change states that are not exposed as a property, modifying multiple properties, changing properties over time or with a process that should not be disclosed. Moreover, actions can be just functions, which do not use the internal state at all, and may simply process input data and return an output. HTTP's POST is the default method for invoking actions on a URI resource.

Events provide a mechanism that enables the Thing to asynchronously push messages. These messages are not stating but rather state transitions (events). Events could be triggered by internal state changes that are not exposed as Properties. Events must follow a consistent delivery approach to ensure that all occurred events are delivered. To that end subscriptions are utilized with HTTP's long polling sub-protocol at, for example, subscribing https://mysensor.example.com/oh will enable the sensor to provide a steady feed of data.

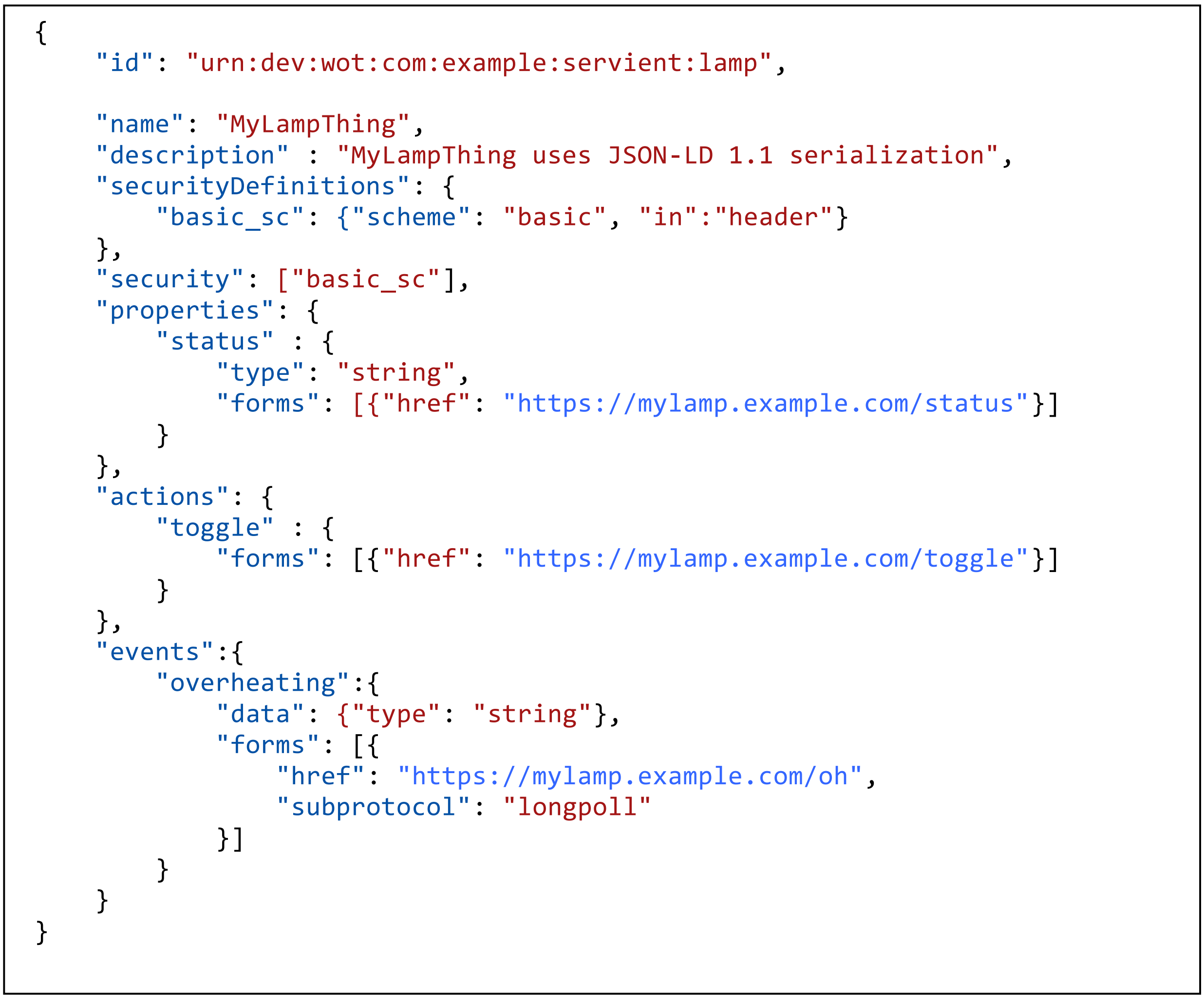
Form is a type of communication metadata that indicates one or more endpoints at which operation(s) on this resource are accessible. Using this metadata, various methods (e.g. GET, POST etc.) can be explicitly specified for properties and/or actions.

Links expose an operation like a regular web link works, as specified by IETF RFC 8288[2].

Versioning is a metadata that provides information about the current version of the Thing Description instance. This can be extended to include firmware and hardware versions.

ExpectedResponse is a communication metadata used for response messages (e.g. contentType of the response).

Figure 8: Thing Description Sample [3] shows an example of Thing Description describing a lamp. The lamp is accessible over HTTP protocol and is secured over a basic authentication security configuration (using an unencrypted username and password). This TD is serialized in JSON[3] format. The lamp has: Property “status”, which can be used to check whether it is on or off; Action “toggle” to turn it on or off; and Event “overheating” to indicate the lamp is overheated.



```
{
    "id": "urn:dev:wot:com:example:servient:lamp",

    "name": "MyLampThing",
    "description" : "MyLampThing uses JSON-LD 1.1 serialization",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        }
    },
    "actions": {
        "toggle" : {
            "forms": [{"href": "https://mylamp.example.com/toggle"}]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Figure 8: Thing Description Sample [3]

[2] https://tools.ietf.org/html/rfc8288

[3] https://www.json.org/

3.1.1.2 WoT Binding Templates

In order to provide support for multiple protocols, the current version of WoT, specifies protocol binding templates. These templates enable Things, which communicate over different protocols, still to interact together. The WoT Binding Templates are an informal collection of communication metadata blueprints that explain how to interact with different IoT Platforms [2]. For example, if an HTTP-enabled Web Thing, providing data in plain JSON, needs to interact with an CoAP-enabled OCF[4] Thing, which serializes data in CBOR[5], then it is needed only to provide corresponding Binding Templates for these two Things in their TDs. Of course, the prerequisite is that there exist implementations of protocol bindings for the used binding templates, see Figure 9. For example, W3C WoT already provides few Binding Implementations for protocols such as HTTP, CoAP, MQTT etc. But implementations for specific protocols, such as for example Siemens S7comm[6] (as needed in Use Case 1), do not exist. Thus, we need to implement them.

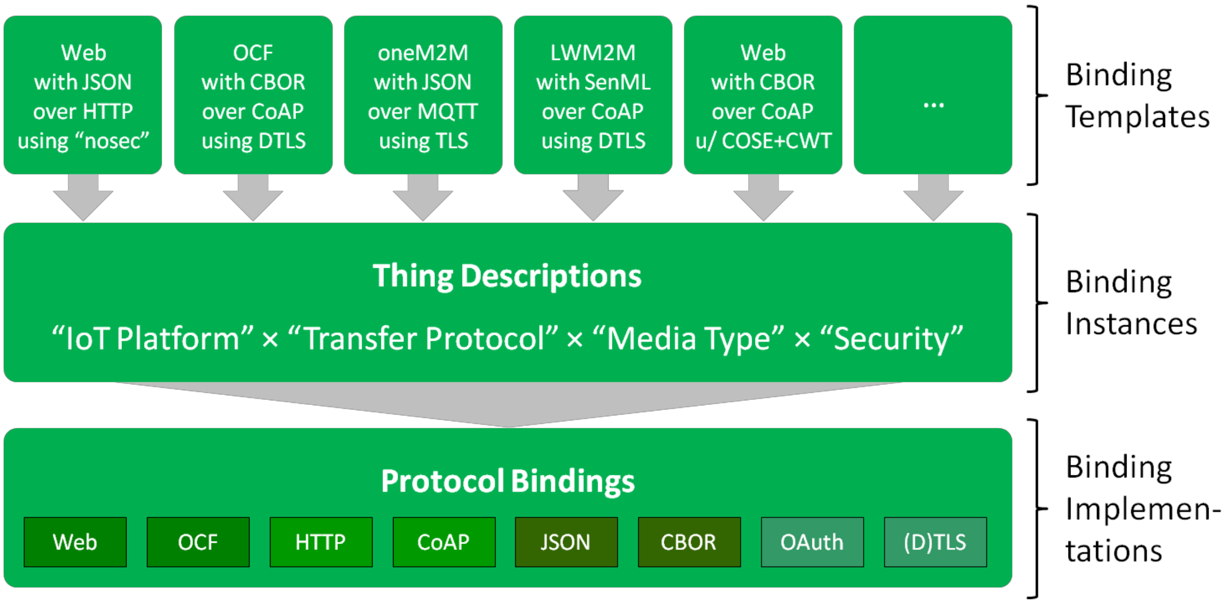

Web with JSON over HTTP using "nosec"
OCF with CBOR over CoAP using DTLS
oneM2M with JSON over MQTT using TLS
LWM2M with SenML over CoAP using DTLS
Web with CBOR over CoAP u/ COSE+CWT
…
Binding Templates
Thing Descriptions
"IoT Platform" × "Transfer Protocol" × "Media Type" × "Security"
Binding Instances
Protocol Bindings
Web
OCF
HTTP
CoAP
JSON
CBOR
OAuth
(D)TLS
Binding Implemen-tations

Figure 9: From Binding Templates to Protocol Bindings [2]

3.1.1.3 WoT Scripting API

The Wot Scripting API provides a programing interface for a Thing as described by its Thing Description. It provides a convenient and standardized way of accessing Thing's metadata, Properties, Actions, Events and so forth. In this way IoT applications can be developed easier. Moreover, these applications are easier to maintain as, for example, they don't need to be changed when Things are changed, as far as new Things provide equivalent data used by the application logic. Furthermore, standardized APIs enable portability for application modules, for instance, to move compute-intense logic from a device up to a local gateway, or to move time-critical logic from the cloud down to a gateway or edge node [2]. The Wot Scripting API is an optional building block.

3.1.2 iotschema.org

iotschema.org is a community organization for extending schema.org to connected Things. The organization provides an open, publicly available, repository of semantic definitions for connected Things [4]. It is an

[4] Open Connectivity Foundation (OCF): https://openconnectivity.org/
[5] http://cbor.io/
[6] S7comm: https://wiki.wireshark.org/S7comm

extension of well-known schema.org to enable descriptions of Things in the physical world and their data. iotschema.org provides a way for domain experts to easily create semantic definitions that are relevant to their application domain. iotschema.org reuses existing standardized semantic definitions whenever possible.

W3C Thing Description (TD) abstracts a Thing in terms of Properties, Actions, and Events. When creating a TD, one needs to specify semantic types of Things Properties, Actions, and Events. For example, it is not enough to know that “status” is a Property. For an application developer it is valuable to know also that the Thing has the light capability, a binary switch control etc. Further on, it is required to know that the “status” is a Property of type SwitchStatus, as defined by iotschema.org. There it is specified that this Property has data type Boolean. Thus, the application client “knows” not only the URL of the Property (from its TD), but it also “knows” what the Property is about and what data to expect when invoking the URI. Moreover, the type SwitchStatus has a unique Web identifier. Hence it is possible to discover all Things from a Thing registry that have the light capability or the SwitchStatus Property. Semantics, provided by iotschema.org, greatly help when developing IoT applications. In particular, it enables an application development based on so called Recipes. Recipes are application templates, created based on semantic descriptions from iotschema.org. They can be used to automate application development by matching real Thing’s Interaction Patterns, which are annotated with iotschema.org semantics, and Recipe requirements.

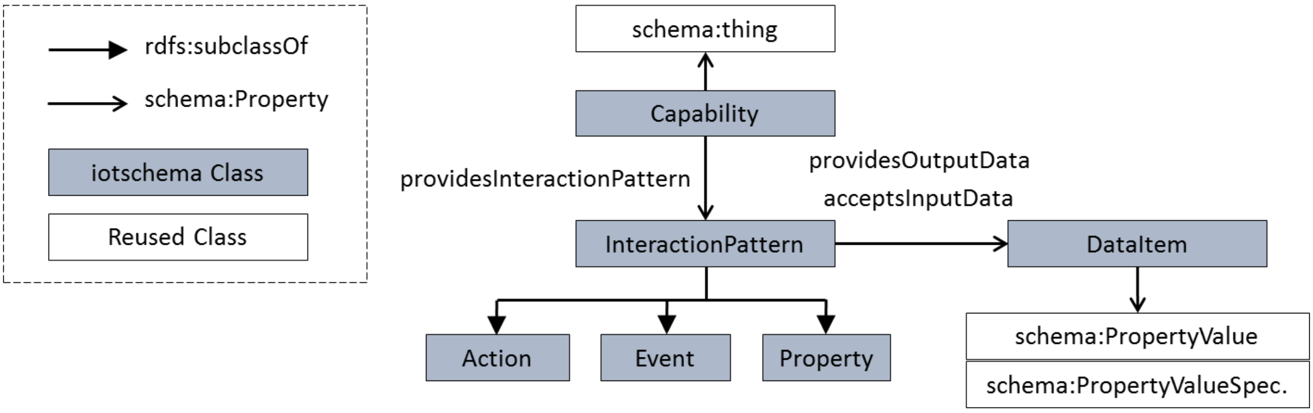
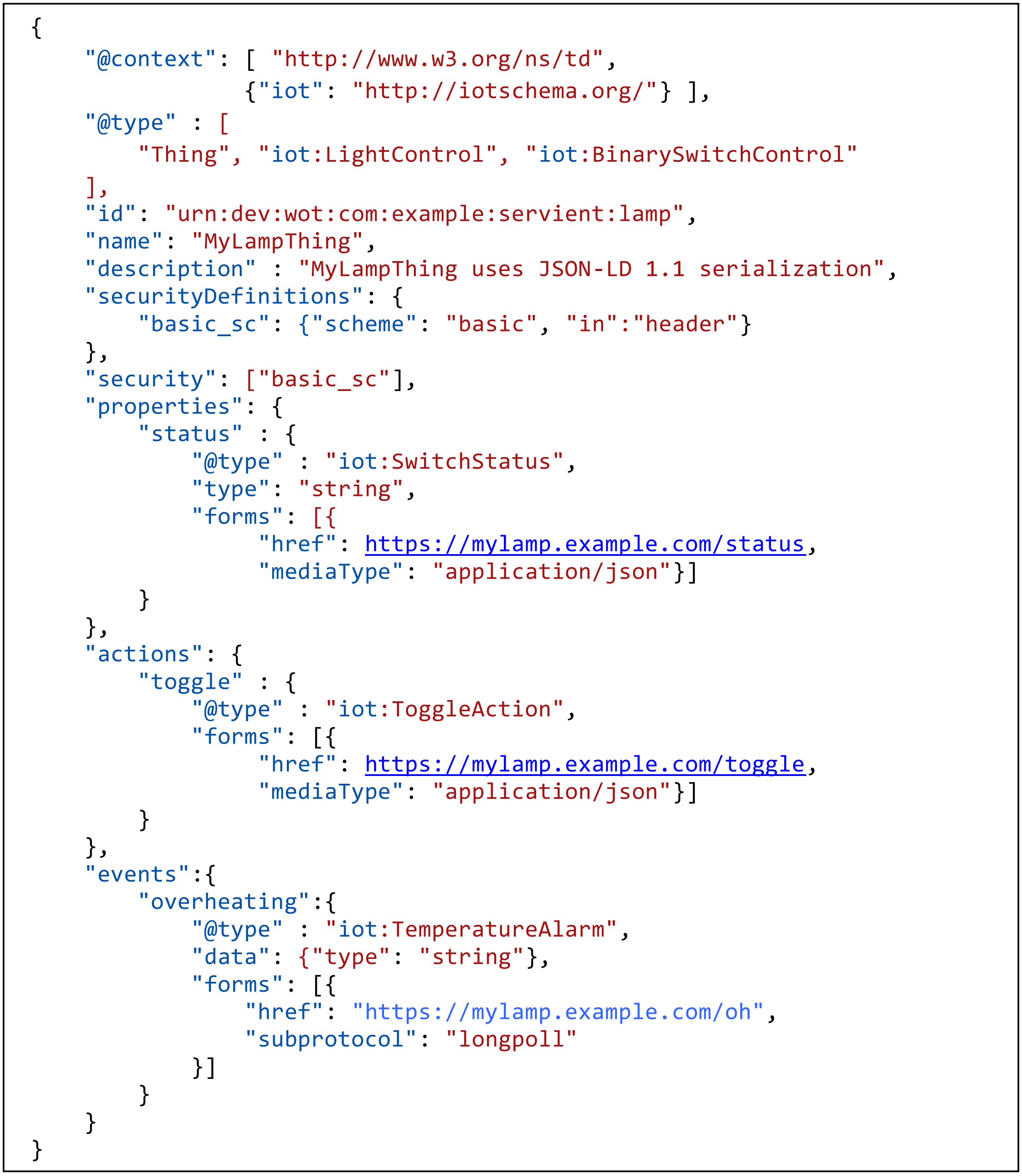


Figure 10: iotschema.org - Thing capability model



```
{
    "@context": [ "http://www.w3.org/ns/td",
                  {"iot": "http://iotschema.org/"} ],
    "@type" : [
        "Thing", "iot:LightControl", "iot:BinarySwitchControl"
    ],
    "id": "urn:dev:wot:com:example:servient:lamp",
    "name": "MyLampThing",
    "description" : "MyLampThing uses JSON-LD 1.1 serialization",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "@type" : "iot:SwitchStatus",
            "type": "string",
            "forms": [{
                 "href": https://mylamp.example.com/status,
                 "mediaType": "application/json"}]
        }
    },
    "actions": {
        "toggle" : {
            "@type" : "iot:ToggleAction",
            "forms": [{
                 "href": https://mylamp.example.com/toggle,
                 "mediaType": "application/json"}]
        }
    },
    "events":{
        "overheating":{
            "@type" : "iot:TemperatureAlarm",
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Figure 11: Thing Description annotated with iotschema.org

The complete iotschema.org can be accessed online[7] and browsed online[8]. Semantic specifications for smart objects, contributed by SEMIoTICS, are also online, see for example semantic description for a camera (used in use case 1)[9]. Semantic specifications for brownfield devices will be mapped to the semantic model of iotschema.org and will be published in the next version of this deliverable (D3.9).

[7] https://github.com/iot-schema-collab/iotschema
[8] http://iotschema.org/docs/full.html
[9] http://iotschema.org/Camera

3.1.3 JSON-LD

JSON for Linking Data (JSON-LD) [5] is a serialization format for JSON (a widely adopted serialization and messaging format on the Web). JSON-LD enables JSON data to be interlinked and structured based on semantic models. Thus, it brings the Linked Data paradigm to JSON. There exist implementations and tools for processing and querying JSON-LD data.

3.1.4 Semantic Integration in SEMIoTICS

In the previous section we have described main building blocks that we will use as technology blocks in realization of semantic integration in SEMIoTICS. These building blocks are to the large extent based on standards and are widely adopted in IoT communities.

Thing Description will be used to semantically describe field device resources, their interfaces, security meta-data, and so forth. For some of brownfield devices there exist already various kinds of device descriptions. Therefore, to reuse existing semantics we will need to provide a semantic mapping from brownfield semantic models into IoT semantic models, as expected by W3C TD and iotschema.org.

The mechanism of Binding Templates we will use in SEMIoTICS in order to provide bindings for various brownfield protocols (e.g., S7comm, Profibus[10], Modbus[11] etc.) into common Web application layer (e.g., HTTP, CoAP etc.).

In SEMIoTICS we can use the WoT Scripting API to expose Things (field devices) that have been integrated over Binding Templates and described with Thing Descriptions. In this way we can provide a uniform standardized access to Thing's and their data, which can greatly reduce development effort for IoT applications at the Edge and in the Cloud.

Thing Descriptions are serialized to JSON-LD as it offers a good trade-off between machine-understandable semantics and usability for Web developers.

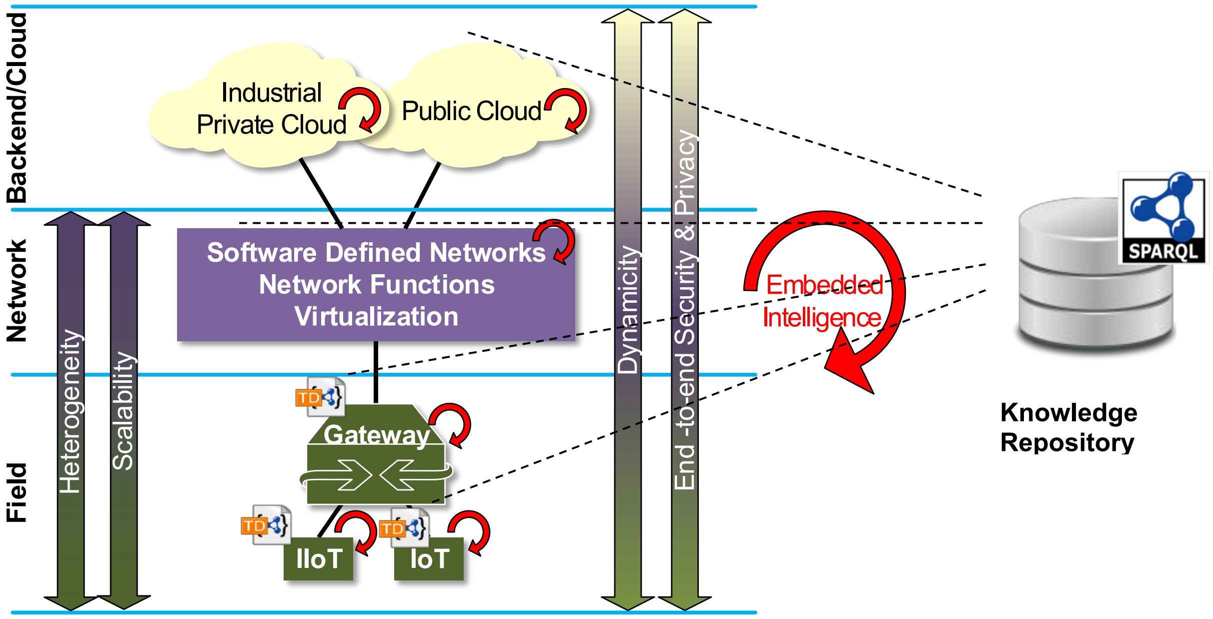


Figure 12: W3C WoT in SEMIoTICS

[10] https://www.profibus.com/
[11] https://en.wikipedia.org/wiki/Modbus

3.2 Mapping the Semantics from Brownfield Automation Devices into IoT Semantics

3.2.1 Industrial Domain

In Section 2.1 we stated the goal of semantic integration, which is to enable realization of new IoT applications that have not been envisioned at the time of engineering an existing automation system. To this goal, we have discussed a common semantic access layer between brownfield devices and new IoT devices. In order to integrate both kinds of devices we need to map and integrate semantics from existing brownfield devices into IoT or IIoT application semantics. Only then it will be possible to discover required Things when developing an application, and to put them into semantically-correct interactions.

Figure 13 depicts the concept for the integration of brownfield automation systems into the industrial IoT domain by considering semantic aspects of systems. We distinguish a few layers of concern divided into two blocks, i.e., existing automation systems and IIoT-based automation systems. The existing semantics from brownfield automation systems is largely contained in various forms of device descriptions (see *Field Device Semantics* in Figure 12). Field Device Semantics is standardized throughout different standards such as Electronic Device Description[12] (EDD) and its EDD Language[13] (EDDL), GSD[14], FDT/DTM[15], IO Device Description[16] (IODD)etc. AutomationML[17] is also to be mentioned here as a standardized data format based on XML for the storage and exchange of plant engineering information.

Another level of semantics is introduced by data models from various *Field Communication* protocols, see Figure 13. In the industrial domain, common protocols are for exampleHART[18], PROFIBUS[19], Modbus, and many others.

In the second block we have IIoT-based automation systems and their semantics. Different IoT ecosystems are based on different IIoT information models. For example, OPC UA[20] is an established standard in this area. Its model enables information integration, where vendors and organizations can model their complex data and take advantage of the service-oriented architecture. W3C WoT Thing Description is another prominent candidate in this layer, see Section 3.1.1.1 for more information. Apart from the standard-based models, there exist IoT information models from ecosystems, which are provided by large industrial players. One such example is Siemens' MindSphere[21] IoT asset model. Finally, IIoT information models need to be extended with application-level, domain-specific semantics (see Figure 13). At this level there are various candidates, and two prominent ones are: iotschema.org (see Section 3.1.2) and OPC UA Companions[22].

[12] http://www.eddl.org/
[13] https://webstore.iec.ch/publication/23481
[14] https://www.profibus.com/products/gsd-files/
[15] https://fdtgroup.org/
[16] http://www.io-link.com/
[17] https://www.automationml.org/o.red.c/home.html
[18] https://www.fieldcommgroup.org/technologies/hart
[19] https://www.profibus.com
[20] https://opcfoundation.org
[21] https://siemens.mindsphere.io/en
[22] https://opcfoundation.org/forum/opc-ua-companion-standards/

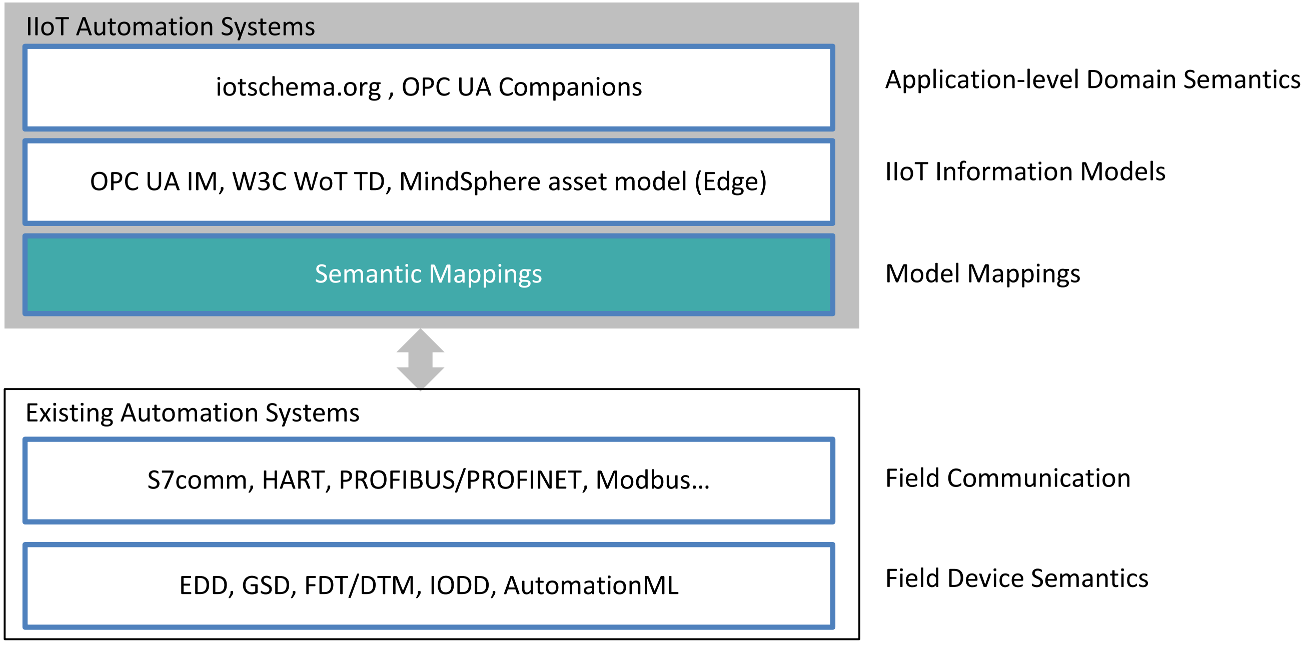

IIoT Automation Systems
iotschema.org , OPC UA Companions
OPC UA IM, W3C WoT TD, MindSphere asset model (Edge)
Semantic Mappings
Application-level Domain Semantics
IIoT Information Models
Model Mappings
Existing Automation Systems
S7comm, HART, PROFIBUS/PROFINET, Modbus…
EDD, GSD, FDT/DTM, IODD, AutomationML
Field Communication
Field Device Semantics

Figure 13: Mapping the existing Brownfield semantics Into the IIoT semantics

Semantic Mappings is a layer that we introduce in the SEMIoTICS project with the aim to map and integrate brownfield semantics with IIoT semantics (see Figure 13). In this layer we have to harmonize semantics from a particular brownfield semantic standard with the IIoT standardized semantics, e.g., iotschema.org (including both the semantics and serialization format). Once we have a harmonized model, we will offer this model in chunks that cover a specific domain. The SEMIoTICS IoT Gateway will be able to install these chunks (semantic nodes or packs), and thus to enable an engineer to accomplish the brownfield integration.

3.2.2 Healthcare Domain

A Figure 14 depicts the key semantics relevant for the SARA UC. At the field level SARA solution results from the integration of four subsystems: Body Area Network, Robotic Rollator, Robotic Assistant, Home Automation.

Within each subsystem the communication among the devices is enabled by a specific protocol: the devices belonging to the Body Area Network communicate using the Bluetooth protocol, the devices on board of the Robotic Rollator exchange information using the Controller Area Network Protocol (CAN-BUS), the devices part of the Robotic Assistant uses a proprietary protocol, Home automation devices relies on ZigBee.

The communication between these four subsystems is enabled by the existence of four devices acting as communication bridges between the devices belonging to different subsystems:

- a smartphone enables the communication between the Bluetooth devices belonging to the BAN and the SARA backend services relying on cellular connectivity
- the controller of the Robotic Rollator can communicate both with the hub of the BAN (i.e. a smartphone) via Bluetooth, with the devices on board of the rollator via CAN-BUS, with the Robotic Assistant and the Home gateway via WiFi.
- the Robotic Assistant communicates with the Robotic Rollator and the Home gateway via WiFi
- the Home gateway enables the communication with the ZigBee devices belonging to the Home Automation subsystem, with the Robotic Rollator and the Robotic Assistant via WiFi and with SARA backend services using landline connection.

However, as already introduced by the previous sections, the existence of communication gateways does not guarantee per se the possibility for the SARA application (or any other IoT application) to access in a uniform way the device belonging to different subsystems (e.g. the possibility for a Gait Analysis function to read in a uniform way both the IMUs on the smartphone and the IMUs on the robotic rollator): there will be the need to develop Semantic Mappings to enable this.

The Semantic Mappings developed in the context of SEMIoTICS cannot ignore the existing semantics which appear relevant of the SARA UC: HL7 FHIRE[23], LOINC[24], SNOMED CT[25], SAREF[26], UniversAAL[27], SmartBAN[28], CORA[29], AuR[30], CLoE-IoT[31].

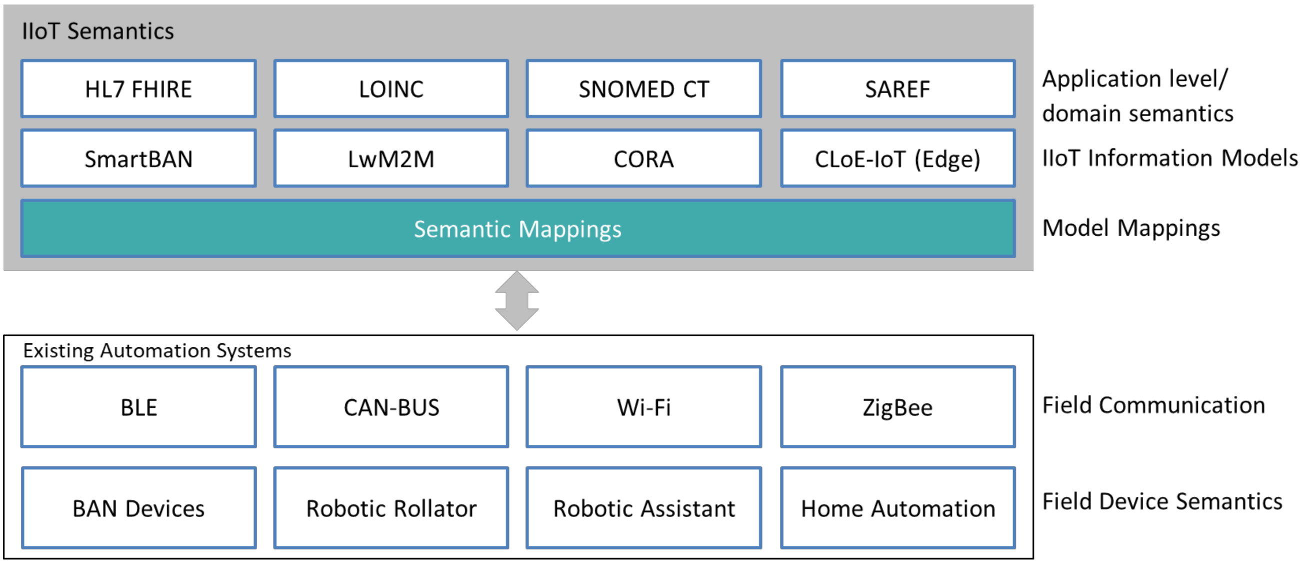


Figure 14: Mapping the existing semantics to the IIoT semantics-SARA Health Scenario

3.2.3 IHES Generic IoT

The IHES Generic IoT UC will be implemented as a demo implementing use case scenarios identified in D2.2 of SEMIoTICS project. The demonstrator will focus on distributed intelligent systems. As of today, this envisioned approach is not yet very common in IoT systems where a more centralized cloud-centric approach is preferred. This implies that currently there are no specific ontologies or semantic patterns available to code the interactions of those intelligent devices. In this respect SEMIoTICS will be the perfect testbed to introduce these new communication patterns. The only limitations will be related to the actual middleware available for the simple MCU sensing units that should be adapted to the new semantics. A 1-to-N reliable protocol should be available to allow these devices to smoothly cooperate together. Sadly, current cloud-oriented middleware (e.g. Microsoft Cloud Azure or Amazon AWS) could not be used since they are designed specifically for the cloud-centric approach. Thus, their semantic, patterns, enabling reference frameworks, could not be used: a new reference design and a completely new semantic will be thus defined in SEMIoTICS for these specific IHES UC devices & intelligent nodes. An IHES system will be a local / lightweight deployment of data analytics and local processing able to interoperate at semantic level with existing legacy components by e.g., exploiting

[23] http://www.hl7.org/fhir/summary.html
[24] https://loinc.org/get-started/what-loinc-is/
[25] https://www.snomed.org/about
[26] http://www.etsi.org/technologies-clusters/technologies/smart-appliances
[27] https://www.universaal.info/
[28] http://www.etsi.org/technologies-clusters/technologies/smart-body-area-networks
[29] https://standards.ieee.org/develop/project/1872.2.html
[30] http://www.ieee-ras.org/industry-government/standards/autonomous-robotics-group
[31] http://cloud.esl.eng.it/cloe-iot/#/main

a newly defined interaction pattern, and network-related metadata for a Thing (to be exposed by providing a WoT Thing Description document).

Such kind of network-related metadata involves information on device registration on SEMIoTICS, that IHES nodes will convey over standard MQTT bridge, such as:

- **name**: the name of the device on the network;
- **MAC**: the physical network address of the device;
- **location**: where the device is located;
- **type**: to indicate the type of the device and sensors implemented;
- **function**: a brief description of the device's functions (e.g. capabilities exported by devices and accessing entry points);
- **interfaces**: list of the device's interfaces, their location (endpoint), their type (e.g. MQTT), and if they are secured or not.

Utilizing this data directly from the thing description will enable seamless interoperability between devices in the field, including devices from different domains. In addition to the above, and with regards to exposing the thing via its description, metadata found in the common representation of for the Web of Things can be utilized.

3.3 Metadata related to security, privacy, and dependability

While security metadata does not have a semantic relevance per se, there are two important links:
Firstly, when a new device is connected to SEMIoTICS, two processes will start. Identification and authentication will ensure that the newly connected device is benign (or otherwise will be rejected). Different modes of identification and authentication will be described in detail in D4.5. However, authentication only ensures a logical connection, not a semantic one. Therefore, semantic bootstrapping is required, and needs to be compatible with the more security-oriented authentication process.
Secondly, during the authentication and semantic bootstrapping various metadata will be collected and stored together. The following security metadata will be stored:

- **Mode of authentication**: Several modes of authentication are supported by SEMIoTICS, providing different levels of security. Examples include password-based authentication, two-factor authentication, and smart card-based authentication, and no authentication. The mode of authentication affects the trustworthiness of the data delivered by the respective device. Thus, this information is relevant, e.g., for patterns: A pattern can require that particularly sensitive data may only be communicated via devices using strong authentication mechanisms.
- **Identity provider**: The security manager in the backend supports several identity providers, both internal and external. For each device authentication, the identity provider will be stored. This information is relevant in the scenario that at some point in time an identity provider is identified as having been compromised: In this case all connections authenticated using the compromised identity provider must be terminated.
- **Time of authentication**: Security metadata will include the time at which a new device has been authenticated. This information may become useful for example during investigations into an attack at a later time.
- **(Network) Location**: The security metadata also contains information about to which SEMIoTICS device a new device was connected, e.g. to which gateway a new sensor was connected.

Metadata for SPDI patterns are specifications regarding each one of the properties; some of which need to be monitored to enable the functionality of the pattern engine. The pattern engine has to effectively support the SPDI properties; to that end monitoring of certain metadata must be facilitated. It should be noted that the semantic metadata presented herein can be exposed through the descriptions of the different components that will be used to instantiate IoT orchestration Recipes.

For the purpose of security, an example of metadata that could be observed is how many login efforts were attempted; using this information the patterns could activate certain security measures to detect/prevent/mitigate an attack. For privacy, monitoring of data encryption/decryption could be a reasonable hint to indicate that a malicious entity leaks data. Additionally, monitoring the encryption metadata to check if encryption is applied on data at rest and in transit as described by the patterns is essential. In regard to dependability, the patterns could monitor the reliability values of a component and cross check them with the DependCert list to examine if it operates according normally. In terms of interoperability, worth monitoring are metadata such as, protocols, data formats, semantic and programming interfaces that drive the initialization of the communication. Moreover, monitoring what kind of certification the communicating components have could improve the efficiency of establishing the connection, in terms of time and computational power.

In this context, some metadata information that can be useful in the context SPDI pattern monitoring and verification are listed below.

For **Security**, some security metadata such as tokens, domains, authentication and accountability interfaces have already been mentioned in this section, but additional information related to SPDI properties may need to be included. Some examples are listed below, while the decision, which subset will be included in each case and the exact content within the fields, will depend on the exact setup and patterns monitored for the given use case.

Confidentiality metadata may include information such as if encryption is applied (on data at rest and data in transit), what type of encryption should be used and the size of keys for the chosen cipher suite. **SecCert** can also be used to list the collection of certificates that the specific component bears.

Moreover, to guarantee **Integrity** in all states of data, metadata that indicate relevant information must be facilitated, this involves toggles that i) data integrity checks in transit using specific secure protocols (e.g. TLS) ii) data at rest integrity checks, such as hardware (e.g. TPM) or software (e.g. filesystem level) ones iii) and data in processing if integrity checks are included at the function level.

Further, to guarantee **Availability**, metadata that displays the degree to which a device/service operates and is accessible at any given time (e.g. uptime). This data may include information on network components, such as SDN controllers and nodes, alternate paths, signal strength, noise etc. As in all cases, the exact measures and thresholds of availability have to be defined on a per case basis.

Privacy metadata can be utilized to ensure data is handled in a private manner according to data protection laws. Some potential instances for this type of metadata are: **DataSensitivity** that indicates if data handled by this device is sensitive (e.g. health relate data) and if additional measures are used (true/false) to protect it (e.g. pseudoanonymization, anonymization), with an additional **hasConsent** field with a Boolean value confirming if the data subject has given his/her consent; **DataUsage**, that provides information on when the data was acquired, for how long it will be kept, if there are there any duplicates, if it is marked for safe deletion etc.; Finally, **PrivCert** metadata facilitates a compilation of certifications that the component subsequently handling the data may hold.

Dependability metadata can be used to guarantee devices and services run in a way they are supposed to. This type of metadata involve **reliability** values, such as delay, packet losses etc. (for hardware these values are usually provided by the manufacturer); **FaultResponse,** that marks information regarding if and what operations are used to ensure that the component continues normal operation despite software or hardware faults (e.g. replicated paths for forwarding traffic in parallel); finally, **DependCerts,** that contains a list of certifications that a device/application holds, if any.

Interoperability metadata can be used to specify information with regards to establishing seamless connectivity between, for example, two devices. This includes, configurations, protocols, data formats, information models, ontologies, common semantic and programming interfaces etc.; finally, **InteropCerts,** which accommodates a list of standards and certifications that a device/application integrates related to its interoperability (e.g. two devices might build upon the same standard, which makes them interoperable by default).

In addition to the above, extra fields to hold information against various types of attacks may also be used. These could include **AuthenticationRequests,** providing information on successful/unsuccessful login attempts; **AuthorisationRequests,** that hold information on successful/unsuccessful authorization checks to get access to a protected endpoint or invoke a protected operation; **InteractionRequests**, that contains data used to limit, and track requests on open interfaces (e.g. to mitigate DDoS attempts on exposes interfaces); or **ResourceUsage,** that provides information on use of network and computer resources (e.g. to deal with resource exhaustion attacks). In cases of sensitive data or data of critical importance from a security perspective, **DataAccess** can be defined to track access level of the data, who & when accessed it, effectively providing means for auditability and accountability.

Details on the concept of security metadata for semantic bootstrapping will be elaborated in Deliverable 4.12.

3.4 QoS-related metadata

In SEMIoTICS the monitoring of QoS related parameters will be driven by Architectural Patterns (see Task 4.1, Deliverable D4.1), much like the SPDI properties mentioned in Section 3.3 above. In this context, QoS requirements of the various IoT orchestrations supported by SEMIoTICS are defined in Recipes and then encoded as patterns for monitoring and enforcement.

To achieve such QoS-aware orchestrations, the devices should, ideally, expose through their Thing Description typical QoS-related information such as the Goodput (i.e. application-level throughput), Packet loss (percentage of packets lost to packets sent), Errors (detected packets corrupted), Latency (i.e. network delay), Packet delay variation (i.e. packet jitter), and Out-of-order delivery (delivery of data packets in a different order from which they were sent). In addition to the above, the QoS-related properties of Dependability and Availability, as analyzed in the context of Section 3.3 above are also considered. In all cases, the descriptions could include certain values to classify their performance (e.g. latency < 1ms), as well as point to monitoring interfaces, if available, that would allow the orchestrator to monitor the real-time value / performance of these QoS parameters.

3.5 Approach to Achieve the Bootstrapping and Semantic Interoperability at the Field Layer

In this section we summarize the SEMIoTICS approach to achieve the bootstrapping and semantic interoperability as presented by various parts of Section 3. Let us first do this for the problem of semantic interoperability. **Semantic interoperability** is concerned with the ability of information systems to exchange data with unambiguous, shared meaning. Our goal is to achieve semantic interoperability between brownfield and greenfield devices. The problem consists of two parts. One that is concerned with interoperability at the **communication level** (different devices communicate over different protocols), and another one that is concerned with **semantics** (different devices use different semantics, data- and serialization- formats etc.).

Throughout Section 3 we said that SEMIoTICS approach would be built on W3C WoT standard and iotschema.org. In order to solve the communication problem, we will use the WoT Binding Templates (see Section 3.1.1.2). Binding Templates, together with Binding Implementations, will enable brownfield devices from SEMIoTICS use cases to be interoperable at the communication level with WoT-enabled (greenfield) devices. We will implement Binding Implementations for those protocols that we need in our use cases and are at the moment not covered by the open-source implementation from W3C WoT community (e.g., S7comm protocol that we need in use case 1).

In order to tackle the challenge of semantic integration, SEMIoTICS will build on W3C standardized Thing Description (see Section 3.1.1.1), and iotschema.org (see Section 3.1.2). Semantic integration is a more difficult problem than the integration at the communication level. In order to solve this challenge, we have to have IoT semantic models that also harmonize semantic models from brownfield systems. For two examples how this harmonization or mapping should be accomplished see Figure 13 and Figure 14. However, it is worth noting that this process should be standardized. Otherwise no user, who already uses standardized brownfield semantics, will use not-standardized IoT semantics. The standardization process is however long and needs to be accomplished by standardization organizations (not by an EU project such as SEMIoTICS). That is why we decided to build our approach on iotschema.org (see Section 3.1.2), where we are directly involved in an ongoing standardization process of IoT semantic models. Thus, our goal is to extend iotschema.org with semantic specifications as required to cover semantic brownfield integration (as defined by SEMIoTICS use cases). This task includes mapping from an existing (brownfield) models or creating new semantic models when there is no brownfield semantics available. Once the IoT (harmonized) semantic models are available by iotschema.org we have to make these models easily applicable in different domains and use cases. Semantic models are in general hard to use by non-experts. That is why in SEMIoTICS we will create semantic nodes (packs) out of certain iotschema.org semantic models. Semantic packs will serve as chunks of models that can be used for semantic mapping and configuration of brownfield devices. For example, for each interaction (data point) of a brownfield device, the user can use a semantic pack to configure that interaction with the iotschema.org IoT semantics. The semantic mapping (configuration) will be manual process. We will provide tolling, which easies this task (e.g., a user provides inputs or chooses an offered enumeration value from a semantic template). Once the semantic mapping (configuration) is accomplished, the tool will automatically generate a W3C-valid Thing Description, together with correct iotschema.org semantics (see Section 3.1.2) and the valid serialization format inside (see Section 3.1.3). In that phase, the semantic interoperability based on W3C WoT standard and iotschema.org will be enabled at the SEMIoTICS field layer.

It is wort of noting that the semantic interoperability as described here is based on light-weight semantic models. iotschema.org is an extension of a well-known schema.org. Thus, we rely on RDF Schema[32] as a semantic formalism. RDF(S) fully satisfies our need as the role of semantics in the project is to make data from brownfield devices *integrated* and *interoperable* with data from green field devices, and further on, the role of semantics is to enable creation of *common application layer* based on such unified data. Moreover, RDF(S)-based semantics allows us to use machine reasoners to *match* requirements from applications with capabilities of devices, as well as to *validate* semantically annotated Thing Descriptions with SHACL Shapes[33].

Let us now reflect our approach to the task of **bootstrapping**. The goal of this process is to integrate a new (brownfield or greenfield) device in the SEMIoTICS platform, and to enable creation of new applications. Essentially this task is a process that implements the tasks of integration at the communication- and semantic-level (see above). The process is broken into a sequence of steps, see Figure 16. For greenfield devices, which are described with a TD, these steps will be automated (plug & play bootstrapping), whereas for the brownfield devices the process of mapping/harmonization will still involve a manual work. However, this task will be done once, at the design time. The outcome can be reused for brownfield devices of the same type. The sequence of steps from Figure 16 are implemented in SEMIoTICS IIoT Gateway, see Section 4. With this regard, our goal to accomplish the semantic bootstrapping at the SEMIoTICS field layer is achieved.

Finally, we would like to emphasis the importance of the subject of the task T3.3, i.e., semantic-based bootstrapping and interfacing at the field level in the **SEMIoTICS use cases**.

In the wind energy use case (UC1) the goal is to create new, added-value, applications that have not been envisioned at the time of creating the automation systems, which controls a wind turbine. For these new applications we may need new additional field devices. The challenge is thus to integrate the existing (brownfield) systems with newly added (greenfield) devices. Brownfield and greenfield devices communicate over different protocols (e.g., S7comm vs. HTTP), they also often use different serialization formats (e.g., XML vs. JSON), and they adhere to different semantics (e.g., no semantics or machine not-interpretable semantics vs. semantics based on iotschema.org). Apart from this, in UC 1 it is very important to easily scan network for newly plugged devices and register them in plug and play fashion.

In the SARA use case (UC2) we must deal with a wide range of device semantics (different kinds of devices with different functions), data formats (syntactic representations), measurement unit conventions (for sensor

[32] RDF(S): https://www.w3.org/TR/rdf-schema/
[33] SHACL: https://www.w3.org/TR/shacl/

readings), and communications protocols (e.g. WiFi, ZigBee, Bluetooth). Various key aspects of SARA functionality, moreover, require that data from multiple sources is collated, aggregated and/or analysed in a coherent collective fashion. The reliable detection of "fall incidents", for example, may involve the continuous comparative evaluation of data from wearable IMU devices, RR handle-mounted pressure sensors and RA video cameras (among others).

In the third use case (UC3) Local Embedded Analytics on IHES Sensing Units creates unprecedented distributed event-driven type of semantically complex data-patterns messages that require to be properly handled in order to achieve scalability and semantic interoperability. From this perspective SEMIoTICS ecosystem with its pattern driven approach will be the perfect testbed where experiment the local analytics / edge computing approach in real life conditions.

The semantic interoperability within the SEMIoTICS framework is a key enabler on which the IHES system is leveraged. To interface the IHES sensing units in a coherent manner, we have evaluated the adoption of the W3C WoT standard to interface the connected the microcontrollers to the framework. A full support of the standard is not feasible, e.g. the WoT servient is not deployable on a MCU due to its very limited resources. Thus, as an alternative, in order to provide to the ecosystem the whole set of functionalities provided by the system, we have considered to develop within SEMIoTICS specific dedicated component that will act as a bridge from the low level MQTT messages to the other components in SEMIoTICS, similar in principle to what has been planned for the brownfield devices within the UC1 scope.

4 IMPLEMENTATION OF SEMANTICS IOT GATEWAY

The first version of SEMIoTICS deliverable D3.3 provides a concept for device bootstrapping and semantic integration of field level devices. In this (next) version of the deliverable the concept has been implemented. This section we start by reviewing information about architectural components of IoT Gateway, which are required for the realization of the presented concept. These components are presented in Figure 15: SEMIoTICS IoT Gateway, and are agreed with other SEMIoTICS project partners in the work on SEMIoTICS architecture, see Figure 1 and Section 2.3 in SEMIoTICS deliverable D2.4. In the scope of this deliverable we will consider the following components: *GW Semantic Mediator*, *Local Thing Directory*, and *Semantic API & Protocol Binding*, see Figure 15: SEMIoTICS IoT Gateway.

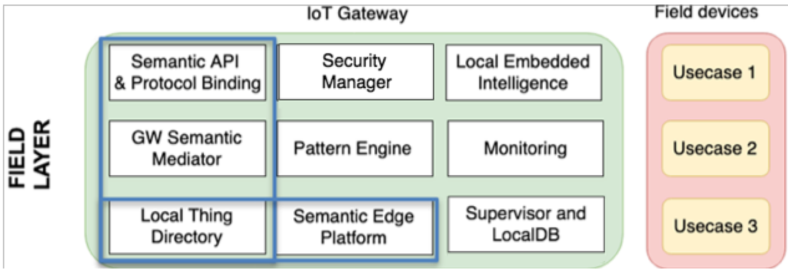


Figure 15: SEMIoTICS IoT Gateway

Figure 16 shows an updated sequence diagram of activities that occur during the bootstrapping process. The goal of this process is to integrate a new device in the SEMIoTICS platform by using SEMIoTICS IIoT Gateway. Once this process is completed, it will be possible to create new applications based on data from the new device, as well as the data from other available devices in the platform. In order to achieve this goal, the gateway needs to make the device data accessible, and it has to provide full semantic description of the device, i.e., semantics about device capabilities, its data, communication protocols, contextual information (e.g., location, domain of use) etc.

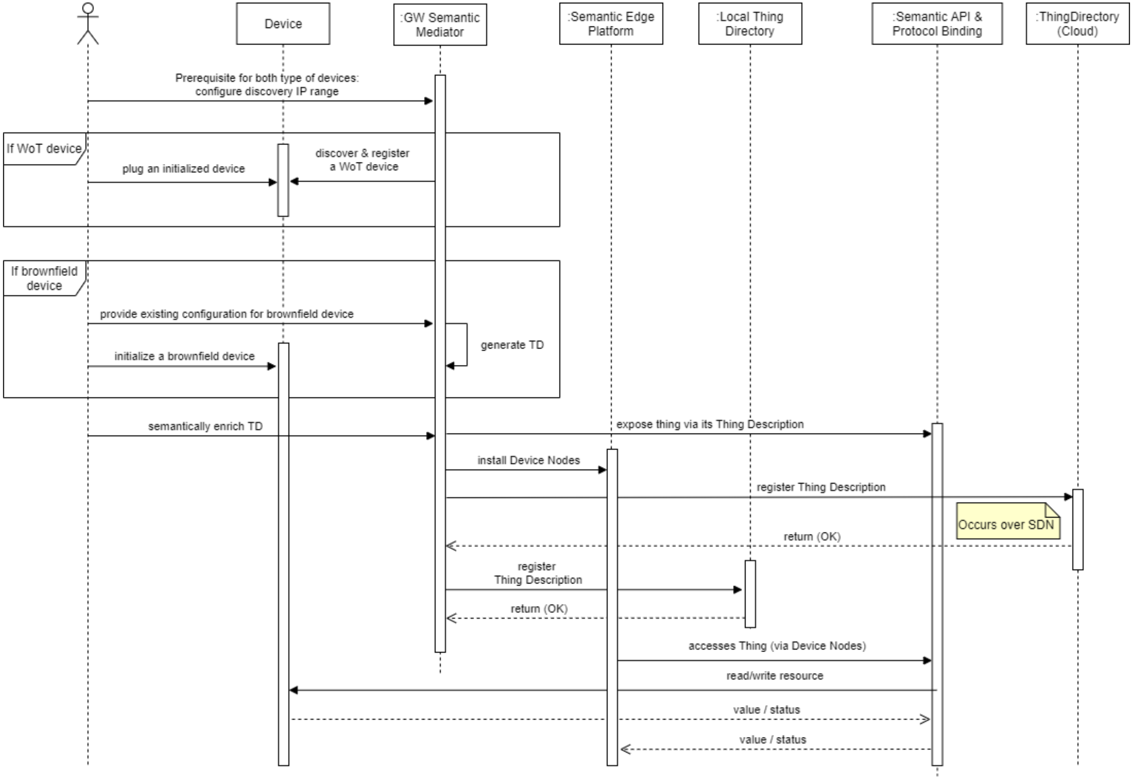

Device
:GW Semantic Mediator
:Semantic Edge Platform
:Local Thing Directory
:Semantic API & Protocol Binding
:ThingDirectory (Cloud)
Prerequisite for both type of devices: configure discovery IP range
If WoT device
plug an initialized device
discover & register a WoT device
If brownfield device
provide existing configuration for brownfield device
generate TD
initialize a brownfield device
semantically enrich TD
expose thing via its Thing Description
install Device Nodes
register Thing Description
Occurs over SDN
return (OK)
register Thing Description
return (OK)
accesses Thing (via Device Nodes)
read/write resource
value / status
value / status

Figure 16: Sequence diagram for Bootstrapping and interfacing SEMIoTICS field level devices

In the bootstrapping process we distinguish two different classes of device. The first class consists of devices that already have a Web-based RESTful interface and are described by W3C Thing Description (WoT devices). The second class comprise of all other devices that yet need to be made accessible over a Web-based RESTful interface (brownfield devices). These devices do not have a semantic description, or certain semantic meta-data exists, but needs to be mapped to standardized semantic IoT models. This is a case, for example, with brownfield devices. They may have various forms of device descriptions, including standard-based descriptions (e.g., EDD). However, in order to realize IoT applications, it is convenient to map these brownfield descriptions into description based on standardized IoT semantic models. They are two reasons for this. First, IoT applications are typically cross-domain applications. Brownfield device descriptions focus on certain domain. In order to integrate such devices with devices from another domain, we need to have harmonized semantic models that cover multiple domains. Second, brownfield device descriptions usually need to be enriched with additional semantics (e.g., to support new classes of IoT application). Thus, we need richer IoT models and a mapping approach for brownfield device descriptions.

Let us consider now a sequence diagram of activities that occur during the bootstrapping of WoT device, see Figure 16. The user performs the first step during the initialization of a new device. This assumes provision of information such as an IP address, device capability, domain of use, location etc. Since the device already has a Thing Description (TD), this information is directly put in its TD. The device can then be registered with SEMIoTICS IIoT Gateway (with GW Semantic Mediator, which is an internal component of the Gateway).

If a brownfield[34] device needs to be initialized, then a user in addition to previously mentioned information needs to specify metadata related to the communication protocol and the encoding format. This information will be important part of a Thing Description and is used by SEMIoTICS IIoT Gateway to realize a protocol binding. The protocol binding is a process, which enables a device brownfield protocol (e.g., Modbus) to be

[34] Brownfield refers to the implementation of new systems to resolve certain problem while accounting for established systems.

mapped to an IoT application protocol, e.g., HTTP, CoAP, MQTT etc. If the device has a brownfield configuration meta-data, in this process, it will be mapped into a standardized Thing Description of the device (see *:GW Semantic Mediator* in Figure 16). The brownfield configuration meta-data needs also to include communication meta-data (e.g., which protocol the device communicates over). The information will be used by the Mediator for the protocol binding.

For the sake of simplicity, Figure 16 shows the bootstrapping process only from a semantic perspective (as this is the focus of task T3.3). However, in parallel the process of authentication of a device also must take place. For this purpose, the device will identify itself to the security manager in the gateway. The authentication process will be described in detail in D4.5. If authentication was successful, the security manager then generates session keys for communication between device and semantic mediator. The semantic mediator may provide the Thing Description to the Local Thing Directory if and only if it received the respective session keys; if the Semantic Mediator does not receive the session keys, that means that the authentication failed and the device may not be used.

Once the registration is completed, a user may configure a device and provide additional semantic annotations (see the interaction *semantically enrich TD* in Figure 16). These annotations are typically contextual information such as location of a device, its specific capability or configuration. This task completes the realization of semantically enriched Thing Description. The device can then be registered with SEMIoTICS IIoT Gateway (with *GW Semantic Mediator*). The mediator will create a Device Node for each interaction pattern of the device. Device Node is a programmable component that enables interaction with the device. The Mediator automatically generates Device Nodes, based solely on information from devices' TDs. Generated Device Nodes can then be installed in *:Semantic Edge Platform*. The mediator exposes capabilities of these devices over a standardized Web API and runs on a W3C WoT servient (see Section 3.1.1.3). A W3C WoT servient in Figure 16 is represented with SEMIoTICS component called *:Semantic API & Protocol Binding*. This API is used for realizing IoT applications at the Edge level too. After exposure of a device over a WoT servient, the device can be used in Edge and Cloud-based applications via interactions provided by its Device Nodes. Device Nodes represent a means for an application to programmatically access device's functionality via *:Semantic Edge Platform*. Each interaction from *:Semantic Edge Platform* will create a call to the WoT servient, which will forward the call to a Thing (device), see Figure 16. Note however that such interactions are done over a unified and standardized WoT API, no matter whether we initiate an interaction for a WoT device or brownfield device. In case of an interaction with a brownfield device, the forwarded call is performed over an implementation of a protocol binding. In the context of this deliverable we implemented such a protocol binding for S7comm protocol. With this implementation at hand, a greenfield device can communicate with Siemens S7 controller (used in use case 1) over a W3C WoT servient.

The final version of a Thing Description, created (or updated) in the procedure described above, is stored in *:Thing Directory* and *:Local Thing Directory*. :*Local Thing Directory* runs on the gateway and can be queried for all semantic information related to attached devices. The Gateway also registers all locally available TDs by *:Thing Directory* in the Cloud. In this way all semantic information about field level devices is also available in the backend. This information will be used by Recipe Cooker to create IoT application at the Cloud level.

For the final stage of Figure 16, it has to be noted that this figure shows only the semantic perspective. Not every user or application may be permitted to connect to a Thing. Security policies and SPDI patterns will be able to specify limitations on which applications and users may use a Thing. For this purpose, the Security Manager will evaluate such access requests and deny them if necessary.

It is also worth of noting that certain activities in the sequence diagram are accomplished over Software Defined Network (SDN), see Figure 16.

In comparison to the sequence diagram for bootstrapping and interfacing of field devices in deliverable D3.3, here we have added a component called Semantic Edge Platform (SME), see Section 3.5.8 in deliverable D2.5. SME eases the interaction with SEMIoTICS IoT Gateway (graphic user interface for network scanning and user input, e.g., IP address range etc.), including the interaction with Local Thing Directory too.

Figure 17 shows the current set-up for the implementation of the industrial use case, see Section 2.1. With respect to the three levels in SEMIoTICS (Field, Network, and Backend/Cloud, see Figure 12), in this set-up we consider only the Field level, as that is the level where SEMIoTICS IoT Gateway resides.

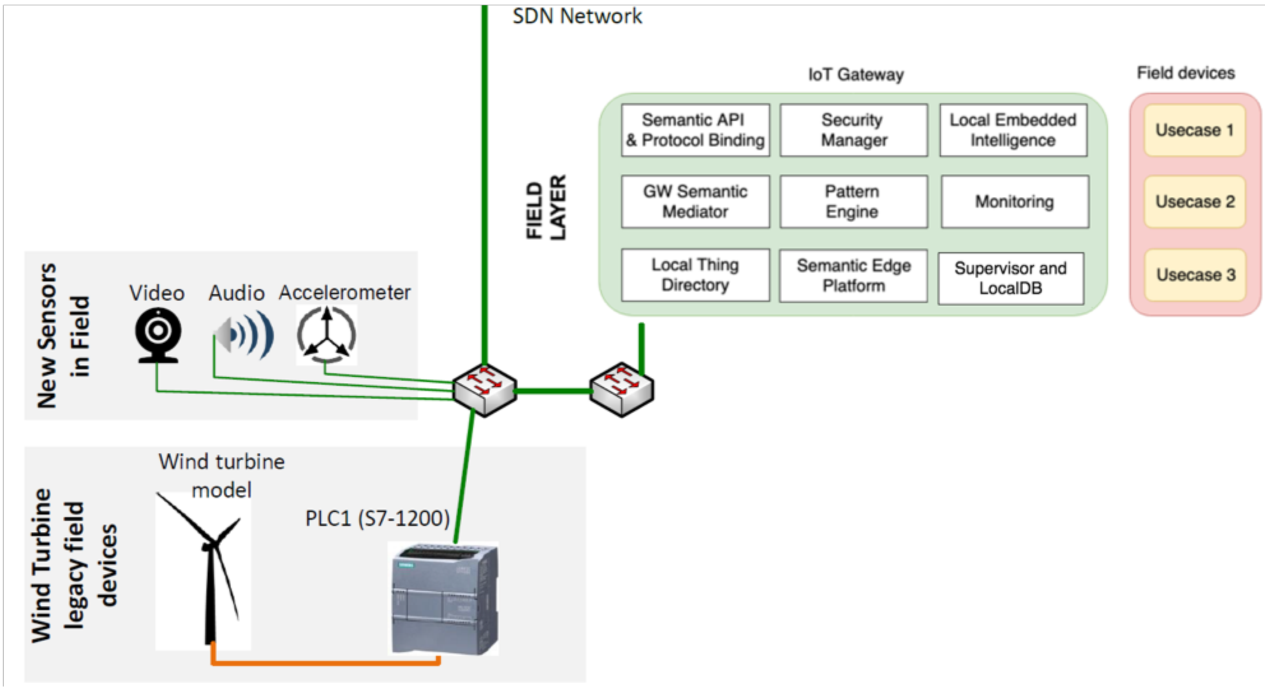


Figure 17: Current implementation setup for industrial use case (field level)

Figure 17 shows two groups of devices, required for the implementation of the Wind Energy scenario. These are devices from an existing wind turbine (Brownfield Devices), and new IoT devices that may be added to the system as an extension (i.e., a camera, microphone and accelerometer). Values from wind turbine sensors and actuators are exposed over a SIMATIC S7 controller or an OPC-UA server. Sensors and actuators, from an IoT device, may be exposed over a CoAP, MQTTT or HTTP server. For example, we may consider a Raspberry Pi (RPi) device or similar one to provide access to these sensors and actuators, and its data. The role of an IoT Gateway is to expose functionality of field devices over a uniform interface with a clear semantics, i.e., machine interpretable descriptions of field devices. For that purpose, we use the W3C Web of Things with its Thing Description and iotschema.org. Thus, there is a W3C WoT servient that runs on SEMIoTICS IoT Gateway. Siemens Nanobox[35] has been used as the hardware to run the IoT Gateway. Apart from this (southbound) functionality, IIoT Gateway will also be used for transfer of data to MindSphere. That (northbound) functionality will be implemented with MindConnect LIB[36] in deliverable D5.7. The semantics created in the Field level and exposed over a Thing Description will be also used for creating MindSphere asset model. In this way we will have a transparent IoT semantics, not only at the Field level, but also across complete SEMIoTICS platform.

In the following, we will provide information about the implementation of each component of SEMIoTICS IoT Gateway, marked in Figure 15.

4.1 GW Semantic Mediator

4.1.1 WoT (Greenfield) Devices

[35] https://goo.gl/C2YyJY
[36] https://goo.gl/HhkZqX

The goal of semantic integration (see SEMIoTICS deliverable D3.3) is to enable realization of new IoT applications that have not been envisioned at the time of engineering of an existing automation system. In the current implementation (for greenfield devices) the mediator does need to do much as the device already has (semantically annotated) Thing Description. Thus, its function is just to make a device programmatically accessible in accordance with the meta-data from Thing Description. It means that for each interaction pattern from the TD, GW Semantic Mediator will create a Device Node of the device. Device Node is a programmable component that enables interaction with the device. For example, if a TD of a device contains inclinometer property and an action for IP camera, then the mediator may generate two nodes: one for reading the current inclination data, and another one for streaming the video from the camera. So generated Device Nodes can be installed in Semantic Edge Platform and used in Edge and Cloud-based applications. Table 6 shows our script that generates and installs such Device Nodes.

Table 6: Creating Device Nodes

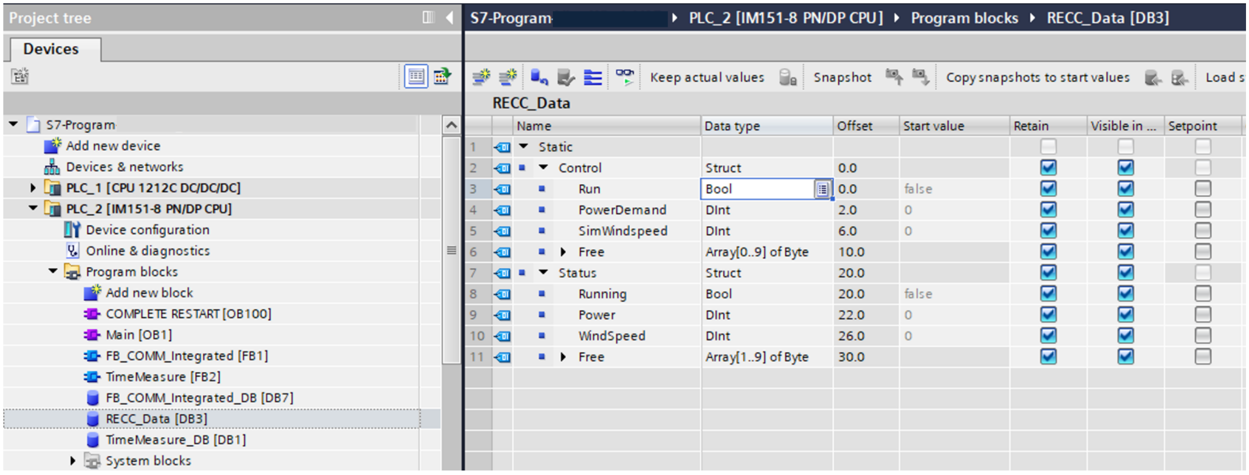
```
npm install
rm -rf ~/.node-red/package-lock.json
rm -rf GeneratedNodes/*
for file in IPshapes/* ; do
    node  NodeGen.js --file=$file
    sleep 2
done
mkdir -p ~/.node-red/SchemaNodes
for d in GeneratedNodes ; do
    cp -R $d ~/.node-red/SchemaNodes/
done
npm install --prefix ~/.node-red ~/.node-red/SchemaNodes/GeneratedNodes/*
npm install
rm -rf ~/.node-red/package-lock.json
rm -rf GeneratedNodes/*
for file in IPshapes/* ; do
    node  NodeGen.js --file=$file
    sleep 2
done
mkdir -p ~/.node-red/SchemaNodes
for d in GeneratedNodes ; do
    cp -R $d ~/.node-red/SchemaNodes/
done
npm install --prefix ~/.node-red ~/.node-red/SchemaNodes/GeneratedNodes/*
```

4.1.2 brownfield Devices

In order to integrate brownfield devices, GW Semantic Mediator needs first to generate a Thing Description (TD) for them. TD can then be used by a WoT Servient (the Semantic API & Protocol Binding component) to expose devices and enable interactions with them. Hence, the first challenge we needed to solve is to create a TD for a brownfield device.

Brownfield devices may be very different as they originate from very different domains. In the context of use case 1 we consider a wind turbine and a Siemens controller, which controls the turbine. Figure 18 shows Siemens engineering station, TIA Portal[37], with all existing information about the wind power automation system. This includes program blocks, communication details, information how devices are connected to each other etc. These artifacts are what in reality exist, and they are our starting point for the brownfield integration.

[37] https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html



Project tree

Devices

- S7-Program
  - Add new device
  - Devices & networks
  - PLC_1 [CPU 1212C DC/DC/DC]
  - PLC_2 [IM151-8 PN/DP CPU]
    - Device configuration
    - Online & diagnostics
    - Program blocks
      - Add new block
      - COMPLETE RESTART [OB100]
      - Main [OB1]
      - FB_COMM_Integrated [FB1]
      - TimeMeasure [FB2]
      - FB_COMM_Integrated_DB [DB7]
      - RECC_Data [DB3]
      - TimeMeasure_DB [DB1]
    - System blocks

S7-Program [redacted] ▸ PLC_2 [IM151-8 PN/DP CPU] ▸ Program blocks ▸ RECC_Data [DB3]

RECC_Data

	Name	Data type	Offset	Start value	Retain	Visible in ...	Setpoint
1	▾ Static				□	□	□
2	▾ Control	Struct	0.0		☑	☑	□
3	Run	Bool	0.0	false	☑	☑	□
4	PowerDemand	DInt	2.0	0	☑	☑	□
5	SimWindspeed	DInt	6.0	0	☑	☑	□
6	▸ Free	Array[0..9] of Byte	10.0		☑	☑	□
7	▾ Status	Struct	20.0		☑	☑	□
8	Running	Bool	20.0	false	☑	☑	□
9	Power	DInt	22.0	0	☑	☑	□
10	WindSpeed	DInt	26.0	0	☑	☑	□
11	▸ Free	Array[1..9] of Byte	30.0		☑	☑	□

Figure 18: Siemens engineering station - TIA Portal

Our goal is to extract information from TIA Portal that is relevant for exposing brownfield devices for interactions with other (greenfield) devices. In particular, in the right-hand side of Figure 18 we see few variables that we want to make available for interactions with other devices, e.g., Power, PowerDemand, Run, WindSpeed etc. Meta-information about our brownfield devices, including information about these variables, their data types, IP address of the controller and others, we can obtain from AutomationML files that describe our field devices and the complete automation system. As shown in Figure 19, the role of SEMIoTICS Mediator is to map relevant meta-data from these files into a TD model[38].

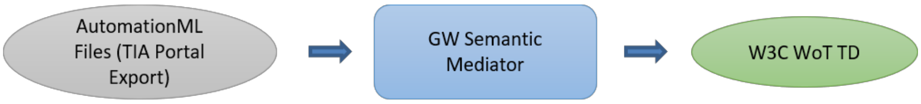
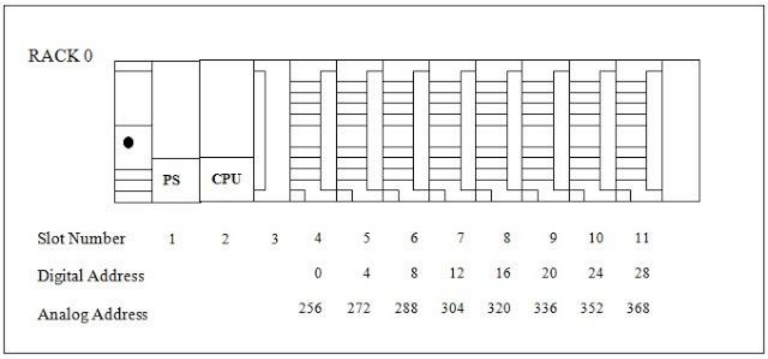


Figure 19: GW Semantic Mediator: mapping existing field meta-data into wot TD

Figure 20 depicts how hardware and address space are organized in Siemens PLCs. We see that modules (CPU, power supply, input output units etc.) are mounted on a rail, called the *rack*. Racks are numerated (Figure 20 shows the rack 0). The number of racks depends on complexity of an automation system. Each module in the rack has a number too. Mounting order of the modules, starting from the left-hand side, is: 1. Power Supply (PS) module; 2. CPU; 3. Interface Module (IM). Different Signal Modules (SMs) are placed from position 4, and onwards. We see that a *slot* number is assigned to each mounted module. In order to read or write values to a PLC we need to know the rack and slot number of its position on a rail, as well as an IP address of it.

[38] Mediator has further roles too, i.e., to create Device Nodes, installs them in Semantic Edge Platform, saves created TDs in TD Directories etc.



Slot Number	1	2	3	4	5	6	7	8	9	10	11
Digital Address				0	4	8	12	16	20	24	28
Analog Address				256	272	288	304	320	336	352	368

Module Starting Addresses of the Signal Modules on Rack 0

Figure 20: ADDRESSING OF MODULES in Siemens PLC[39]

Figure 21 shows excerpt from an AutomationML File that we exported from TIA Portal for our SEMIoTICS project. It is worth noting that the relevant information for PLC2 are: Rack 0; Slot 2; IP address: 192.168.0.2 (see Figure 18, and values marked in yellow in Figure 21).

```
  <InternalElement ID="9b97a633-fb65-46d6-8d35-64f19088699b" Name="Rack_0">
<Attribute Name="TypeName" AttributeDataType="xs:string">
  <Value>Rail</Value>
</Attribute>
<Attribute Name="PositionNumber" AttributeDataType="xs:int">
  <Value>0</Value>
</Attribute>
<Attribute Name="BuiltIn" AttributeDataType="xs:boolean">
  <Value>false</Value>
</Attribute>
<Attribute Name="TypeIdentifier" AttributeDataType="xs:string">
  <Value>System:Rack.ET200S</Value>
</Attribute>
<InternalElement ID="25fcb9a7-3090-4fff-aba5-f8abfdcd5b78" Name="PLC_2">
  <Attribute Name="TypeName" AttributeDataType="xs:string">
    <Value>IM 151-8 PN/DP CPU</Value>
  </Attribute>
  <Attribute Name="DeviceItemType" AttributeDataType="xs:string">
    <Value>CPU</Value>
  </Attribute>
  <Attribute Name="PositionNumber" AttributeDataType="xs:int">
    <Value>2</Value>
  </Attribute>
  <Attribute Name="BuiltIn" AttributeDataType="xs:boolean">
    <Value>false</Value>
```

[39] http://plcprogrammablelogiccontroller.blogspot.com/2011/09/addressing-of-modules-in-siemens-plc.html?_sm_au_=iVVJfk2TD1P05SBBt2tQvK032Hv7C

```
            </Attribute>
            <Attribute Name="TypeIdentifier" AttributeDataType="xs:string">
              <Value>OrderNumber:6ES7 151-8AB01-0AB0</Value>
            </Attribute>
            <Attribute Name="FirmwareVersion" AttributeDataType="xs:string">
              <Value>V3.2</Value>
            </Attribute>
            <InternalElement ID="78c6d195-14aa-436a-9da1-149539a1f24c" Name="Default tag table">
              <SupportedRoleClass RefRoleClassPath="AutomationProjectConfigurationRoleClassLib/Ta
gTable" />
            </InternalElement>
            <InternalElement ID="9f90c0e8-7ea6-4ba4-aac7-
db4471ae3326" Name="PROFINET interface_1">
              <Attribute Name="Label" AttributeDataType="xs:string">
                <Value>X1</Value>
              </Attribute>
              <Attribute Name="PositionNumber" AttributeDataType="xs:int">
                <Value>1</Value>
              </Attribute>
              <Attribute Name="BuiltIn" AttributeDataType="xs:boolean">
                <Value>true</Value>
              </Attribute>
              <InternalElement ID="f1032f4a-2811-4cf4-88c9-34af59e6d09c" Name="IE1">
                <Attribute Name="Type" AttributeDataType="xs:string">
                  <Value>Ethernet</Value>
                </Attribute>
                <Attribute Name="NetworkAddress" AttributeDataType="xs:string">
                  <Value>192.168.0.2</Value>
```

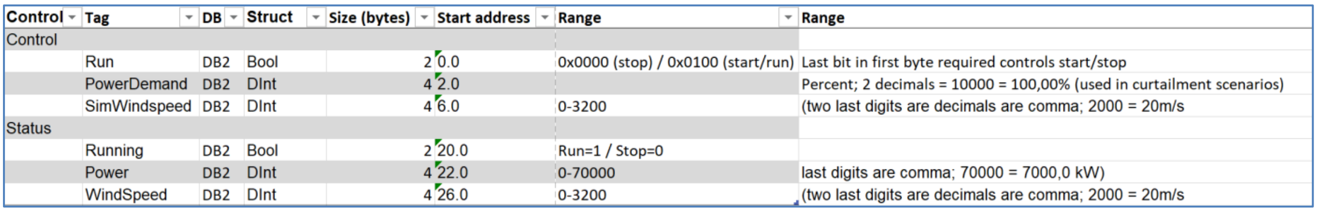
Figure 21: excerpt from an AutomationML File obtained from TIA Portal

Some of these values are predefined from manufacture of a device. For default values of racks and slots that are bound to certain types of Siemens PLCs, see Appendix 8.1.

Apart from the values for racks and slots, let us now consider the actual variables from Figure 18 too. We can extract essential meta-data (tags) for reading and changing variables in the wind turbine, see Figure 22. For example, if we want to start or stop the turbine, then we just need to write a Boolean value to the variable *Run*. This value should be written with the address offset 0.0. The next variable in the control program is *PowerDemand*, see Figure 22. It is a double integer (DInt), which takes 4 bytes. For the complete list of data types and their sizes, see Appendix 8.2. If *PowerDemand* needs to be changed, it can be written from the address offset 2.0 (since the first 2 bytes have been reserved by the variable *Run*[40]). Similarly, if we want to read the status of the turbine, then we need to read *Status* variables, see Figure 22. Again, it is important to know from which location in the memory stack we start reading data (i.e., the address offset) and how many bytes we have to read for each particular data type. In order to check whether the turbine is running, we can read the variable *Running*, i.e., to read a Boolean value starting from 20th byte. Figure 23 provides content of

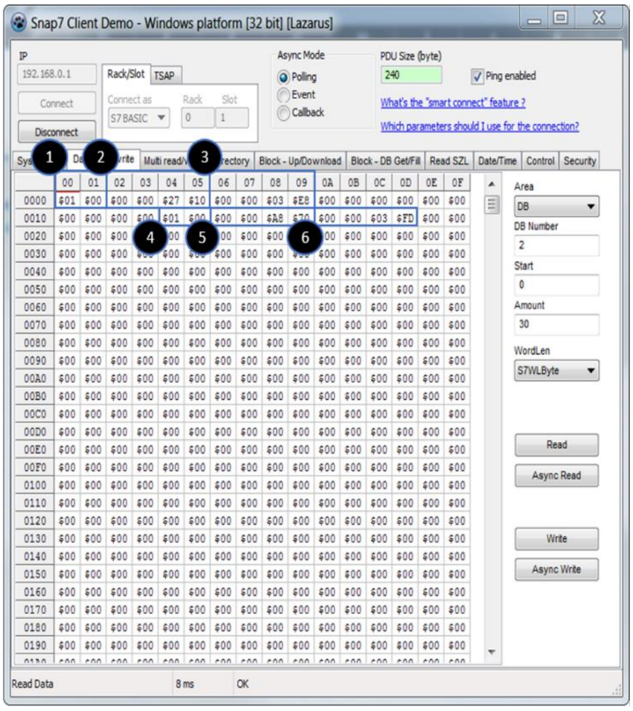
[40] Note that a Boolean variable in Siemens Step 7 data type system takes only one bit. But in PLC programs usually more space is reserved.

the payload of a complete data block from the wind turbine controller[41]. One can see what the current value of each variable is. On the right-hand side of the figure you see also the interpretation of data. For example, the variable Run has value “01 00”, which means “RUN” the turbine (as opposed to value “00 00” – “STOP” the turbine). As already mentioned, the variable occupies 2 bytes and the address offset is 0.



Control	Tag	DB	Struct	Size (bytes)	Start address	Range	Range
Control							
	Run	DB2	Bool	2	0.0	0x0000 (stop) / 0x0100 (start/run)	Last bit in first byte required controls start/stop
	PowerDemand	DB2	DInt	4	2.0		Percent; 2 decimals = 10000 = 100,00% (used in curtailment scenarios)
	SimWindspeed	DB2	DInt	4	6.0	0-3200	(two last digits are decimals are comma; 2000 = 20m/s
Status							
	Running	DB2	Bool	2	20.0	Run=1 / Stop=0	
	Power	DB2	DInt	4	22.0	0-70000	last digits are comma; 70000 = 7000,0 kW)
	WindSpeed	DB2	DInt	4	26.0	0-3200	(two last digits are decimals are comma; 2000 = 20m/s

Figure 22: Specification of data tag list available TIA Portal



```
      00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
0000 $01 $00 $00 $00 $27 $10 $00 $00 $03 $E8 $00 $00 $00 $00 $00 $00
0010 $00 $00 $00 $00 $01 $00 $00 $00 $A8 $70 $00 $00 $03 $FD $00 $00
0020 $00 $00 $00          00      00 $00 $00       00 $00 $00 $00 $00 $00
0030 $00 $00 $00     $00     $00 $00 $00     $00 $00 $00 $00 $00 $00
0040 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0050 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0060 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0070 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0080 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0090 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00A0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00B0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00C0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00D0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00E0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
00F0 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0100 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0110 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0120 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0130 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0140 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0150 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0160 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0170 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0180 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
0190 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
```

1 Run			
	Value	01 00	RUN
		00 00	STOP
	Bytes	2	
	Start address	0	
2 Power Demand			
	Value	00 00 00 00	0
		00 00 27 10	1
	Bytes	4	
	Start address	2.0	
3 Sim Wind Speed			
	Value	00 00 00 00	0 m/s
		00 00 0C 80	32 m/s
	Bytes	4	
	Start address	6.0	
4 Running			
	Value	01 00	RUN
		00 00	STOP
	Bytes	2	
	Start address	20.0	
5 Power			
	Value	00 00 00 00	0kW
		00 01 11 70	7000kW
	Bytes	4	
	Start address	22.0	
6 Wind Speed			
	Value	00 00 00 00	0 m/s
		00 00 0C 80	32 m/s
	Bytes	4	
	Start address	26.0	

Figure 23: content of the Payload of Wind Turbine Controller’s data block

Meta-data, related to the variables, can also be obtained from TIA Portal Openness[42], see Figure 24. For example, there we see that the structure *Status* starts from 20th byte, i.e., 160th bit (see the offset value marked in yellow, in Figure 24). Further, we see what variables are available (e.g., *Running* and *Power*), as well as their data types etc.

```
<Member Name="Status" Datatype="Struct" Remanence="Retain">
        <AttributeList>
         <IntegerAttribute Name="Offset" Informative="true" SystemDefined="true">160</IntegerAttribute>
         <BooleanAttribute Name="ExternalAccessible" SystemDefined="true">true</BooleanAttribute>
```

[41] Provided by an open source project: http://snap7.sourceforge.net/

[42] https://support.industry.siemens.com/cs/document/108716692/tia-portal-openness%3A-introduction-and-demo-application?dti=0&lc=en-RU

```
      <BooleanAttribute Name="ExternalVisible" SystemDefined="true">true</BooleanAttribute>
      <BooleanAttribute Name="ExternalWritable" SystemDefined="true">true</BooleanAttribute>
      <BooleanAttribute Name="UserVisible" Informative="true" SystemDefined="true">true</BooleanAttribute>
      <BooleanAttribute Name="UserReadOnly" Informative="true" SystemDefined="true">false</BooleanAttribute>
      <BooleanAttribute Name="UserDeletable" Informative="true" SystemDefined="true">true</BooleanAttribute>
      <BooleanAttribute Name="SetPoint" SystemDefined="true">false</BooleanAttribute>
    </AttributeList>
    <Member Name="Running" Datatype="Bool">
      <AttributeList>
        <IntegerAttribute Name="Offset" Informative="true" SystemDefined="true">0</IntegerAttribute>
        <BooleanAttribute Name="ExternalAccessible" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="ExternalVisible" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="ExternalWritable" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="UserVisible" Informative="true" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="UserReadOnly" Informative="true" SystemDefined="true">false</BooleanAttribute>
        <BooleanAttribute Name="UserDeletable" Informative="true" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="SetPoint" SystemDefined="true">false</BooleanAttribute>
      </AttributeList>
    </Member>
    <Member Name="Power" Datatype="DInt">
      <AttributeList>
        <IntegerAttribute Name="Offset" Informative="true" SystemDefined="true">16</IntegerAttribute>
        <BooleanAttribute Name="ExternalAccessible" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="ExternalVisible" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="ExternalWritable" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="UserVisible" Informative="true" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="UserReadOnly" Informative="true" SystemDefined="true">false</BooleanAttribute>
        <BooleanAttribute Name="UserDeletable" Informative="true" SystemDefined="true">true</BooleanAttribute>
        <BooleanAttribute Name="SetPoint" SystemDefined="true">false</BooleanAttribute>
      </AttributeList>
```

Figure 24: excerpt File obtained from TIA Portal Openness

So far, we have described what and how the mediator should use the existing meta-data during the integration process of brownfield device to the IoT domain. In order to automate this process, the mediator needs to access the meta-data in machine processable format. These are typically XML files that can be obtained from engineering stations such as TIA Portal. This is manual work. Once completed, the mediator can use this metadata to automatically extract required information and use it for creation of a Thing Description and Device Nodes.

Figure 25 shows an excerpt of Thing Description, which is generated by the mediator for the wind turbine (in use case 1).

```
"properties": {
        "WindSpeedValue": {
            "description" : "Shows Wind Speed",
            "type": "double",
            "forms": [{
                "href": "s7comm://192.168.0.2/0/0/5000/DB3,DINT26"
            }]
        },
        "changeWindSpeed": {
            "description" : "changes Wind Speed",
            "type": "double",
            "writeOnly" : true,
            "forms": [{
                "href": "s7comm://192.168.0.2/0/0/5000/DB3,DINT6/800"
            }]
        },
        "RunWindTurbine": {
            "description" : "turn on",
            "type": "boolean",
            "writeOnly" : true,
            "forms": [{
                "href": "s7comm://192.168.0.2/0/0/5000/DB3,X0.0/true"
            }]
        }
    }
```

Figure 25: excerpt Thing Description for wind turbine

As already said, GW Semantic Mediator also generates Device Nodes as a convenient means to interact with either greenfield or brownfield device. These nodes can be installed in Semantic Edge Platform and used for creating applications. Figure 26 shows a code excerpt that can be used for reading the WindSpeed variable from the wind turbine. Note that a Device Node for the WindSpeed can be automatically generated from the Thing Description, which is shown Figure 25 (IP address, rack, slot etc.). Second, we also need to address correctly the variable (see 'DB3, DINT26' in Figure 26). Since WindSpeed is a double integer, which is to be read from data block 3, from offset 26, we have to encode this information as 'DB3, DINT26'. The full list of mapping for our implementation, which is based on open source project node-red-contrib-s7[43], can be found in Appendix 8.3.

```
var nodes7 = require('nodes7');
var conn = new nodes7;
```

[43] https://github.com/netsmarttech/node-red-contrib-s7

```
var doneReading = false;
var doneWriting = false;

var variables = { WindSpeed: 'DB3, DINT26' // WindSpeed value as double integer
(32-bit) at byte 26 of DB 03
};

conn.initiateConnection({port: 24, host: '192.168.0.2', rack: 0, slot: 2},
connected);

function connected(err) {
     if (typeof(err) !== "undefined") {
           // We have an error.  Maybe the PLC is not reachable.
           console.log(err);
           process.exit();
     }
     conn.setTranslationCB(function(tag) {return variables[tag];});  // This
sets the "translation" to allow us to work with object names
     conn.addItems(['WindSpeed']);
     conn.readAllItems(valuesReady);
}

function valuesReady(anythingBad, values) {
     if (anythingBad) { console.log("SOMETHING WENT WRONG READING VALUES!!!!");
}
     console.log(values);
     doneReading = true;
     if (doneWriting) { process.exit(); }
}

function valuesWritten(anythingBad) {
     if (anythingBad) { console.log("SOMETHING WENT WRONG WRITING VALUES!!!!");
}
     console.log("Done writing.");
     doneWriting = true;
     if (doneReading) { process.exit(); }
}
```

Figure 26: excerpt Code to read WindSpeed value in a Device Node

Finally, the mediator can generate Device Nodes for each interaction possibilities with a brownfield device. For the example TD shown in Figure 25, the following Device Nodes are generated (see the orange nodes in Figure 27).

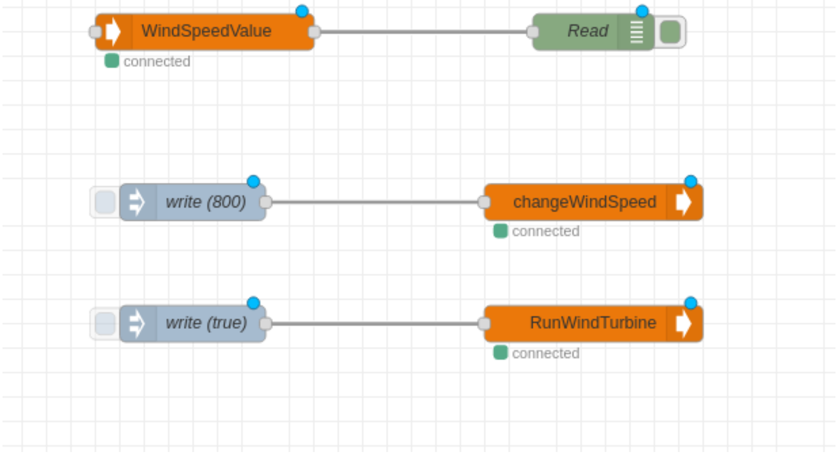


Figure 27: Device Nodes, automatically generated from Thing Description

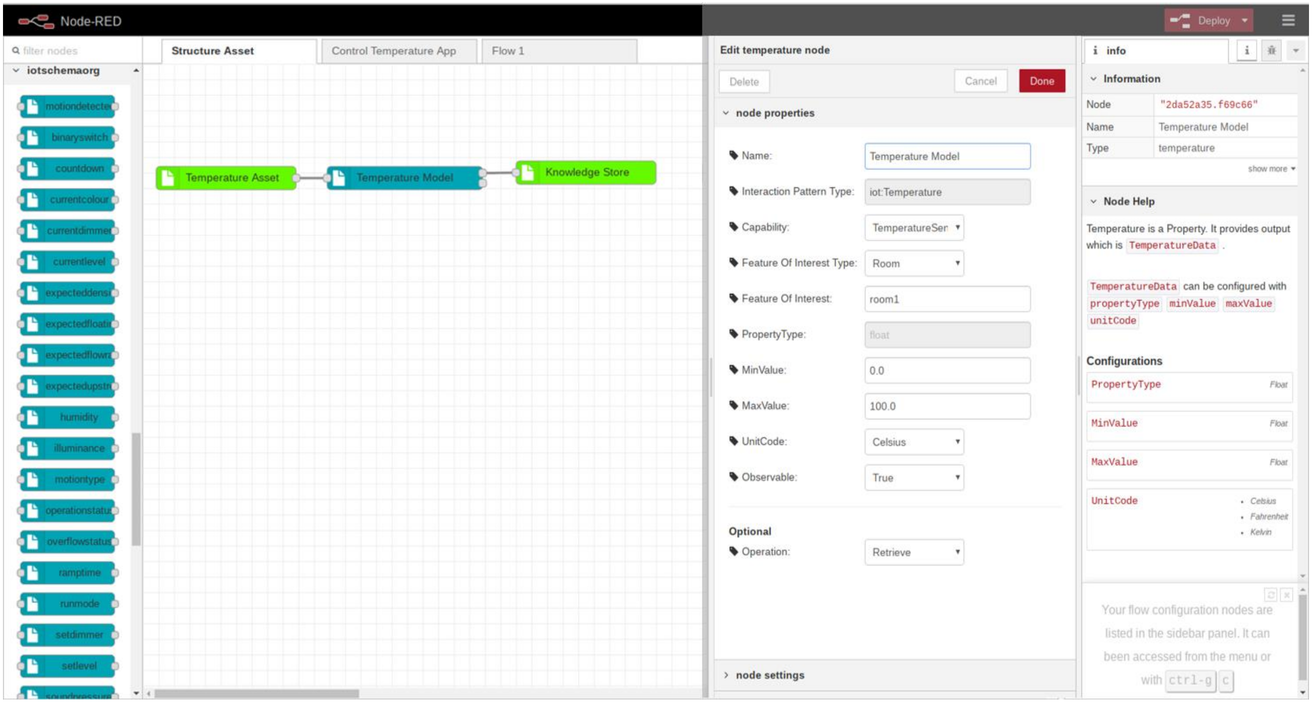
4.1.3 Semantic Configuration Nodes

Thing Description that is generated by the mediator is not semantically annotated with terms from a domain model, e.g., iotschema.org. In order to enable a user to enrich the Thing Description we provide Semantic Nodes. These nodes are automatically generated from iotschema.org and are optionally used by the user of SEMIoTICS Gateway.

Figure 28 shows a Semantic Node for a temperature sensor (see the petrol-colored node). On the right-hand side of the figure one can see a list of semantic properties defined by iotschema.org for a temperature sensor (TemperatureSensing Capabilitiy[44]).

Semantic Nodes are available in Semantic Edge Platform for the purpose of creating semantically annotated Thing Description for both a brownfield and greenfield device. They enable a user (without expertise in semantic technologies) to configure semantic properties of a device, i.e., to choose offered values from the semantic model via a graphic component. Once a user has semantically configured a brownfield device over the Semantic Nodes, the mediator will automatically generate an enriched Thing Description. From that moment on, it will be possible to discover a device over its TD and Local Thing Directory, based also on the enriched semantic meta-data.

[44] http://iotschema.org/TemperatureSensing



Edit temperature node

node properties

Name: Temperature Model
Interaction Pattern Type: iot:Temperature
Capability: TemperatureSen
Feature Of Interest Type: Room
Feature Of Interest: room1
PropertyType: float
MinValue: 0.0
MaxValue: 100.0
UnitCode: Celsius
Observable: True

Optional
Operation: Retrieve

node settings

Information

Node	"2da52a35.f69c66"
Name	Temperature Model
Type	temperature

show more

Node Help

Temperature is a Property. It provides output which is TemperatureData .

TemperatureData can be configured with propertyType minValue maxValue unitCode

Configurations

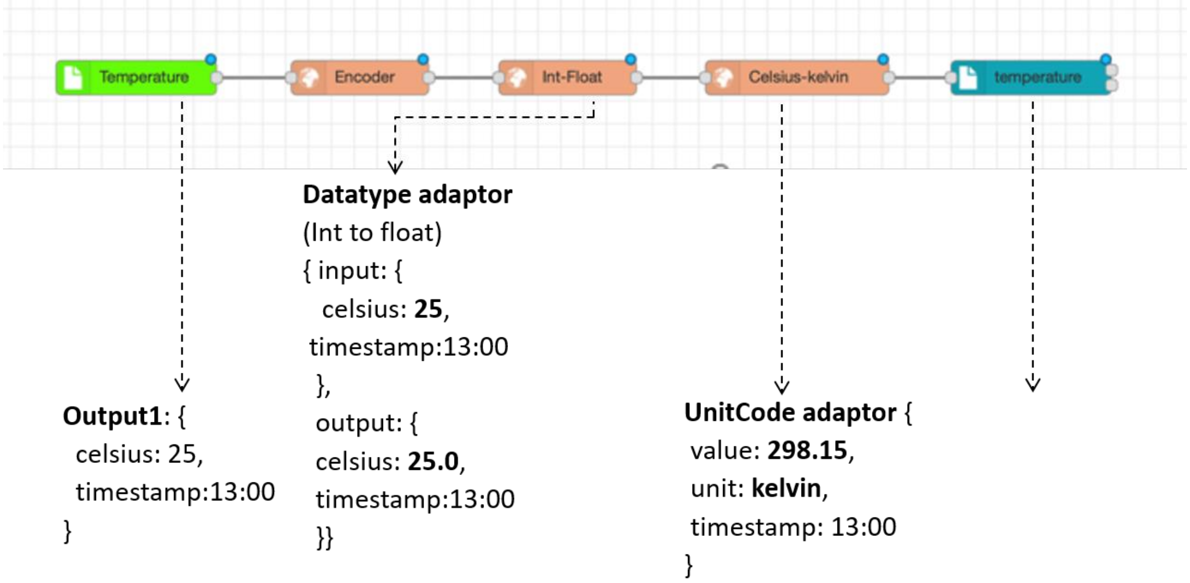
PropertyType	Float
MinValue	Float
MaxValue	Float
UnitCode	Celsius Fahrenheit Kelvin

Your flow configuration nodes are listed in the sidebar panel. It can been accessed from the menu or with ctrl-g c

Figure 28: Semantic Node for Temperature Sensor from iotschema.org

4.1.4 Adaptation nodes

When Semantic Nodes are used (see Section 4.1.3) several aspects should be taken into consideration for the integration of existing devices with iotschema.org semantics. The input/output data schema of a device should be adopted to be compliant with iotschema.org specification. This means that value type, encoding format, unit of measurement of data of an existing device should be adopted as prescribed by a corresponding iotschema semantic model. For example, if a temperature thing gives an integer value as output, but iotschema Semantic Node for temperature prescribes that the temperature data should be float, then the output of the temperature sensor should be converted from integer to float. The conversion may be motivated with the need to integrate the device's semantics with iotschema.org temperature definition. But sometimes we want to make this conversion just for a specific purpose of an application. No matter what the motivation for the conversion is, we offer Adaptation Nodes for such purposes. In particular, we offer a mediation API to adopt a data format from diverse IoT ecosystem or diverse serialization format to iotschema.org data format. The mediation API is also offered as a set of nodes in Semantic Edge Platform. For example, we provide nodes to convert a data from integer to float or from float to double. We provide nodes to convert data in string format to JSON or vice versa. We provide nodes to convert data from one unit of measurement to another one. A user should use one or more adaptation nodes to integrate an existing device's data with iotschema.org specification as shown in Figure 29. For this purpose, a user may use a Device Node, which is generated by the mediator (see the light green node for Temperature in Figure 29), and creates a flow by wiring it with one or more Adaptation Nodes. She may then connect the Adaptation Nodes with an iotschema node (Semantic Node). She then configures the iotschema node according to the specification of the device. The Semantic Node gives two outputs. The first output provides RDF description, which is configured according to the user's specification. The second output is the run-time value given as output by the device (if that device gives output). The semantic description of a device thus created can be stored in Thing Description Directory for discovery. As described in previous sections the output of a configured iotschema nodes can be used for multiple purposes such as for the generation and annotation of W3C WoT TD, for creating semantically interoperable WoT applications or for validating data being exchanged between devices. In this way, using this approach, semantic integration of heterogeneous devices can be easily achieved even by non-semantic experts, e.g. device vendors, IoT platform providers, IoT application developers, Web developers, and so on.



Datatype adaptor
(Int to float)
{ input: {
celsius: **25**,
timestamp:13:00
},
output: {
celsius: **25.0**,
timestamp:13:00
}}

Output1: {
celsius: 25,
timestamp:13:00
}

UnitCode adaptor {
value: **298.15**,
unit: **kelvin**,
timestamp: 13:00
}

Figure 29: Data-level mediation: iotschema.org ADAPTERS nodes

4.2 Semantic API & Protocol Binding

Semantic API & Protocol Binding is a component responsible for binding different protocol and exposing common semantic API located at the Generic IoT Gateway layer. This functionality is needed in order to integrate brownfield devices into a common IoT access layer. Technology-wise, the functionality is based on W3C Web of Things (WoT) API[45].

In the following we give API that is implemented and tested in SEMIoTICS IoT Gateway (GW). We will start by presenting the implementation for greenfield devices, followed by the implementation for brownfield devices.

SEMIoTICS greenfield device (e.g., Raspberry Pi with attached an IP camera) implements the following interface for starting the camera in a WoT servient, see Table 7. Note that a method for starting a camera is semantically annotated with iotschema.org mark-up for a camera[46].

Table 7: WoT Servient – greenfield device Interface

```
let thing = WoT.produce({
title: "SEMIoTICS Thing",
description: "Camera",
"@context": ["https://www.w3.org/2019/wot/td/v1", {"iot":
"http://iotschema.org/" }],
   "@type": "iot:StartRecording",
   "iot:capability": "iot:Camera",
   actions: {
     startCamera: {
        description: "Start recording the video.)"
     }
```

[45] https://www.w3.org/TR/wot-scripting-api/
[46] http://iotschema.org/Camera

```
    }
});

thing.setActionHandler(
    "startCamera",
    (params, options) => {
       // Code that implements an Action, e.g., "startCamera".
       // Removed for the sake of simplicity.
    });
```

The greenfield device does not require the protocol binding. A client, which needs to access a newly plugged (greenfield) device can access Thing Description (TD) of the plugged device. In this way the client can discover device's functionality. This can be achieved by providing the IP address of the device in the method fetch, as shown in Table 8.

Table 8: Fetching a Thing Description

```
WoTHelpers.fetch("http://semiotics.things.org:8080/ipcamera").then( async (td)
=> {
     let thing = await WoT.consume(td);
}).catch( (err) => { console.error("Fetch error:", err); });
```

By examining the fetched TD a client, for example, finds an action called startCamera. The client can use then the API to invoke the action, see Table 9.

Table 9: Interacting with a Thing (a camera)

```
// start the camera
await thing.invokeAction("startCamera");
```

Brownfield devices can be exposed over W3C WoT API only if there is a protocol binding available for them. Brownfield devices in use case 1 communicate over S7comm protocol. Since the binding for this protocol does not exist in the open source implementation of W3C WoT, we have implemented it. Our implementation, as already said, extends the project node-red-contrib-s7[47]. It enables a WoT servient to handle read and write operations on resources from a Thing Description, e.g., to read the variable *WindSpeed* or to turn off the wind turbine by setting the variable *Run* to false, see Section 4.1.2.

Our implementation of protocol binding assumes the following URL format for S7 resources in Thing Description, see Table 10.

Table 10: URL format for S7comm protocol

```
  s7comm://<host> [:<port>]/<rack>/<slot>/<timeout>/<variable>[/<writevalue>]
```

Apart from the IP address (host and port), there must be specified information about *rack* and *slot*, too. For an explanation about their meaning, see Section 4.1.2. Further, the *timeout* corresponds to the TCP timeout. Notation for the *variable* follows a scheme provided in Appendix 8.3. Finally, variables that should be set with a new value, use the field *writevalue*. Figure 30 shows our method for reading a resource from a brownfield device over a W3C WoT servient.

```
   public readResource(form: S7Form): Promise<Content> {
```

[47] https://github.com/netsmarttech/node-red-contrib-s7

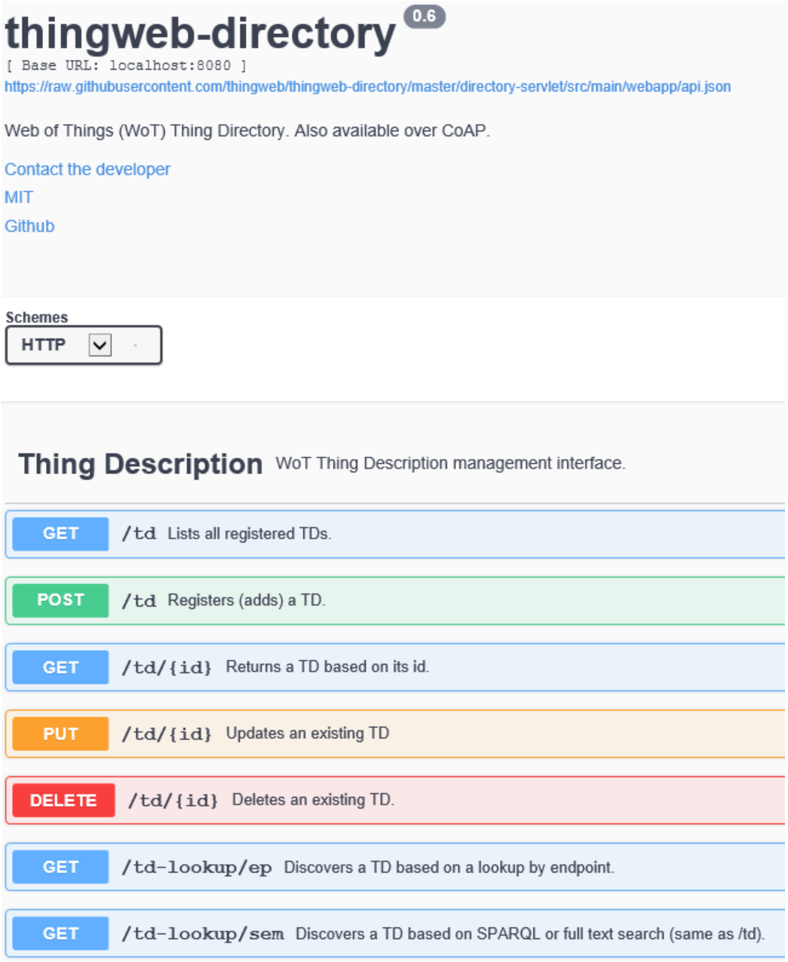
```
        let target = parseS7Uri(form['href']);
        return new Promise<Content>((resolve, reject) => {
            this.client.addItems([target.variable]);
            if (!this.client.connectCBIssued) {
                this.client.initiateConnection({ port: target.port, host: target
.address,

rack: target.rack, slot: target.slot, timeout: target.timeout }, connected);
            }
            else {
                console.log("Already connected");
                this.client.readAllItems(valuesReady);
                this.client.removeItems([target.variable]);
            }
            let myclient = this.client;
            function connected(err: any) {
                if (typeof (err) !== "undefined") {
                    console.log(err);
                    return;
                }
                myclient.readAllItems(valuesReady);
            }
            function valuesReady(anythingBad: any, values: object) {
                if (anythingBad) { console.log("SOMETHING WENT WRONG READING VAL
UES!!!!"); }

                let myvalues = JSON.stringify(values);
                resolve({ type: "text/plain", body: Buffer.from(myvalues) });
                myclient.removeItems([target.variable]);
            }
        });
    }
```

Figure 30: Protocol Binding – Reading a Resource

4.3 Local Thing Directory

The purpose of Local Thing Directory is to store semantic description of Things locally in the SEMIoTICS IoT Gateway. During the device registration process, the gateway (mediator) stores a TD in the Local Thing Directory. For the implementation of this component we use the open source implementation from W3C[48]. Figure 31 shows the API available from our local deployment of Thing Directory in the gateway. There we can see essential CRUD operations on TDs (create/register, read, update, and delete). Moreover, there is a possibility to look up TDs via semantic search.

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



thingweb-directory 0.6

[Base URL: localhost:8080]
https://raw.githubusercontent.com/thingweb/thingweb-directory/master/directory-servlet/src/main/webapp/api.json

Web of Things (WoT) Thing Directory. Also available over CoAP.

Contact the developer
MIT
Github

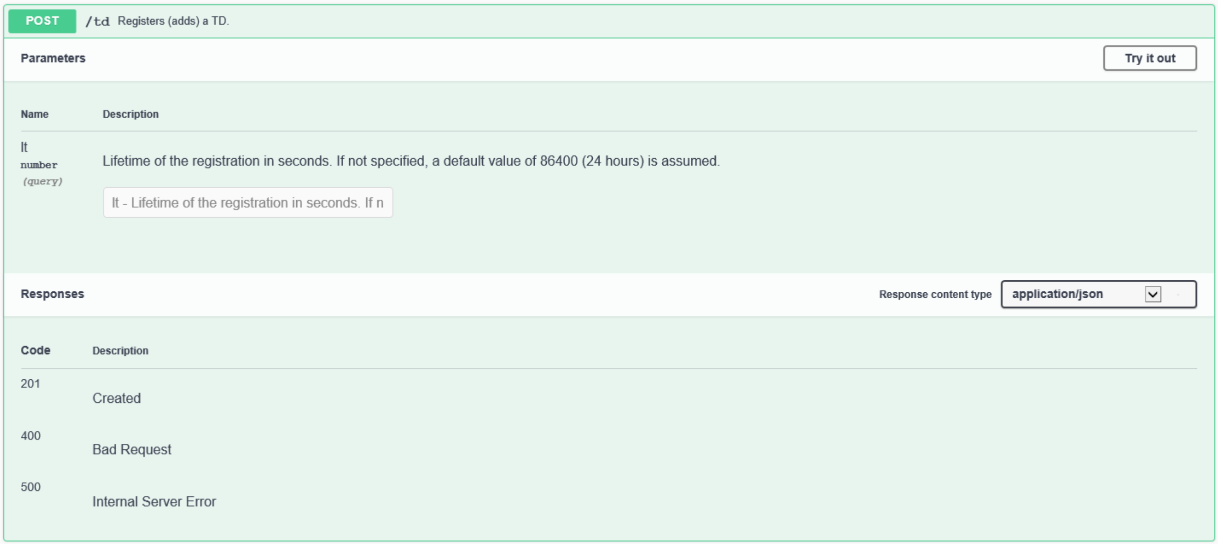
Schemes
HTTP

Thing Description WoT Thing Description management interface.

GET /td Lists all registered TDs.
POST /td Registers (adds) a TD.
GET /td/{id} Returns a TD based on its id.
PUT /td/{id} Updates an existing TD
DELETE /td/{id} Deletes an existing TD.
GET /td-lookup/ep Discovers a TD based on a lookup by endpoint.
GET /td-lookup/sem Discovers a TD based on SPARQL or full text search (same as /td).

Figure 31: OVERVIEW OF THING DIRECTORY API

Let us show part of this API in more detail. Figure 32 depicts the API related to the Thing Description registration. After fetching a TD, the gateway uses this method to store a TD in Local Thing Directory.



POST /td Registers (adds) a TD.

Parameters

Name	Description
lt number *(query)*	Lifetime of the registration in seconds. If not specified, a default value of 86400 (24 hours) is assumed. lt - Lifetime of the registration in seconds. If n

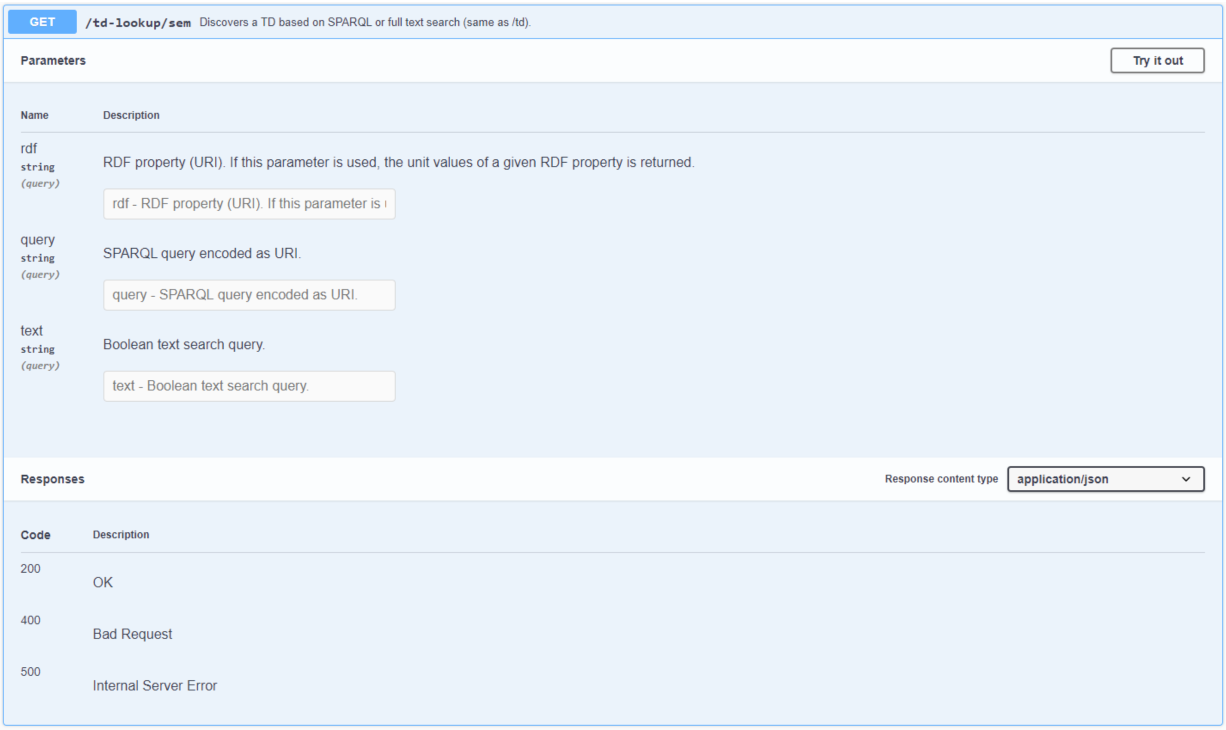
Responses

Response content type application/json

Code	Description
201	Created
400	Bad Request
500	Internal Server Error

Figure 32: API FOR REGISTRATION OF THING DESCRIPTION

An existing TD can be discovered from Local Thing Directory via a SPARQL query, see Figure 33.



GET `/td-lookup/sem` Discovers a TD based on SPARQL or full text search (same as /td).

Parameters

Name	Description
rdf string *(query)*	RDF property (URI). If this parameter is used, the unit values of a given RDF property is returned. rdf - RDF property (URI). If this parameter is u
query string *(query)*	SPARQL query encoded as URI. query - SPARQL query encoded as URI.
text string *(query)*	Boolean text search query. text - Boolean text search query.

Responses

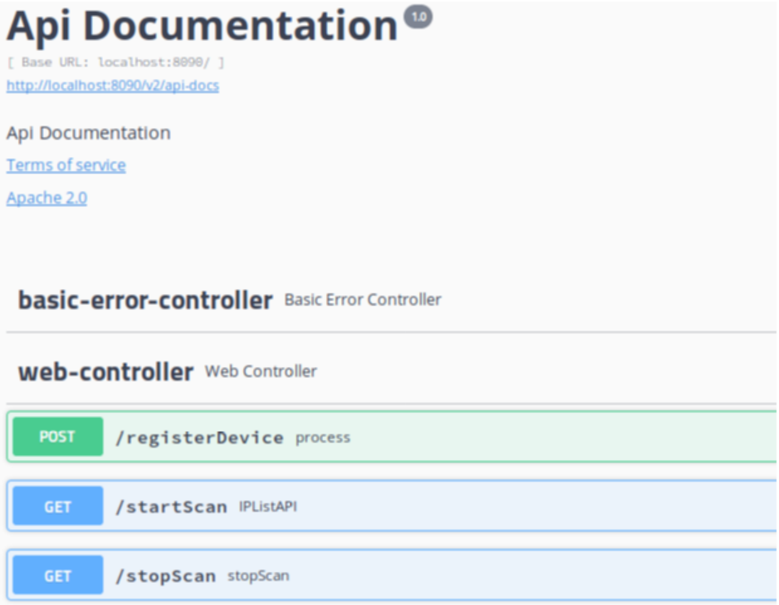
Response content type: application/json

Code	Description
200	OK
400	Bad Request
500	Internal Server Error

Figure 33: API FOR discovery OF THING DESCRIPTION

4.4 Semantic Edge Platform

Semantic Edge Platform (SME) provides a convenient interface for different components and functionalities of IoT Gateway, which are accessible over API but not necessarily have a user interface. Thus, for example SME enables a user to configure SEMIoTICS IoT Gateway, choose a network interface, define an IP address range when scanning a network for new devices, and initiate the device bootstrapping process. Figure 34 shows API of IoT Gateway, which is accessible over Semantic Edge Platform.



Api Documentation 1.0

[Base URL: localhost:8090/]
http://localhost:8090/v2/api-docs

Api Documentation
Terms of service
Apache 2.0

basic-error-controller Basic Error Controller

web-controller Web Controller

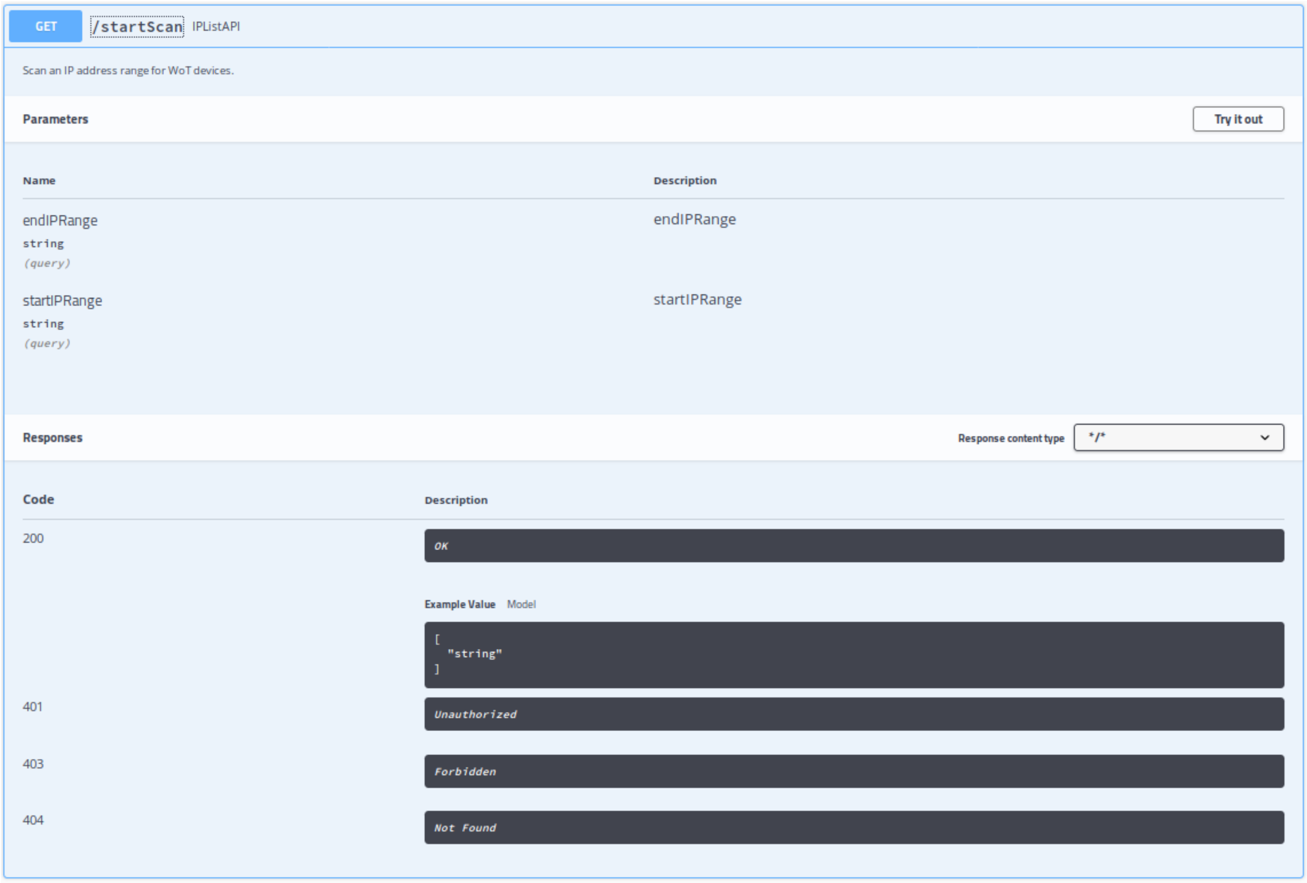
POST /registerDevice process

GET /startScan IPListAPI

GET /stopScan stopScan

Figure 34: API of IoT Gateway exposed over Semantic Edge Platform

Figure 35 presents the method that is used for scanning a network, where new devices are to be bootstrapped.



GET /startScan IPListAPI

Scan an IP address range for WoT devices.

Parameters

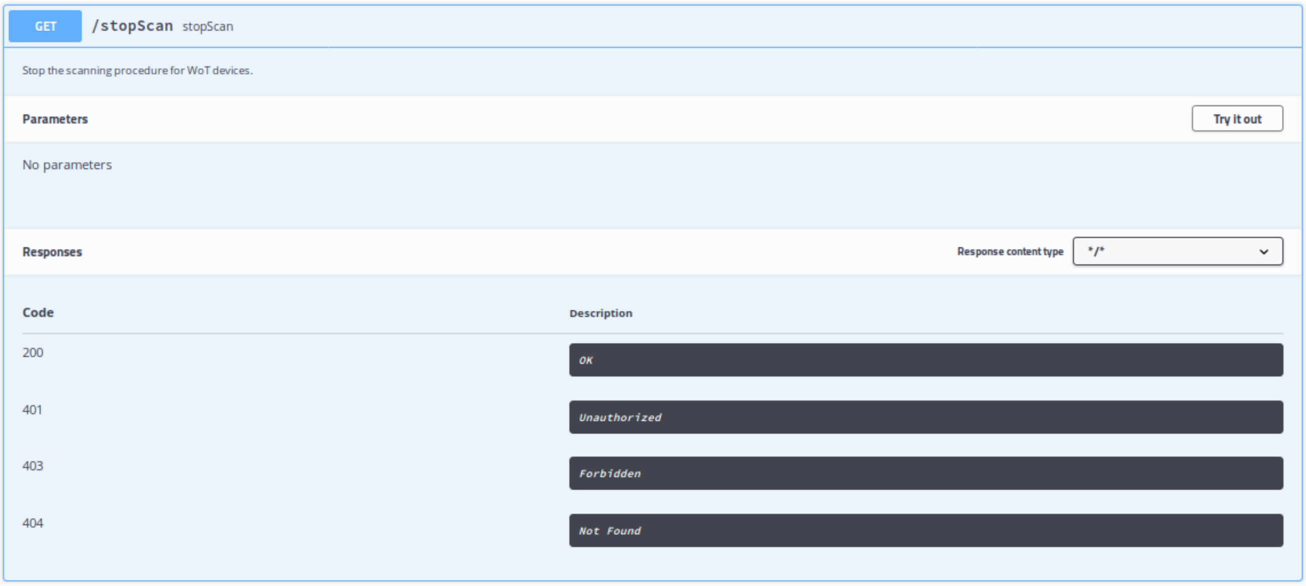
Name	Description
endIPRange string *(query)*	endIPRange
startIPRange string *(query)*	startIPRange

Responses Response content type */*

Code	Description
200	*OK* Example Value Model ["string"]
401	*Unauthorized*
403	*Forbidden*
404	*Not Found*

Figure 35: API of IoT Gateway to start Bootstrapping

Figure 37 shows the API, which enables a user to terminate the network scanning. For large networks this process may take a long time. Sometimes a user knows, which devices are supposed to appear during the scan. Thus, as soon as new devices appear, the process may be halted.



GET /stopScan stopScan

Stop the scanning procedure for WoT devices.

Parameters

Try it out

No parameters

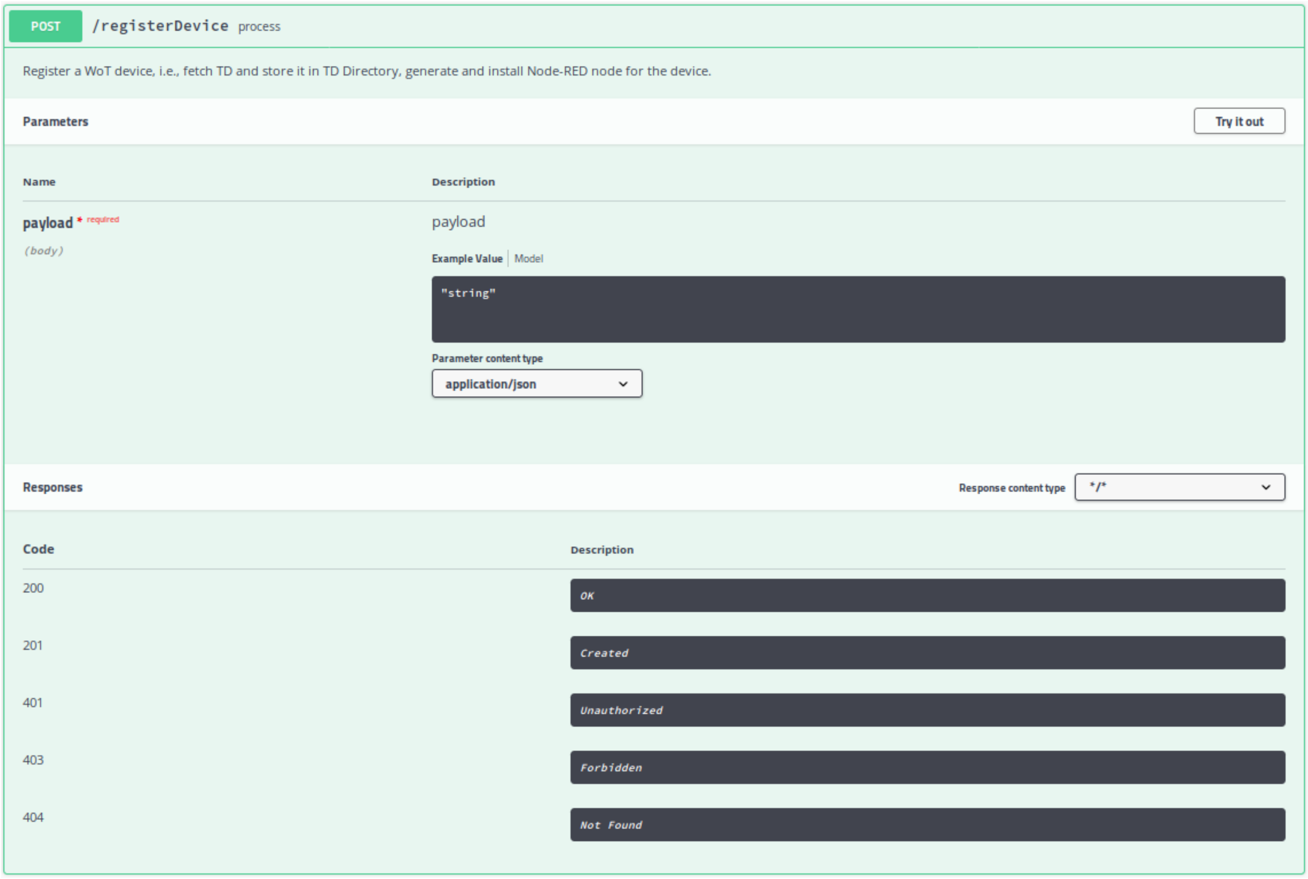
Responses

Response content type */*

Code	Description
200	OK
401	Unauthorized
403	Forbidden
404	Not Found

Figure 36: API of IoT Gateway to stop scanning the network

Figure 37 depicts the API for registering a new device. The method interacts with other gateway's component, i.e., it fetches device's Thing Description, stores it in TD Directory, invokes the GW Semantic Mediator to create a Device Node for the device, and finally installs the node in SME, thereby making it programmatically accessible in SME over WoT API.



POST /registerDevice process

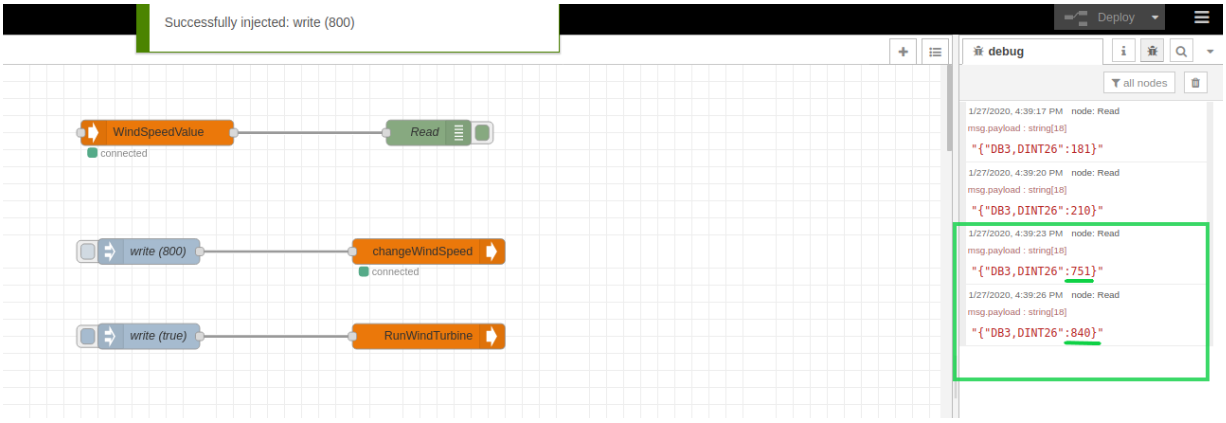
Register a WoT device, i.e., fetch TD and store it in TD Directory, generate and install Node-RED node for the device.

Name	Description
payload * required (body)	payload Example Value | Model "string" Parameter content type application/json

Code	Description
200	OK
201	Created
401	Unauthorized
403	Forbidden
404	Not Found

Figure 37: API of IoT Gateway to register a new WoT device

In order to test the implementation of the overall IoT Gateway we have performed few tests. Figure 38 shows three Device Nodes, which represent the functionality of data points from Figure 18 that is integrated in the gateway and available over W3C WoT API. If we (or an application) interact with these nodes, then brownfield device will change accordingly. For example, if one changes the wind speed, the actual value by the turbine will change as well (see Figure 38).



Successfully injected: write (800)

```
1/27/2020, 4:39:17 PM   node: Read
msg.payload : string[18]
 "{"DB3,DINT26":181}"
1/27/2020, 4:39:20 PM   node: Read
msg.payload : string[18]
 "{"DB3,DINT26":210}"
1/27/2020, 4:39:23 PM   node: Read
msg.payload : string[18]
 "{"DB3,DINT26":751}"
1/27/2020, 4:39:26 PM   node: Read
msg.payload : string[18]
 "{"DB3,DINT26":840}"
```

Figure 38: Testing the Change of WindSpeed value of Wind Turbine in Use Case 1

Figure 39 depicts a test application for both greenfield and brownfield device. Once the bootstrapping and integration process has been completed, we can start developing new applications. For example, we can process the audio signal from a newly plugged microphone. If a noise detection function discovers malfunctional behavior, then we can stop the wind turbine. Figure 39 shows such an application, realized in Semantic Edge Platform. This application can be represented as a recipe and instantiated and configured for different wind turbine. Once the IoT gateway provides a uniform and semantically described access to greenfield and brownfield devices, it becomes possible to rapidly develop new applications.

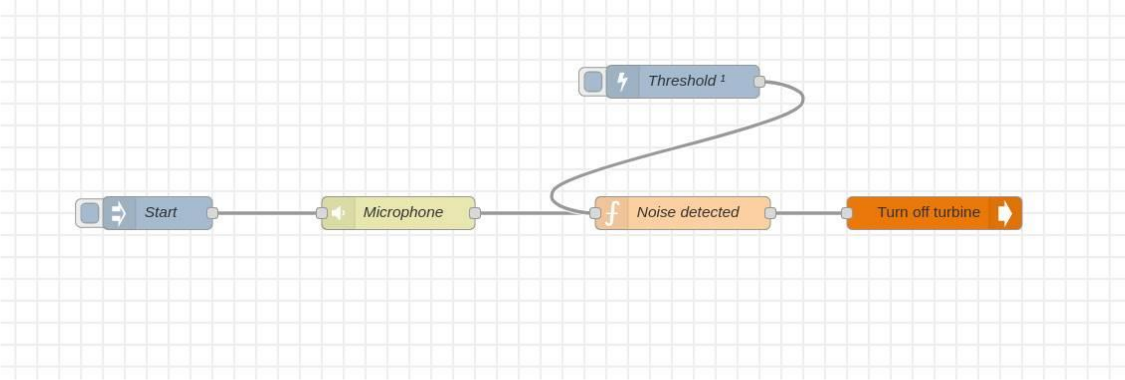


Figure 39: Test Application - safe shutdown of Wind Turbine

4.5 Implementation Details Related to Use Case 2

In the previous sections we have described the implementation of various components of SEMIoTICS Gateway by always referring to devices from use case 1. This section contains the description of a subset of sensors on board used in use case 2, i.e., the SARA Robotic Rollator (RR).

The descriptions of sensors are:

4.5.1 Ultrasonic range sensor

4.5.1.1 Properties

Name	Description	Input	Output
ObstacleSensors	Distance from the object closest to the left sensor	-	obstacleL
ObstacleSensors	Distance from the object closest to center sensor	-	obstacleC
ObstacleSensors	Distance from the object closest to the right sensor	-	obstacleR

4.5.1.2 Actions

None.

4.5.1.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
obstacleL	Number	0	3	Meters
obstacleC	Number	0	3	Meters
obstacleR	Number	0	3	Meters

4.5.2 Gyroscope

4.5.2.1 Properties

Name	Description	Input	Output
Angular speed		-	AngularSpeed
SetResolution		-	DegPerSecRange

4.5.2.2 Actions

Name	Description	Input	Output
SetResolution	Set the resolution property	DegPerSecRange	-

4.5.2.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
AngularSpeed		-2000	2000	
DegPerSecRange	One of those: 0) ±250 [deg x sec] 1) ±500 [deg x sec] 2) ±2000 [deg x sec]	0	2	Number

4.5.3 Accelerometer

4.5.3.1 Properties

Name	Description	Input	Output
Acceleration	Acceleration measured by the sensor	-	Acceleration
Resolution		-	AccelerationValueRange

4.5.3.2 Actions

Name	Description	Input	Output
SetResolution	Set the resolution of the sensor	AccelerationValueRange	-

4.5.3.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
Acceleration		-16	16	Number
AccelerationValueRange	One of those: 0) ±2g 1) ±4g 2) ±8g 3) ±16g	0	3	Number

4.5.4 Hub motor

4.5.4.1 Properties

Name	Description	Input	Output
GetPower	Power of the motor	-	GetP
SetPower	Value given to control the motorized wheels	SetP	-

4.5.4.2 Actions

Name	Description	Input	Output
Motors	Power of the motor in Watt to AnalogValue	SetP	-

4.5.4.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
AnalogValue	Analog pin values to control the motorized wheels	0	255	Number

4.5.5 Encoder

4.5.5.1 Properties

Name	Description	Input	Output
Encoder	Speed measured on the wheels	-	TickCount

4.5.5.2 Actions

None.

4.5.5.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
TickCount	Position of the wheel	0	360	Number

4.5.6 Force sensor

4.5.6.1 Properties

Name	Description	Input	Output
Force	Force measured by the load sensor on the handles	-	Force

4.5.6.2 Actions

None.

4.5.6.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
Force		-25	25	Kg$_F$

4.5.7 Compass

4.5.7.1 Properties

Name	Description	Input	Output
Magnetic flux density	Magnetic flux density measured by the sensor	-	Magnetic flux density
Resolution		-	MagnValueRange

4.5.7.2 Actions

Name	Description	Input	Output
SetResolution	Set the resolution of the sensor	MagnValueRange	-

4.5.7.3 Data types

Name	Definition	Min Value	Max Value	Unit Code
GausRange		-8.1	8.1	Gaus
MagnValueRange	One of those: 0) ±1.3 [gaus] 1) ±8.1 [gaus]	0	1	Number

All the sensors previously described communicate to the system SARA using the CoAP protocol; thus, we needed the IoT semantic model necessary to convert the sensors into a WoT node, as described in this chapter (protocol binding from CoAP to WoT/HTTP has been used). Following an example of this model, we managed to make data from the left ultrasonic range sensor of the RR (“obstacleL”) available over a W3C WoT servient:

```
{
    "@context ": "https://www.w3.org/2019/td/v1",
    "id": "urn:wotrrsara",
    "title": "WoT_RR_SARA",
    "description": "RR_SARA",
    "security": ["psk_sec"],
    "securityDefinitions": {
        "psk_sec": {"scheme": "psk"}
    },
    "properties": {
        "obstacleL": {
            "description": "obstacleL",
            "type": "number",
            "forms": [
                {
                    "href": "coap://sara.example.it:5683/obstacleL"
                }
            ]
        }
    }
```

}

5 VALIDATION

This section summarizes project's objectives, KPIs, and requirements (relevant for this task), and validates achievements against them.

5.1 Related Project Objectives and Key Performance Indicators (KPIs)

The overall deliverable constitutes the contribution towards fulfilling the project's objectives as shown in Table 11, and project's KPIs as presented in Table 12.

Table 11: Task's objectives

T3.3 Objectives	**D3.3 Chapters**
• **Objective 2:** Development of semantic interoperability mechanisms for smart objects, networks and IoT platforms	2
• **Objective 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 landscapes (smart sensing), and delivery of the respective open API.	3

Table 12: Tasks KPI Table

Objective (with short description)		KPI-ID	Description
2	Semantic interoperability	KPI-2.1	Delivery of semantic descriptions for all the 6 types of smart objects which are necessary for the usage scenarios
6	Development of a Reference Prototype	KPI-6.1	Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%

Deliverable D3.1 provides the concept of the field-level device bootstrapping and registration. The implementation of this concept has been presented in this deliverable, i.e., D3.9. In D3.3 deliverable we have already identified components of SEMIoTICS architecture that will be part of the implementation of the IoT Gateway (see Section 4), choose technology building blocks to implement these components (see Section 3.1), provided the sequence diagram that will be used in the implementation (see Section 4), and provided the use-case device set-up that will be used in the demonstration of the gateway (see Section 4). The four components (GW Semantic Mediator, Semantic API & Protocol Binding, Local Thing Directory, and Semantic Edge Platform) have been implemented in this deliverable.

5.2 Related Project Requirements

Let us revise requirements from Section 1.4 with respect to the provided work and work that has followed in this deliverable.

Requirement 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)), see Table 3: Specific requirements for use case 1

Essentially, this requirement is concerned with all layers of SEMIoTICS architecture, including the field layer and thus IoT Gateway too. In this sense, the gateway has the role to ensure connectivity between the heterogeneous IoT devices (at the field level) and itself. From that point on, the connectivity is ensured via the Networking- and Cloud/Backend layers. The gateway ensures connectivity between the heterogeneous IoT devices and itself. Further, the gateway communicates via SDN to Global Thing Directory and Cloud platforms. The end-to-end connectivity via SDN has been achieved and reported in deliverable D5.3.

Requirement R.FD.5: Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components SHOULD be supported.

This is a requirement for field devices. But the SEMIoTICS IIoT/IoT Gateway implements required Protocol Bindings (from the south-bound interface) in order to enable interaction between field devices and the gateway, see Section 4.1.2 and Section 4.2.

Requirement R.FD.6: Field devices MUST interoperate using a standard communication protocol like REST APIs, COAP, MQTT.

The technology blocks (Section 3.1) are based on RESTful paradigm and the current implementation support COAP and MQTT protocols.

Requirement R.FD.7: Field devices MUST use standardize interoperable message format (e.g. JSON, etc.).

The technology blocks (Section 3.1) are based on JSON and the current implementation supports this serialization format.

Requirement R.FD.8: Field devices MUST support secure bootstrapping / registration protocol. In deliverable D3.9 we will use the security-related meta-data when implementation the bootstrapping in the gateway.

The gateway uses secure communication when security token is provided. We rely on the standard implementation as provided by W3C WoT[49].

Requirement R.FD.12: Greenfield device is expected to expose its capability over a W3C Thing Description, which semantically describes field resources, and to be computationally powerful enough to run a node-wot servient (that exposes the TD).

This requirement has been addressed in the implementation of the gateway. For example, our devices: camera, microphone, and inclinometer are implemented on a Raspberry Pi, which runs a W3C WoT servient and exposes TD of these devices.

Requirement R.FD.13: Brownfield device is assumed to consist of a sensor/actuator and a controller (PLC). The controller is expected to expose capability of its sensor/actuator over a native brownfield protocol (without the need for IIoT Gateway to interact directly with them).

This requirement has been fulfilled in this version of the gateway implementation.

Requirement R.UC1.1: (automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders).

This is a requirement related to the requirement R.GP.1. Both of them deal with the establishment of end-to-end connectivity between different stakeholders, including field devices too. Semantic-based approach and communication interface enable a high degree of automation in the process of establishment of networking setup. We implemented the bootstrapping mechanism that enables plug and play.

Requirement R.UC1.8: Semantic and robust bootstrapping/registration of IIoT sensors and actuators with IIoT gateway MUST be supported.

[49] https://www.w3.org/TR/wot-thing-description/#sec-security-vocabulary-definition

This is a requirement that is the topic of this deliverable as whole. We distinguish the process of semantic bootstrapping/registration of brownfield- and greenfield devices. In this deliverable a concept for the bootstrapping process has been implemented, see Section 4.

Requirement R.UC1.9: Semantic interaction between use-case specific application on IIoT Gateway and legacy turbine control system MUST be supported.

In deliverable D3.3 we provided a standardized interface for interactions (based on W3C Web of Things), and semantically describe it with W3C Thing Description and iotschema.org. In this deliverable we implemented SEMIoTICS IoT Gateway, which enables semantic-based interaction between use-case specific application and legacy turbine control system.

Requirement R.UC1.10: Sufficient compute environment MUST be supported on the IIoT Gateway to run use-case specific applications.

The IIoT Gateway is running on Siemens SIMATIC IPC227E (Nanobox PC) industrial computer. This hardware features 1x Display-Port Grafik 2x 10/100/1000 MBit/s Ethernet RJ45 1x USB3.0, 3x USB2.0 CFAST-Slot DC 24V Industry-STROMVERS Celeron N2807 (2C/2T) 4 GB RAM, 80 GB SSD. As such this is a very powerful environment with enough compute power to run use-case specific applications.

Requirement R.UC1.11: Device composition and application creation SHALL be supported through template approach.

Task 3.3 is concerned with provision of semantic-based interfaces of devices. The Recipe Cooker component in the Cloud/Backend layer will support template-based creation of applications. Thanks to the semantic concept provided in this deliverable, it will be possible to orchestrate applications and devices based on application templates.

Requirement R.UC1.12: Standardized semantic models for semantic-based engineering and IIoT applications MUST be utilized.

In this deliverable we have proposed and used W3C Web of Things Thing Description (TD) as a standardized format for describing IoT things. TD is enriched with iotschema.org – a W3C semantic model that builds on standardized semantics.

Requirement R.UC1.13: Middleware functionality MUST be supported on IIoT gateway, to deal with termination of IIoT sensors, signal processing and termination of interfaces to legacy systems to provide prioritization and QoS for IIoT applications.

The IIoT Gateway is capable to either terminate connection to IIoT sensors or legacy systems.

Requirement R.UC2.5: The SEMIoTICS platform should allow the SARA solution to discover the IoT devices that are registered in the system. IoT devices deployed by the SARA solution are expected to register themselves into the system using various standard protocols (e.g. LwM2M, MQTT, Bluetooth LE, ZigBee, etc.).

In order to fulfill this requirement, the IoT Gateway features components Local Thing Directory, and Semantic API & Protocol Bindings, see Figure 15.

Requirement R.UC2.6: The SEMIoTICS platform SHOULD allow the SARA solution to retrieve the resources exposed by registered devices via their object model (i.e. a data structure wherein each element represents a resource, or a group of resources, belonging to a device). The SEMIoTICS platform SHOULD support at least the OMA LWM2M object model.

Semantically described resources can be retrieved from Local Thing Directory in JSON-LD format (see Section 3.1.3). For a structured description of an example device and its resources, see Figure 11.

Requirement R.UC3.2 - Requirement R.UC3.9: IHES Sensing unit shall be able to interface and register to the IHES Sensing gateway with a standard IP based (i.e. TCP transport) one to many M2M communication protocol to properly handle node registration and capabilities negotiation and IHES Sensing gateway shall support one to many standard IP based (i.e. TCP transport) M2M communication protocol to interface a number N of connecting Sensing units (e.g. broadcast type). An example of the (JSON) registration message from the IHES Sensing Unit to the IHES Sensing Gateway can be found in Figure 5.

Requirement R.UC3.12: IHES Sensing gateway shall be capable to run Linux (e.g. Raspbian OS) and standard graphics and browser libraries.

This requirement has been fulfilled with the deployment on an IoT gateway that runs on Linux and is deployed on Raspberry Pi 3.

Requirement R.UC3.13: IHES Sensing gateway should be able to support HTTP and standard protocols for cloud interfacing (e.g. to make IHES Local DB data available).

IHES Gateway supports HTTP trough the IHES LocalDB API and MQTT + WebSocket protocol thanks to the MQTT Broker.

Requirement R.UC3.14 - Requirement R.UC3.15: The specific M2M protocol adopted on UC3 is based on MQTT.

A MQTT broker service is available to dispatch messages between the IHES Sensing Gateway and its associated sensing units. A use case specific serialized message protocol is required to coordinate the gateway and its associated units and exchange data / events / anomalies between them. JSON is the preferred serialization format adopted. Some examples of the defined message protocol (in JSON format) dispatched through MQTT between IHES Sensing units and Sensing Gateway, can be found in Figure 5. This message protocol defined in task 3.3 will be deployed and validate as part of task 5.6 (Integration and validation of UC3 demonstrator).

Requirement R.UC3.16: Each registered IHES sensing unit should send to the IHES sensing gateway a keep alive signal at a specified time interval (e.g. 20 seconds) to notify the IHES gateway it is correctly working. The sensing gateway should detect by this mean any non-working sensing unit and reconfigure the system accordingly.

This feature will enable the specific UC3 dependability patterns and will be delivered as part of task 5.6 (Integration and validation of UC3 demonstrator).

Requirement R.UC3.17: Sensing units and IHES sensing gateway should share a common clock (i.e. global reference time), precise up to milliseconds, to properly classify events and data acquired during the processing.

This global reference time will be negotiated when a sensing unit node will join a given gateway. Internally the system will work scheduling activities according to this common global reference time. A common clock is required also to implement the local vs. global scenario declared in D2.2.

All R.UC3.X technical requirements will be demonstrated as part of the task 5.6, focused on implementing and validating using SEMIoTICS frameworks the UC3 demo scenario.

6 CONCLUSION

In the first version of this deliverable we have provided a draft of the semantic integration of the Field level into the SEMIoTICS architecture. In particular, this included semantics that aims to make brownfield devices form existing automation systems interoperable with newly bootstrapped IoT devices. In order to achieve this goal, we first provided a use case that motivates the role of semantic integration in SEMIoTICS project and identified goals to be achieved. Second, we reviewed existing technology blocks, including IoT standards, thereby defining a technology basis that will be used in the implementation of semantic Field level integration. Third, we provided a concept in a form of a semantic layer cake on how to integrate existing brownfield automation standards into new IoT semantic models. Finally, we have given the status of the current implementation of the work.

In the actual version of this work, we worked on the implementation of SEMIoTICS IoT Gateway as specified by the concept of the first deliverable. The gateway fulfils requirements at the Field level as specified in this deliverable. It provides the integrated device semantics at this level and enables bootstrapping and creation of new Edge applications. In particular, we automated to the large extend the process of bootstrapping of a device in SEMIoTICS platform. For greenfield device this process is completely performed in plug and play manner. On the other hand, for brownfield devices, a user still has to perform minimal manual work. This work includes provision of device-specific meta-data, which commonly can be exported from existing engineering stations (e.g., TIA Portal). A user may also optionally enrich a device description (Thing Description) with semantic vocabulary such as iotschema.org. After this step, the bootstrapping process for brownfield devices proceeds automatically.

The presented concept and implementation for a semantic gateway enables a quick integration of newly plugged devices with existing (automation) systems. This opens up a wide range of possibilities for creating new applications with the hardware that has not existed at the time of engineering an initial (automation) system. All integrated devices have been described with semantic (machine-interpretable) and standardized descriptions and made discoverable via semantic search. Our solution does not require expertise in semantic technologies. Itis designed to be easily used by engineering and IoT application developers via interactions over the Node-RED platform.

7 REFERENCES

1. W3C, “Web of Things Working Group Charter”: https://www.w3.org/2016/12/wot-wg-2016.html
2. W3C, “Web of Things (WoT) Architecture”, W3C Editor's Draft 21 January 2019, https://w3c.github.io/wot-architecture/
3. W3C, “Web of Things (WoT) Thing Description”, W3C Editor's Draft 02 February 2019, https://w3c.github.io/wot-thing-description/
4. iotschema.org, “Intro Materials”, https://github.com/iot-schema-collab/intro-materials/blob/master/iotschema-intro.md
5. W3C, “JSON-LD 1.1: A JSON-based Serialization for Linked Data”, W3C Working Draft 14 December 2018, https://www.w3.org/TR/json-ld11/
6. G. Hatzivasilis, I. Askoxylakis, and G. Alexandris, “The Interoperability of Things :,” no. September, 2018.

8 APPENDIX

8.1 Appendix A

Default values for indexing the unit: *Rack* (0..7) and *Slot* (1..31) that are to be found in the hardware configuration of a TIA Portal project (for a physical component).

CPU Type	Rack	Slot	Comment
S7 300 CPU	0	2	Always
S7 400 CPU	Not fixed		Follow the hardware configuration.
WinAC CPU	Not fixed		Follow the hardware configuration.
S7 1200 CPU	0	1	
S7 1500 CPU	0	1	
WinAC IE	0	0	Or follow Hardware configuration.

Figure 40: Deafault values for Rack and Slot in Siemens S7

8.2 Appendix B

Data types used in S7comm with corresponding sizes in bits.

Type and Description	Size in Bits	Format Options
BOOL (Bit)	1	Boolean text
BYTE (Byte)	8	Hexadecimal number
WORD (Word)	16	Binary number
		Hexadecimal number
		BCD
		Decimal number unsigned
DWORD (Double word)	32	Binary number
		Hexadecimal number
		Decimal number unsigned
INT (Integer)	16	Decimal number signed
DINT (Double integer)	32	Decimal number signed
REAL (Floating-point number)	32	IEEE Floating-point number
S5TIME (SIMATIC time)	16	S7 time in steps of 10ms (default)
TIME (IEC time)	32	IEC time in steps of 1 ms, integer signed
DATE (IEC date)	16	IEC date in steps of 1 day

TIME _OF_DAY (Time)	32	Time in steps of 1 ms
CHAR (Character)	8	ASCII characters

Figure 41: Step 7 Elementary Data Types

8.3 Appendix C

Addressing schema for variables as expected in SEMIoTICS Mediator and the implementation of Protocol Binding.

Address	Step7 equivalent	JS Data type	Description
DB5,X0.1	DB5.DBX0.1	Boolean	Bit 1 of byte 0 of DB 5
DB23,B1 or DB23,BYTE1	DB23.DBB1	Number	Byte 1 (0-255) of DB 23
DB100,C2 or DB100,CHAR2	DB100.DBB2	String	Byte 2 of DB 100 as a Char
DB42,I3 or DB42,INT3	DB42.DBW3	Number	Signed 16-bit number at byte 3 of DB 42
DB57,WORD4	DB57.DBW4	Number	Unsigned 16-bit number at byte 4 of DB 57
DB13,DI5 or DB13,DINT5	DB13.DBD5	Number	Signed 32-bit number at byte 5 of DB 13
DB19,DW6 or DB19,DWORD6	DB19.DBD6	Number	Unsigned 32-bit number at byte 6 of DB 19
DB21,R7 or DB21,REAL7	DB21.DBD7	Number	Floating point 32-bit number at byte 7 of DB 21
DB2,S7.10*	-	String	String of length 10 starting at byte 7 of DB 2
I1.0 or E1.0	I1.0 or E1.0	Boolean	Bit 0 of byte 1 of input area
Q2.1 or A2.1	Q2.1 or A2.1	Boolean	Bit 1 of byte 2 of output area
M3.2	QM3.2	Boolean	Bit 2 of byte 3 of memory area
IB4 or EB4	IB4 or EB4	Number	Byte 4 (0 -255) of input area
QB5 or AB5	QB5 or AB5	Number	Byte 5 (0 -255) of output area
MB6	MB6	Number	Byte 6 (0 -255) of memory area
IC7 or EC7	IB7 or EB7	String	Byte 7 of input area as a Char
QC8 or AC8	QB8 or AB8	String	Byte 8 of output area as a Char

Address	Step7 equivalent	JS Data type	Description
MC9	MB9	String	Byte 9 of memory area as a Char
II10 or EI10	IW10 or EW10	Number	Signed 16-bit number at byte 10 of input area
QI12 or AI12	QW12 or AW12	Number	Signed 16-bit number at byte 12 of output area
MI14	MW14	Number	Signed 16-bit number at byte 14 of memory area
IW16 or EW16	IW16 or EW16	Number	Unsigned 16-bit number at byte 16 of input area
QW18 or AW18	QW18 or AW18	Number	Unsigned 16-bit number at byte 18 of output area
MW20	MW20	Number	Unsigned 16-bit number at byte 20 of memory area
IDI22 or EDI22	ID22 or ED22	Number	Signed 32-bit number at byte 22 of input area
QDI24 or ADI24	QD24 or AD24	Number	Signed 32-bit number at byte 24 of output area
MDI26	MD26	Number	Signed 32-bit number at byte 26 of memory area
ID28 or ED28	ID28 or ED28	Number	Unsigned 32-bit number at byte 28 of input area
QD30 or AD30	QD30 or AD30	Number	Unsigned 32-bit number at byte 30 of output area
MD32	MD32	Number	Unsigned 32-bit number at byte 32 of memory area
IR34 or ER34	IR34 or ER34	Number	Floating point 32-bit number at byte 34 of input area
QR36 or AR36	QR36 or AR36	Number	Floating point 32-bit number at byte 36 of output area
MR38	MR38	Number	Floating point 32-bit number at byte 38 of memory area